CORE DATA & SWIFTUI MASTERY

THE ULTIMATE GUIDEBOOK

FIRST EDITION

KARIN PRATER

Introduction

Welcome to “SwiftUI & Core Data Mastery: The Ultimate Guidebook.” This book is designed to guide you through the process of building apps with the powerful combination of SwiftUI and Core Data. Before we dive into the content, let’s set the stage for what you’ll need to get the most out of this book.

Prerequisites

To make your learning experience smooth and productive, you should have a basic understanding of Swift and familiarity with SwiftUI. If you’ve built a simple app or two with SwiftUI or have gone through some basic tutorials, you’re likely ready to take on the concepts we’ll cover in this book.

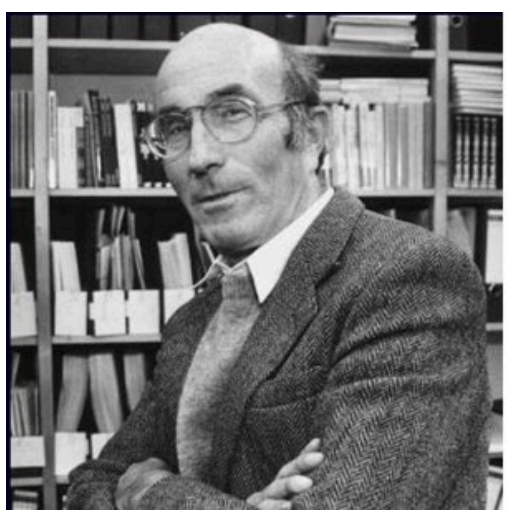
What You’ll Need

Here’s what you should have on hand to follow along with the tutorials and examples presented:

- **A Mac running macOS Ventura (13.0) or later**: The operating system is essential for compatibility with the latest tools and features.
- **Xcode 14 or later**: Make sure you have the latest version installed to avoid any hiccups.
- **An Apple Developer account**: You will need this to access certain resources and capabilities within Xcode. While a free account will allow you to develop and test your apps, consider a paid account if you plan to distribute your apps on the App Store. You can sign up for an Apple Developer account here.

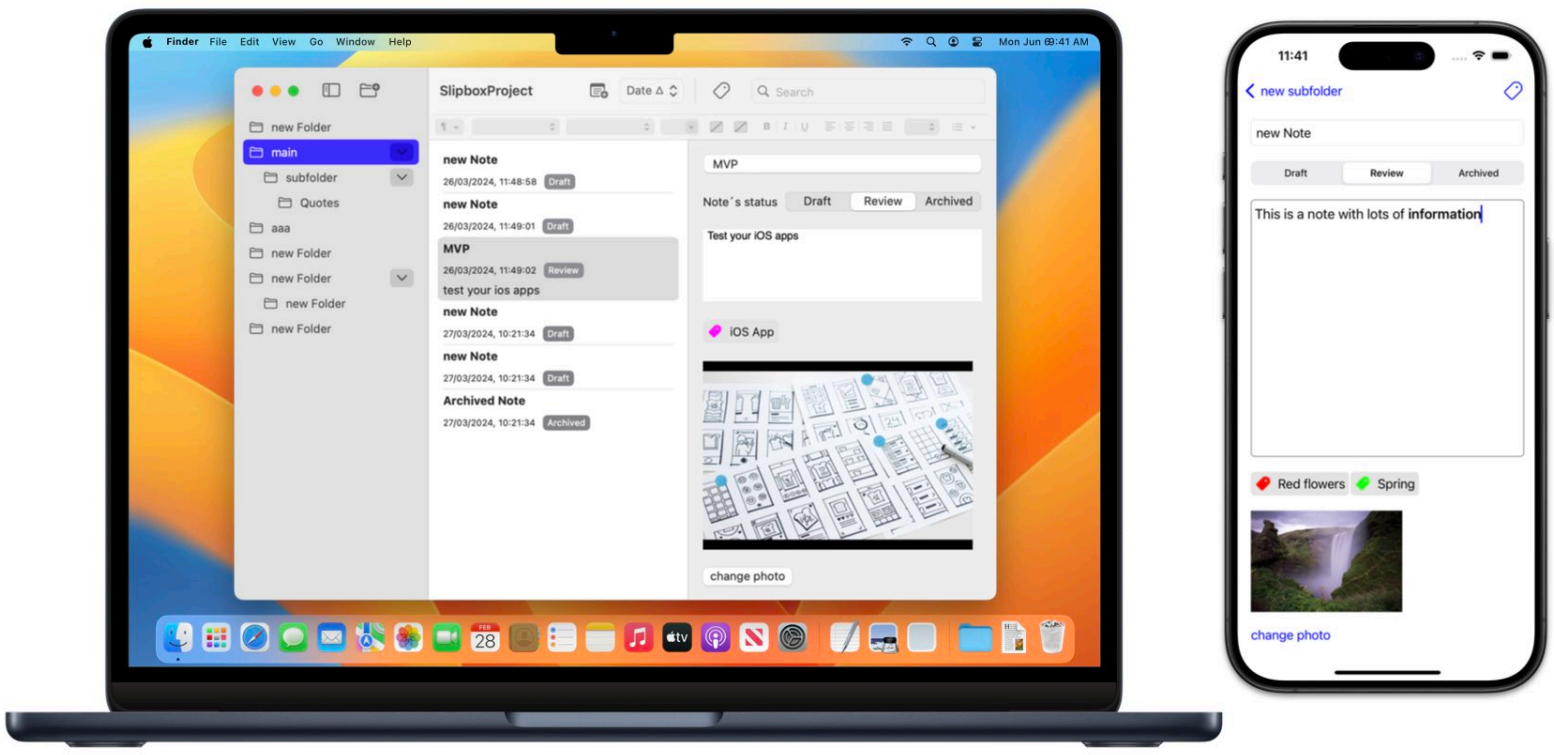
Project-Based Learning

In this project-based book, we will be creating a note-taking app inspired by the note-taking system of German sociologist Lukas Luhmann. Luhmann's unique approach allowed him to publish an impressive number of books and articles by organizing his thoughts using a slip box, or "Zettelkasten" in German. Each note in his system contained links to other related notes, forming a chain of interconnected thoughts.



Our goal is to build a similar note-taking system with various categorizations and features. We will explore how to structure notes into folders, assign different characteristics to each note, and implement functionalities such as rich text editing, keyword tagging, sorting, filtering, and searching.

The final project we will be working on is a cross-platform app, primarily focused on iOS. However, we will also explore how the app looks and functions on macOS, as it can provide valuable insights and a broader perspective.



The app's interface will feature a hierarchical folder structure, allowing you to organize your notes efficiently. Within each folder, you will be able to view notes with their corresponding attributes, such as creation timestamps, tags, and status (draft, review, or archived). Additionally, we will incorporate a rich text editor, enabling you to format your notes with different styles and even include images.

Book Outline

To help you navigate through the book and make the most of your learning experience, let's take a closer look at the sections and topics we will cover:

Section 1: Project Setup and Basics

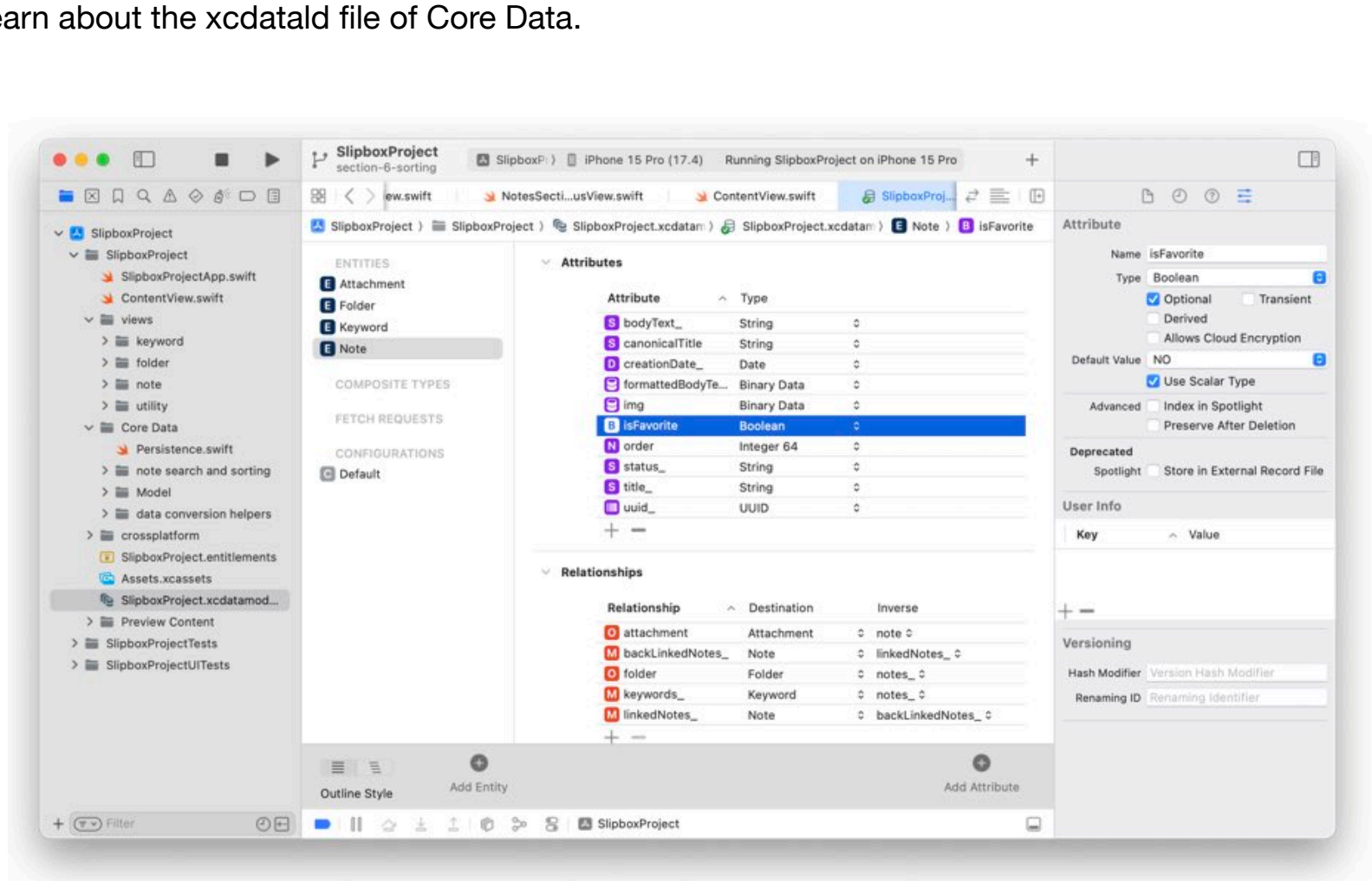
In this section, we will start by setting up the project with Core Data integration in Xcode. We will also explore how to enable iCloud sync to facilitate data synchronization across multiple devices. Additionally, we will discuss deployment strategies for releasing your app to production with iCloud sync. Finally, we will dive into the fundamentals of saving, storing, and retrieving data using Core Data.

Section 2: Unit Testing and Test-Driven Development

Unit testing is a crucial aspect of software development, and in this section, we will focus on writing comprehensive unit tests for our Core Data functionalities. By decoupling the testing process from the user interface, we can efficiently test and validate critical components of our app's functionality. You will also explore the concept of test-driven development, where we continuously write tests to support our project and minimize the introduction of bugs.

Section 3: Schema and Data Modelling

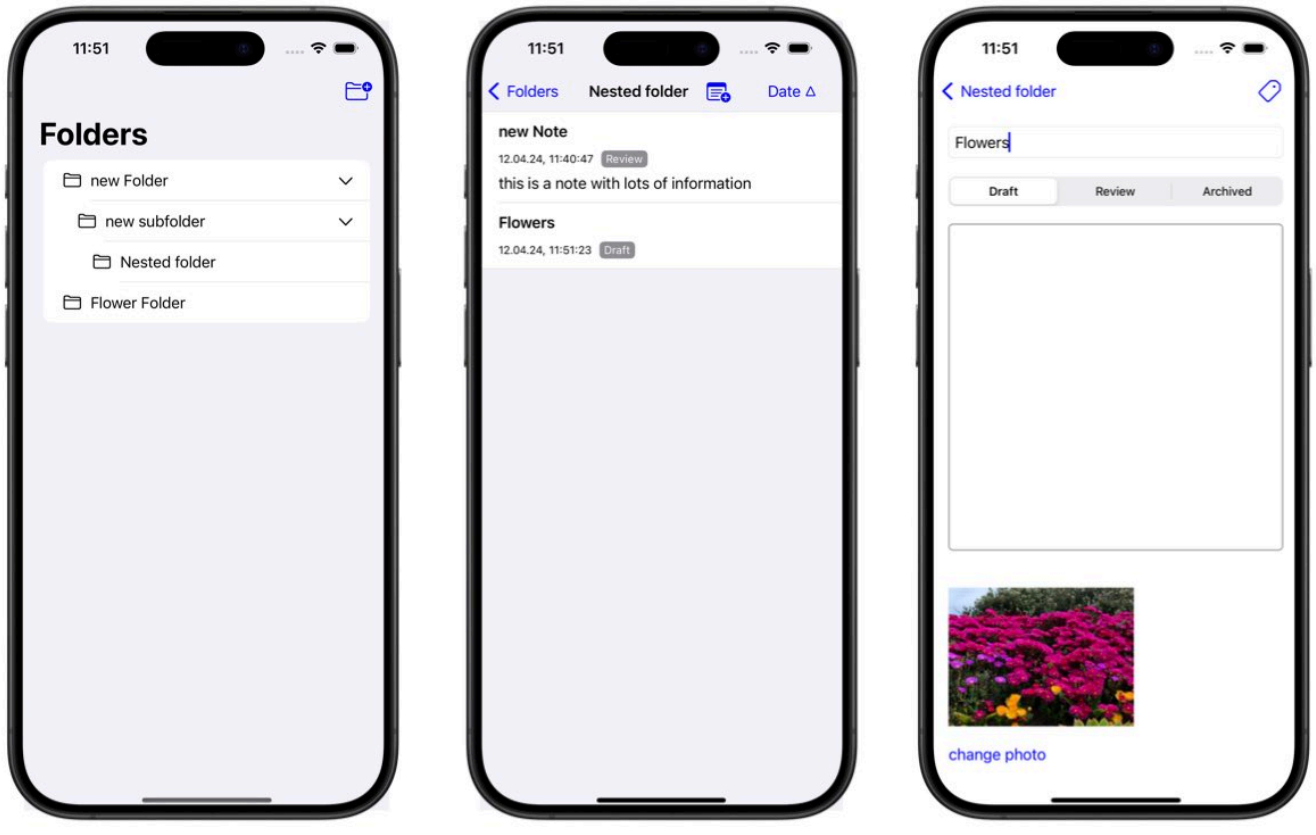
In this important section, we will discuss how to design and describe our data models effectively. You will learn about the xcdatald file of Core Data.



We will explore various scenarios, such as storing enums, texts, images, and custom types like colors. Although Core Data imposes certain restrictions on data storage, understanding how to map and describe our data models will provide a solid foundation for working with other systems and platforms.

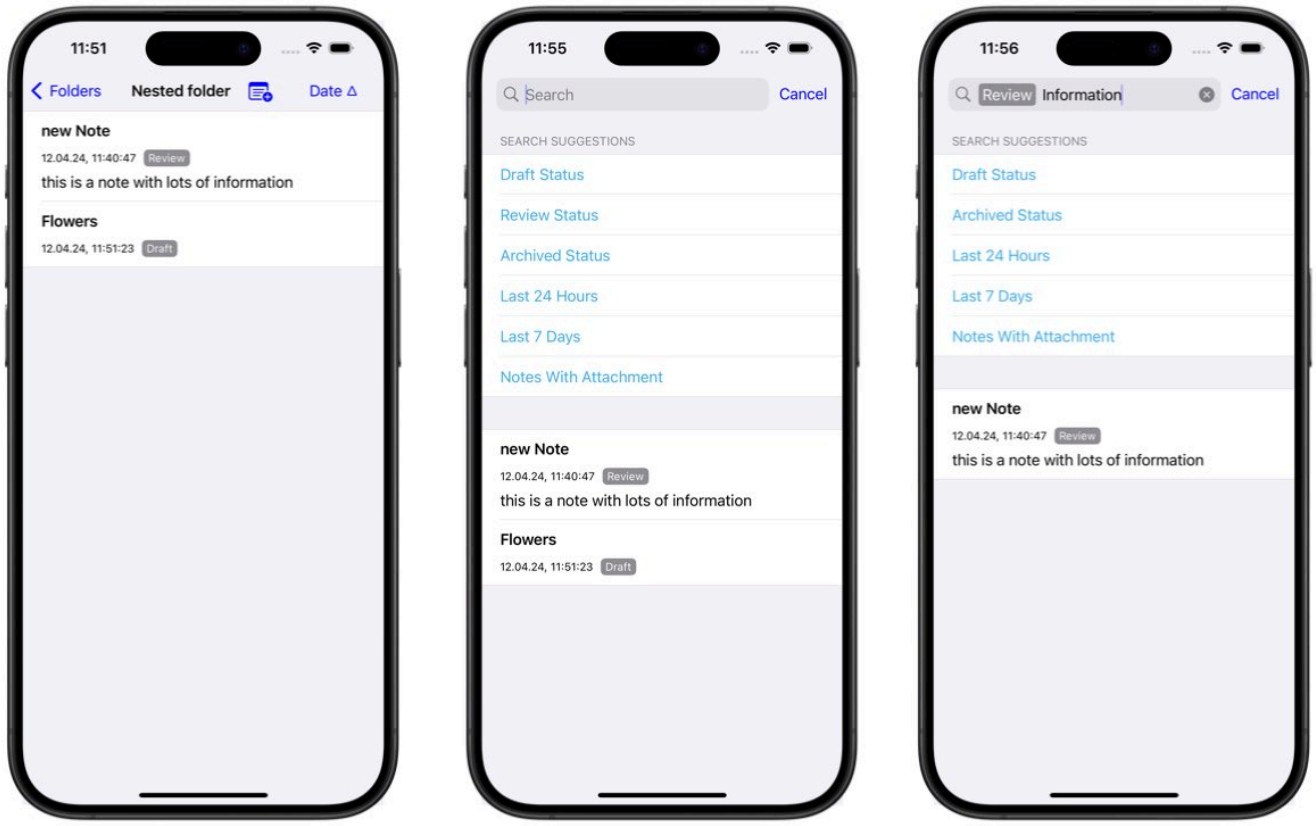
Section 4: Relationships and Entity Linking

Managing relationships between different data types is a crucial aspect of Core Data. We will delve into creating and managing relationships, such as linking folders to notes and adding keywords to notes. Additionally, we will explore the concept of subfolders and nested folder structures. We will also cover how to present and update the user interface to reflect these relationships accurately. Lastly, we will touch on optimization techniques, including background image processing.



Section 5: Fetching and Filtering

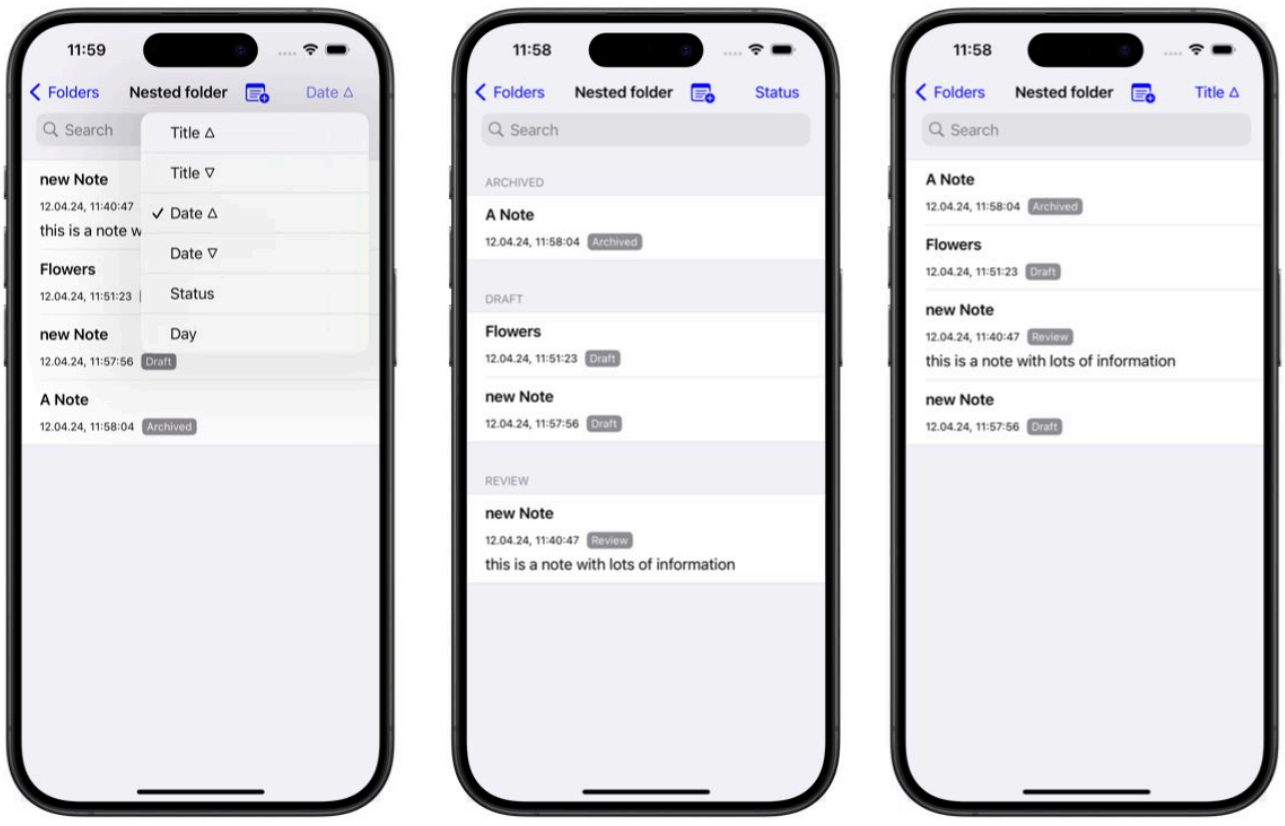
This section focuses on retrieving specific data from Core Data based on various criteria. We will explore how to use NSPredicates to describe and implement complex filtering systems. Through unit tests, you



will learn how to fetch notes based on specific values, date ranges, relationships, and text fields. We will also begin integrating these filtering functionalities into the user interface, starting with the keyword list.

Section 6: Advanced Searching and Sorting

In this final section, we will dive deeper into searching and sorting functionalities. We will combine the complex information flow required for searching and sorting, ensuring that our app updates the displayed data accurately. We will cover different sorting options, modify the list view to reflect the chosen sorting criteria, and implement advanced searching with tokens. By mastering these techniques, you will have a solid understanding of the main building blocks of Core Data.



This book is designed to be followed sequentially, especially if you are new to Core Data. However, if you are already familiar with the basics, feel free to jump ahead to the sections that interest you the most. The primary objective is to provide you with a comprehensive understanding of Core Data through a larger, more complex project. By the end of this book, you will have the knowledge and skills to build your own apps with Core Data, even with complex data structures.

Let’s get started on this exciting journey of mastering Core Data with SwiftUI!

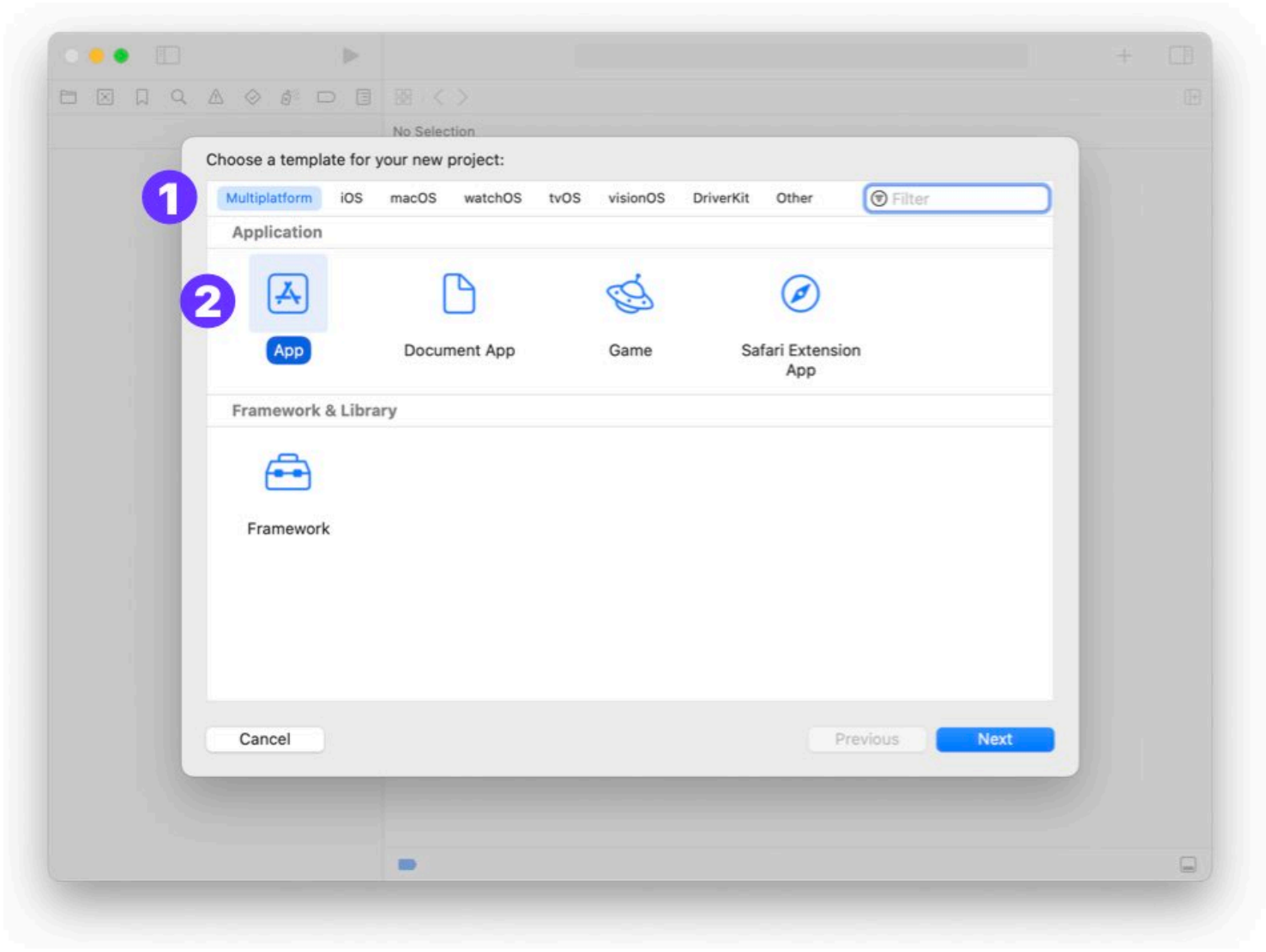
1. SETTING UP THE PROJECT

1.1 SETUP

In this chapter, I'll guide you through the process of creating a new project for CoreData. I will give you a quick breakdown of the generated files that work with CoreData. In later sections, I will give you more detailed information about each of these components.

Creating a New Project

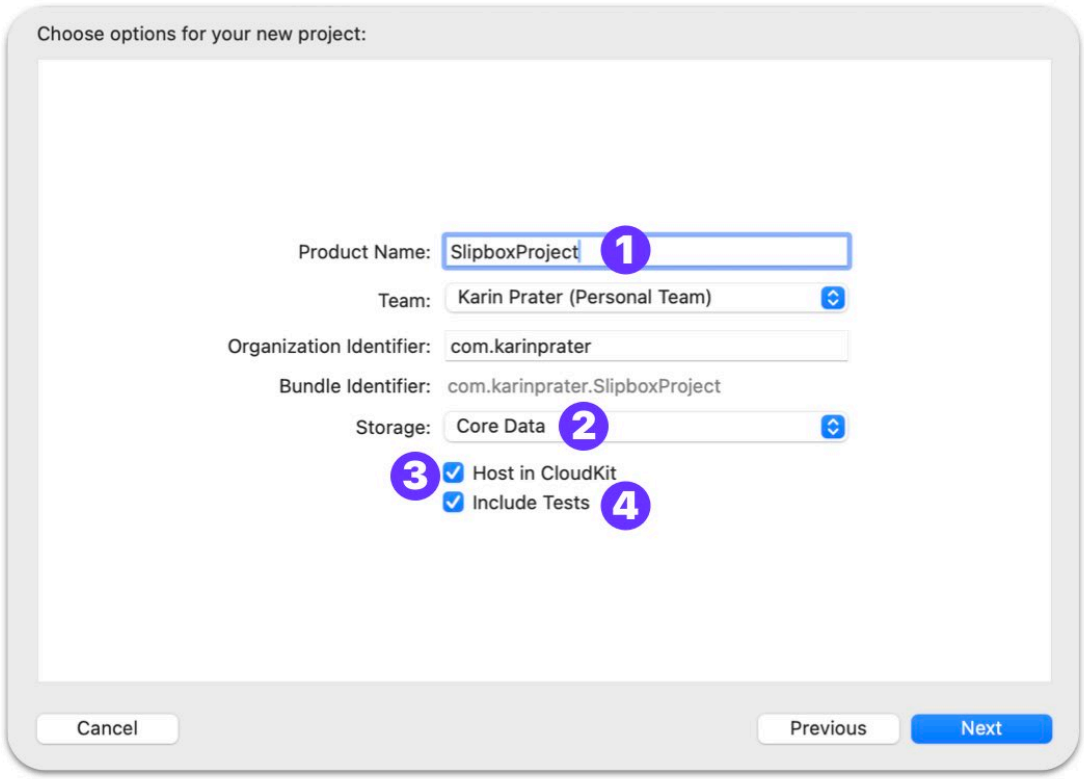
To begin, let's create a new project for Core Data. Open Xcode and select "Multiplatform App" as the project template.



If you plan to add subscriptions in the future, choosing a multi-platform project allows for support of **universal subscription purchases**. That means if the user pays for a subscription on iOS, he/she can use the same subscription for macOS and vice versa.

Next, Name your project e.g. "SlipBoxProject".

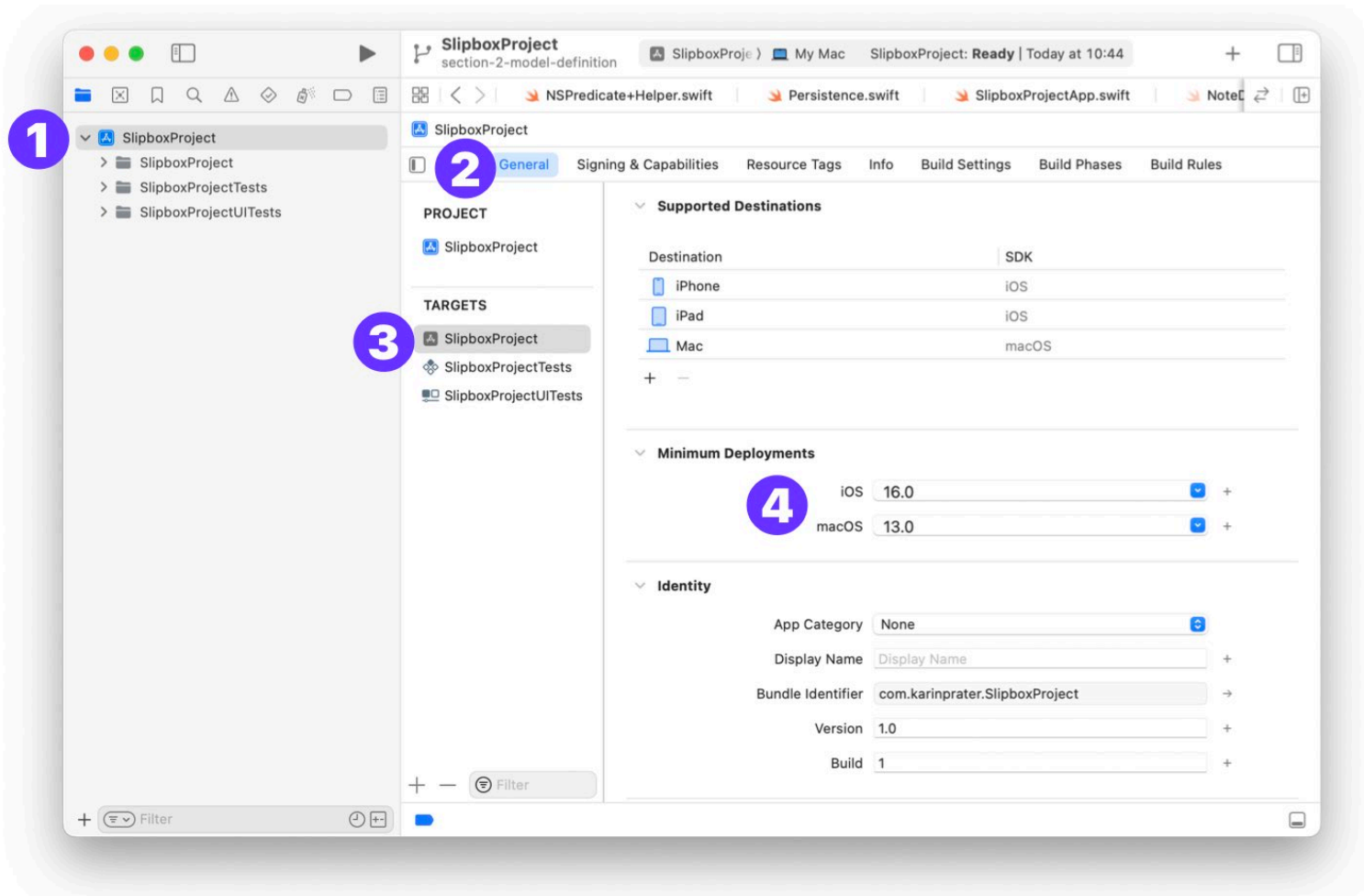
Make sure to check the **(2) "Use CoreData"** and **(3) "Use CloudKit"** options. Note that to fully test CloudKit functionality, you'll need a paid developer account. However, the project will still compile and run without one; you just won't see the syncing capabilities in action.



Additionally, enable the **(4) "Include Tests"** option as we will be testing our Core Data code and business logic.

Project Configuration

Upon creating the project, Xcode adds a bunch of default code. If you navigate to the project settings **(1)** under the target's General tab **(2 + 3)**, you'll see that, since I chose a multi-platform app, there are already destinations set for iPhone, iPad, and Mac.

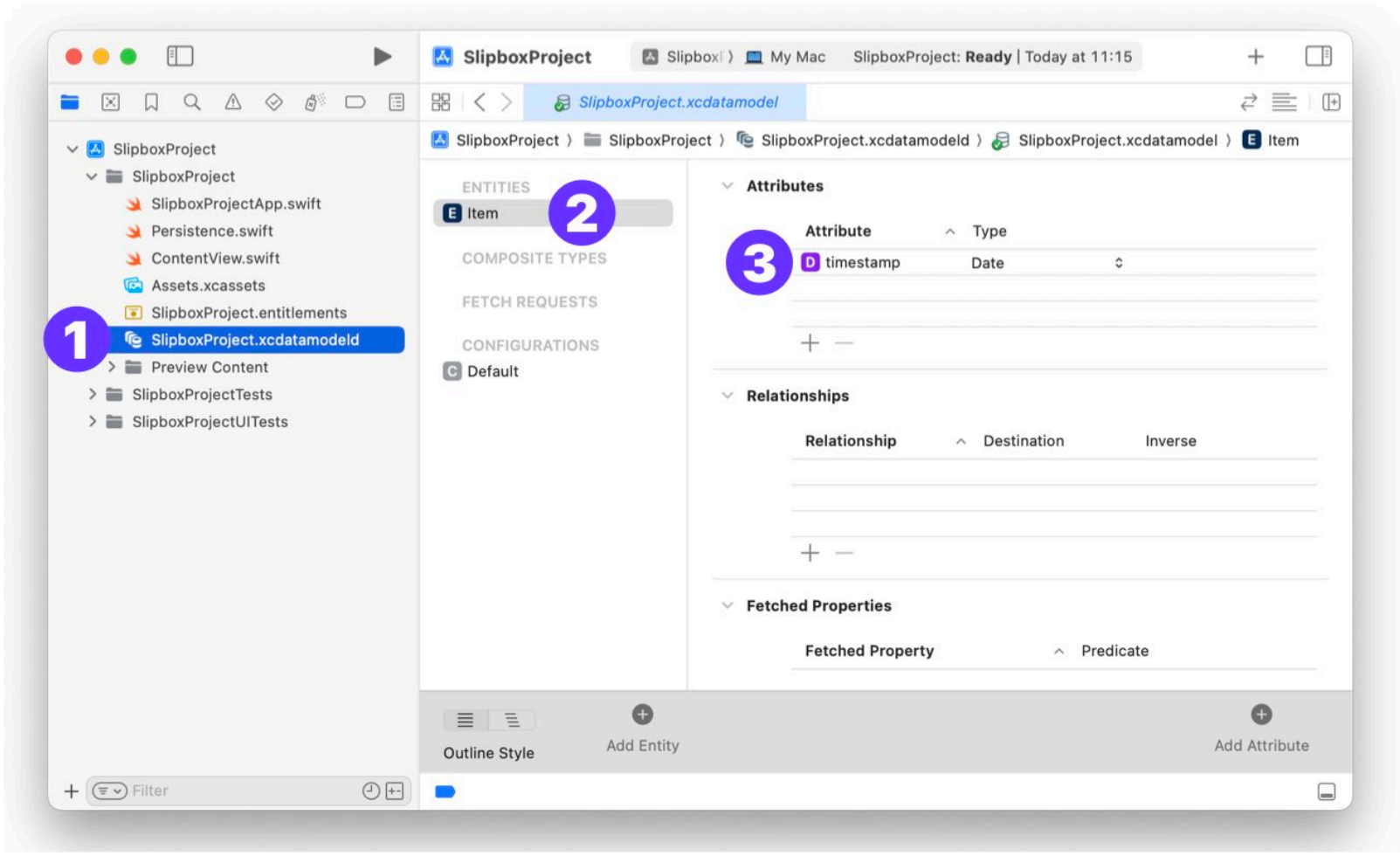


(4) Set your minimum deployment targets according to your needs. I'm going with iOS 16 and macOS 13, and I'm using Xcode version 15.3.

Exploring the Generated Files

Let's take a look at the generated files. There are two folders for tests, but I'll focus on the main files for now. The CoreData model file, "**SlipBox.xcdatamodeld**" **(1)**, is where we define our data schema. The file type is **xcdatamodeld**. The name of the file is the same as your project name.

The autogenerated project includes a generic **(2) "Item" entity** with a **"timestamp" attribute of type Date (3)**.



To define our own data structure, we can add entities and attributes to the model file. We can choose from various attribute types such as integer, string, bool, data, and UUID. Additionally, we can define relationships between entities and declare fetched properties.

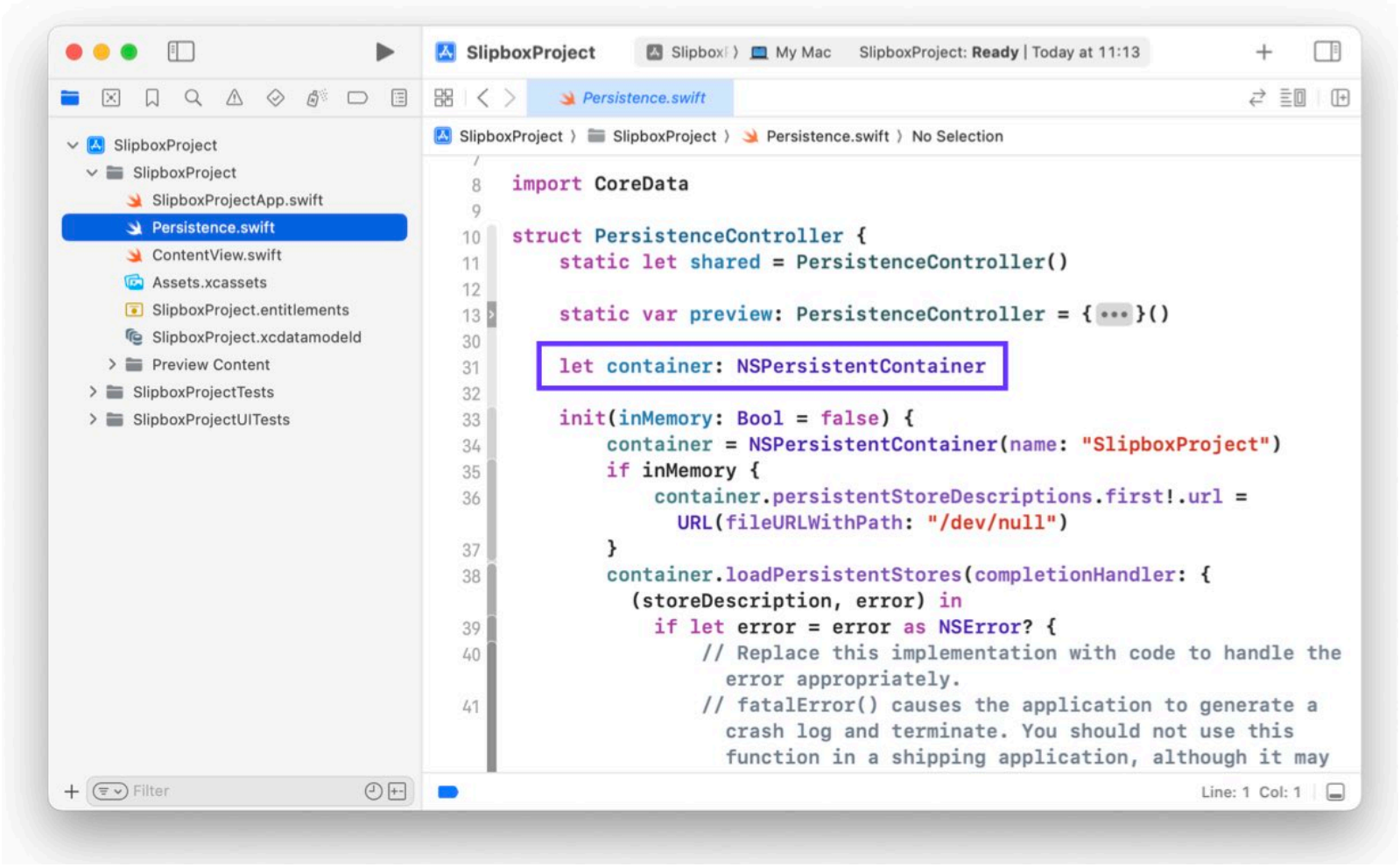
Section 3 will teach you about how to define the schema attributes and in section 4 you will learn about relationships in detail.

The **"xcdatamodeld"** file is crucial for Core Data as it specifies the data model used by our application.

CoreData Stack

CoreData uses a stack of objects to manage the data model, and Xcode provides a convenience file called “Persistence.swift”. This file sets up the CoreData stack, including the persistent container, which is crucial for the CoreData operations.

It is a singleton that provides access to the Core Data stack and manages the file containing the data.



The initializer of PersistenceController has an argument for “inMemory”. In some cases like testing, we don’t want to save data to a file. For these situations, you can use the inMemory. The URL of the permitted file is set to a temporary environment.

It also adds a preview property that helps to work with SwiftUI previews:

```
struct PersistenceController {
    static let shared = PersistenceController()

    static var preview: PersistenceController = {
        let result = PersistenceController(inMemory: true)
        let viewContext = result.container.viewContext
        for _ in 0..<10 {
            let newItem = Item(context: viewContext)
            newItem.timestamp = Date()
        }
         …
        return result
    }()

    …
}
```

For example, you can see it used in ContentView like:

```
    #Preview {
        ContentView().environment(\.managedObjectContext,
                                  PersistenceController.preview.container.viewContext)
    }
```

Integrating CoreData with a SwiftUI Project

To integrate CoreData with SwiftUI, I'll need to inject the managed object context into the SwiftUI environment. This is done in the main App file using the .environment(\.managedObjectContext, ...) modifier.

```
    @main
    struct SlipboxProjectApp: App {
        let persistenceController = PersistenceController.shared

        var body: some Scene {
            WindowGroup {
                ContentView()
                    .environment(\.managedObjectContext,
                                 persistenceController.container.viewContext)
            }
        }
    }
```

Showing Data from CoreData in SwiftUI

To work with Core Data in SwiftUI, we need to access the view context, which is responsible for creating, deleting, and manipulating entities. The "ContentView" file showcases how to access the view context and perform operations on our data. It includes a list of items and allows us to add new items and navigate to a detail view.

```
    import SwiftUI
    import CoreData.

    struct ContentView: View {
        // accessing CoreData viewContext that you can use e.g. to create new data
        @Environment(\.managedObjectContext) private var viewContext

        // reading data from CoreData
        @FetchRequest(
            sortDescriptors: [NSSortDescriptor(keyPath: \Item.timestamp, ascending:
    true)],
            animation: .default)
        private var items: FetchedResults<Item>

        var body: some View {
```

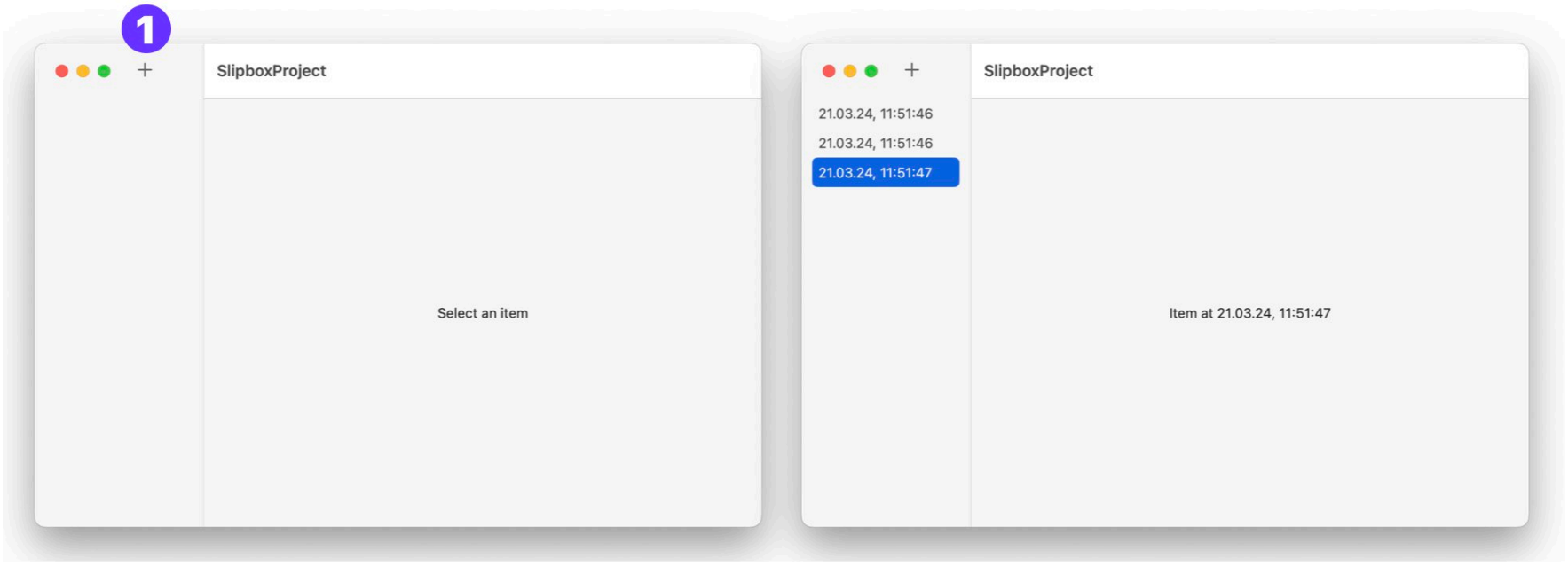
```
            NavigationView {
                List {
                    ForEach(items) { item in
                        NavigationLink { … }
                    }
                    .onDelete(perform: deleteItems)
                }
                .toolbar {
                    ToolbarItem {
                        Button(action: addItem) {
                            Label("Add Item", systemImage: "plus")
                        }
                    }
                }
                Text("Select an item")
            }
        }

        // adding new data from CoreData
        private func addItem() {
            let newItem = Item(context: viewContext)
            newItem.timestamp = Date()
        }

        // deleting data from CoreData
        private func deleteItems(offsets: IndexSet) {
            offsets.map { items[$0] }.forEach(viewContext.delete)
        }
    }
```

Fetching data is done using the **@FetchRequest** property wrapper, which works seamlessly with SwiftUI. Here's how you can fetch and sort items by timestamp. I will talk more about these operations in section 1.3 Note Model and CRUD

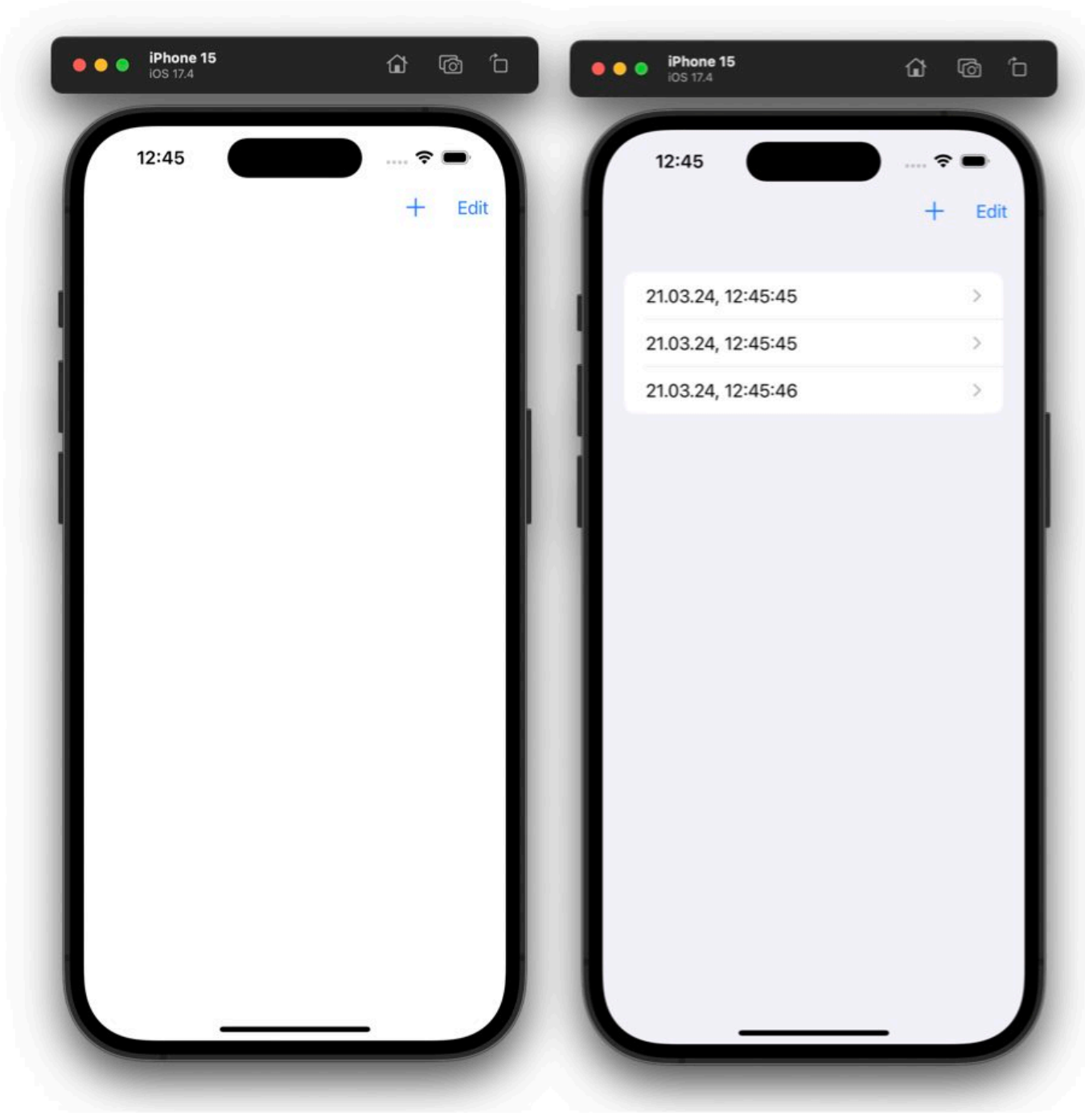
Testing The Template Project

The template project provides a very basic app. You can run the app on macOS. The window shows a 2-column layout. It is empty because we did not add any data yet.

Press the "+" button in the toolbar. A new item is shown with the timestamp:



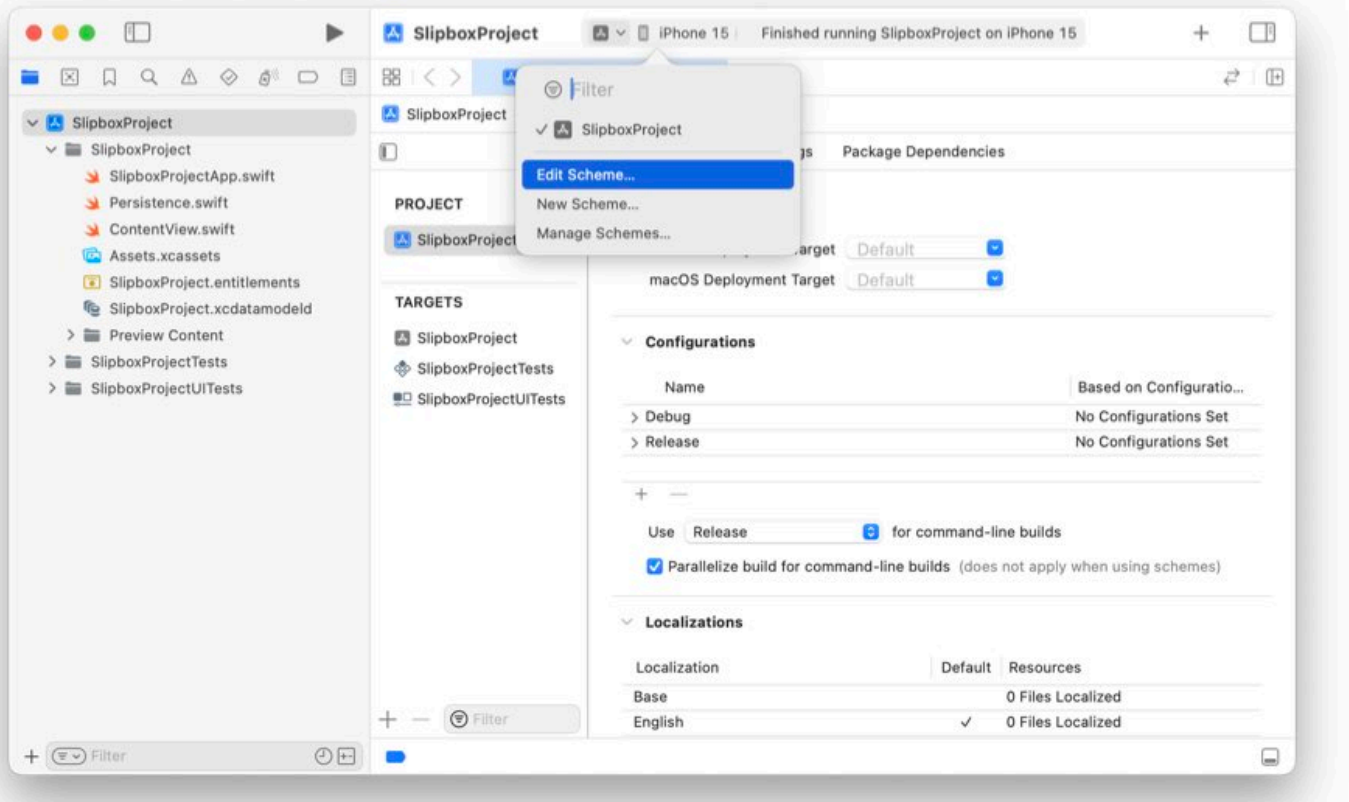
You can build and run the app also for iOS. Again the screen is initially empty because the data storage is empty. For a real-world app, you would load example data the first time the app launches. This would be a better user experience, but we will not do this for now since we first need to learn about data modelling and fetching.



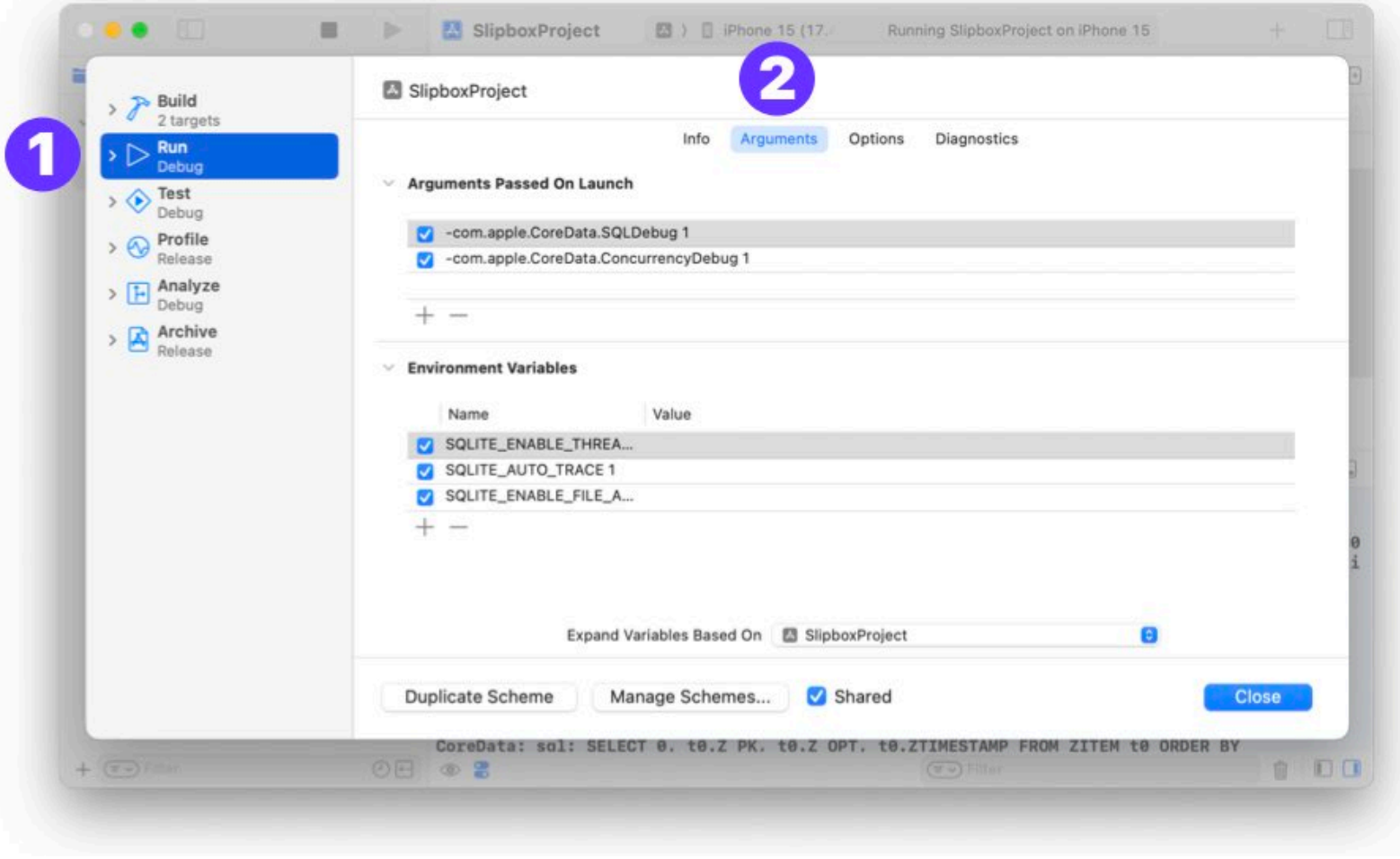
Debugging CoreData

If you want to know what CoreData does internally, you can use launch arguments and environment variables to your Xcode schemes to catch and debug Core Data problems.

To configure a scheme, select it in the Xcode toolbar and then choose **"Edit Scheme”** from the scheme menu in the Xcode toolbar.



Select the **run action (1)** in the left column and switch to the **Arguments tab (2)**. You can then use the “+” buttons to add launch arguments and environment variables:



Add two launch arguments:

- **com.apple.CoreData.SQLDebug 1** You can increase the debug level up to level 4. Set it to at least level 3 to see the data returned by a query. This can create a lot of debug output. Apple recommends leaving this argument disabled when you add it to your scheme. You can then quickly enable it when needed and turn it off again afterwards.

- **com.apple.CoreData.ConcurrencyDebug 1** Enabling this launch argument causes your App (or unit test) to throw an exception if you mistakenly access a managed object on the wrong queue.

Add three environment variables:

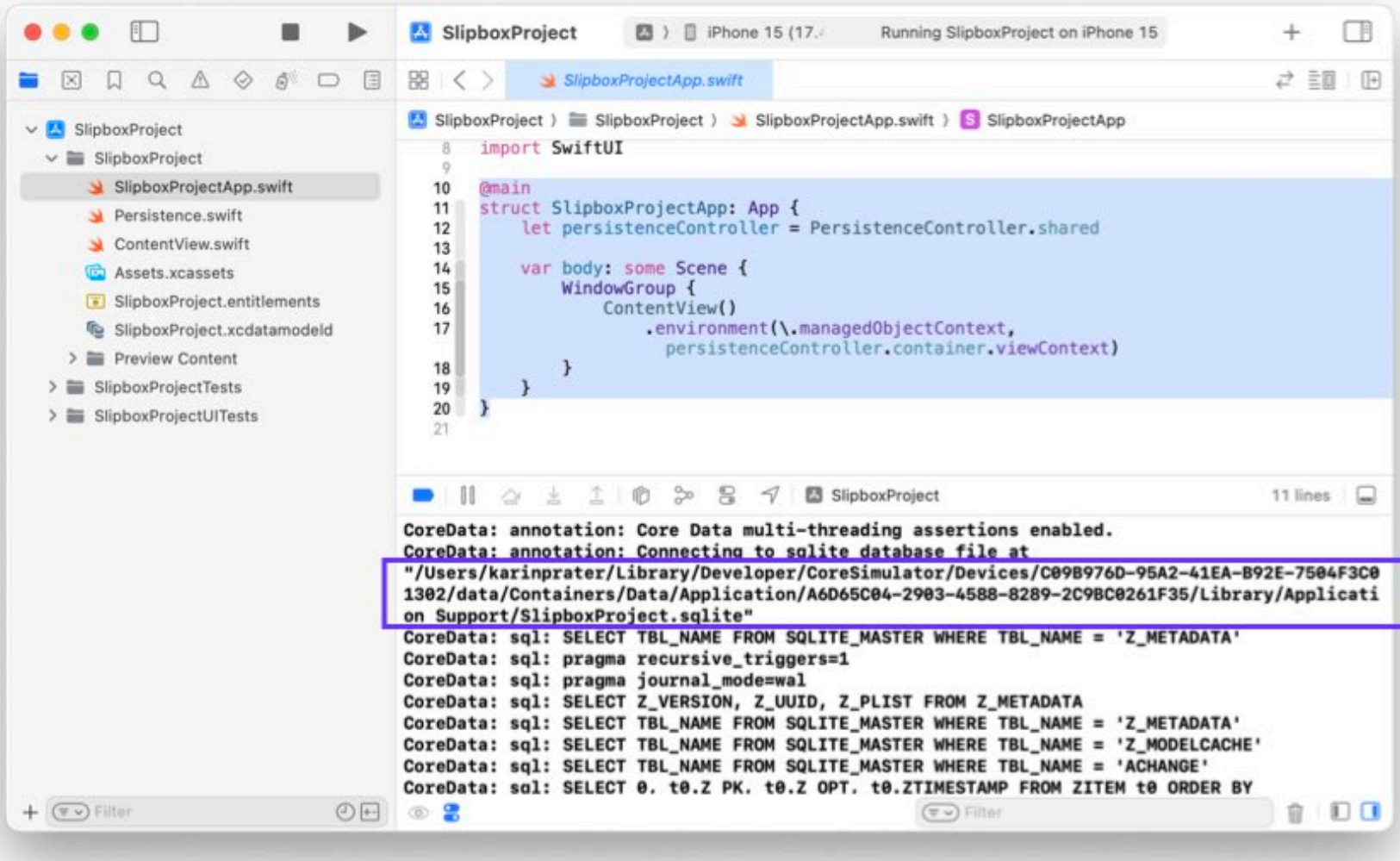
- **SQLITE_ENABLE_THREAD_ASSERTIONS 1**

- **SQLITE_AUTO_TRACE 1**

- **SQLITE_ENABLE_FILE_ASSERTIONS 1**

You can watch this WWDC video for all the details.

Now build and run your project again. You will see a lot of information showing in the Xcode debug area. For example, you will see the file URL of the data storage file:



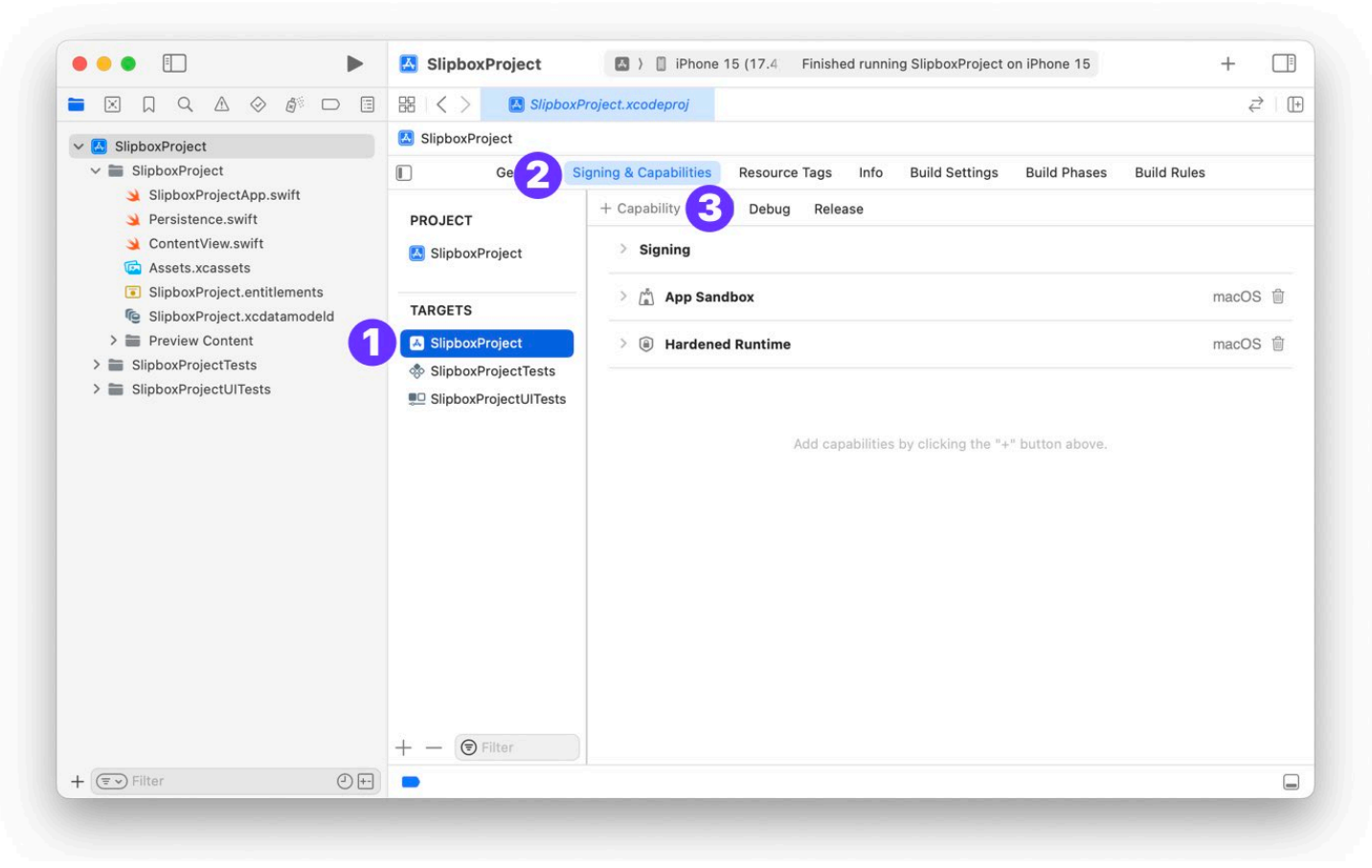
1.2 ICLOUD SYNC

In this section, we will explore how to integrate iCloud sync with Core Data in your SwiftUI project. This allows you to share data across different devices so that the user can see data from the Mac on iOS and vice versa.

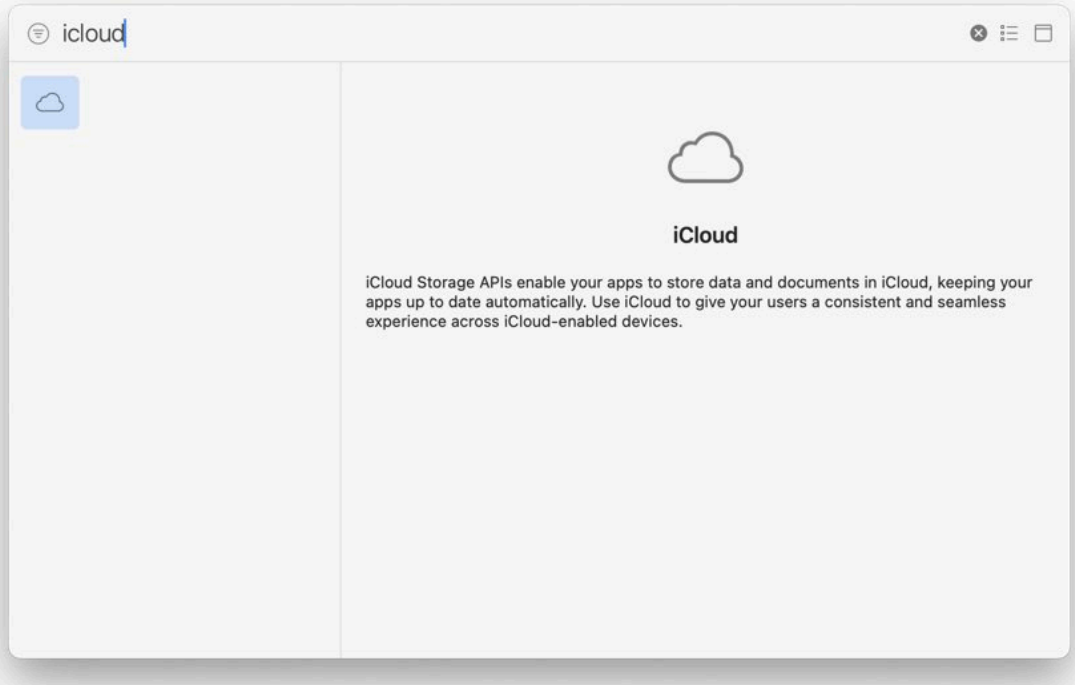
Enabling iCloud sync requires a few adjustments and configuration changes, particularly in the capabilities section of your project settings. Additionally, we will take a look at the iCloud dashboard, understand the workflow, and discuss common issues related to iCloud sync.

Capabilities and Developer Account

Now, let's navigate to the target settings of our project and go to the "Capabilities" tab. Tap on the **"+ Capabilites" button (3).**

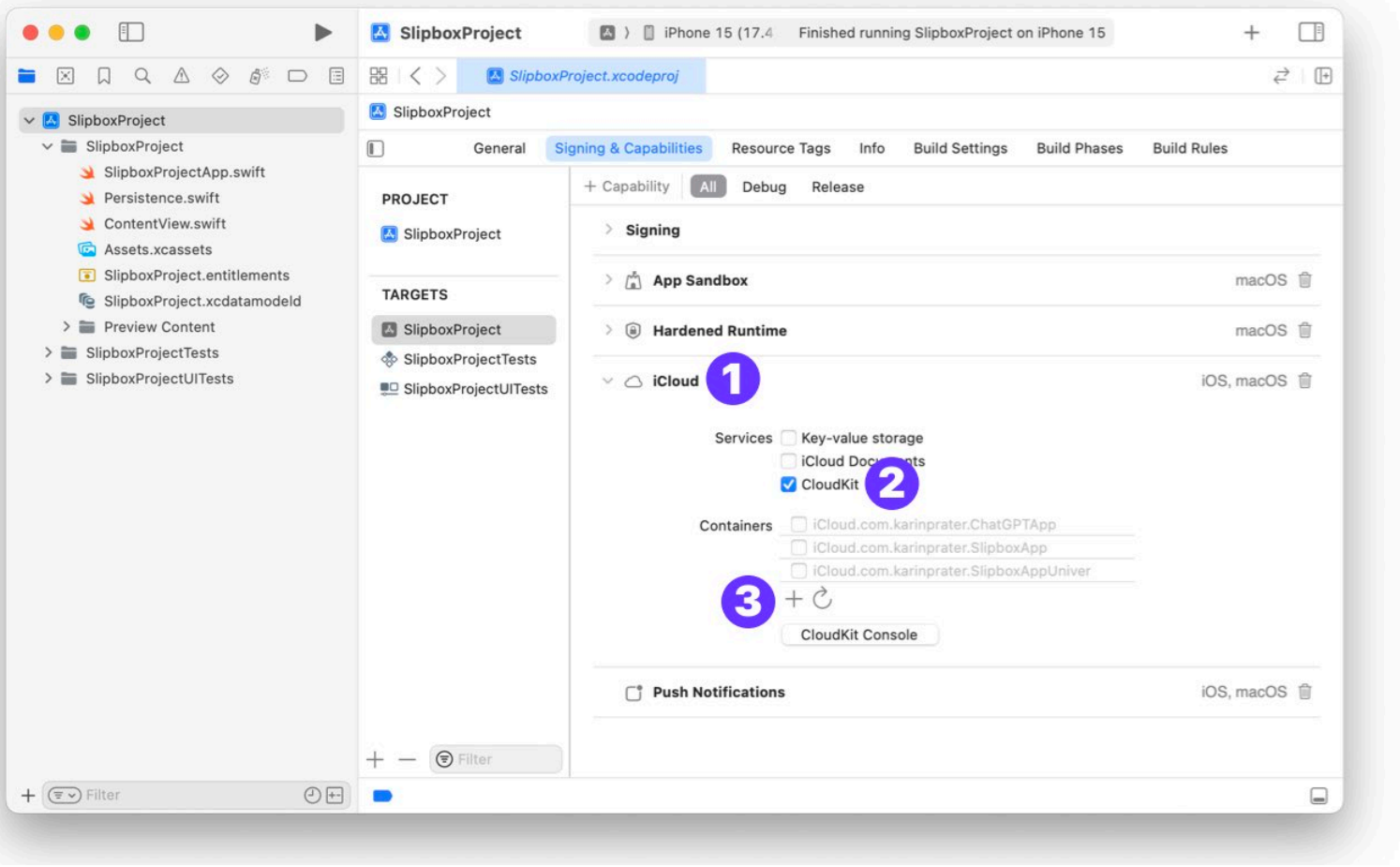


In the editor search for iCloud and add it to your project:

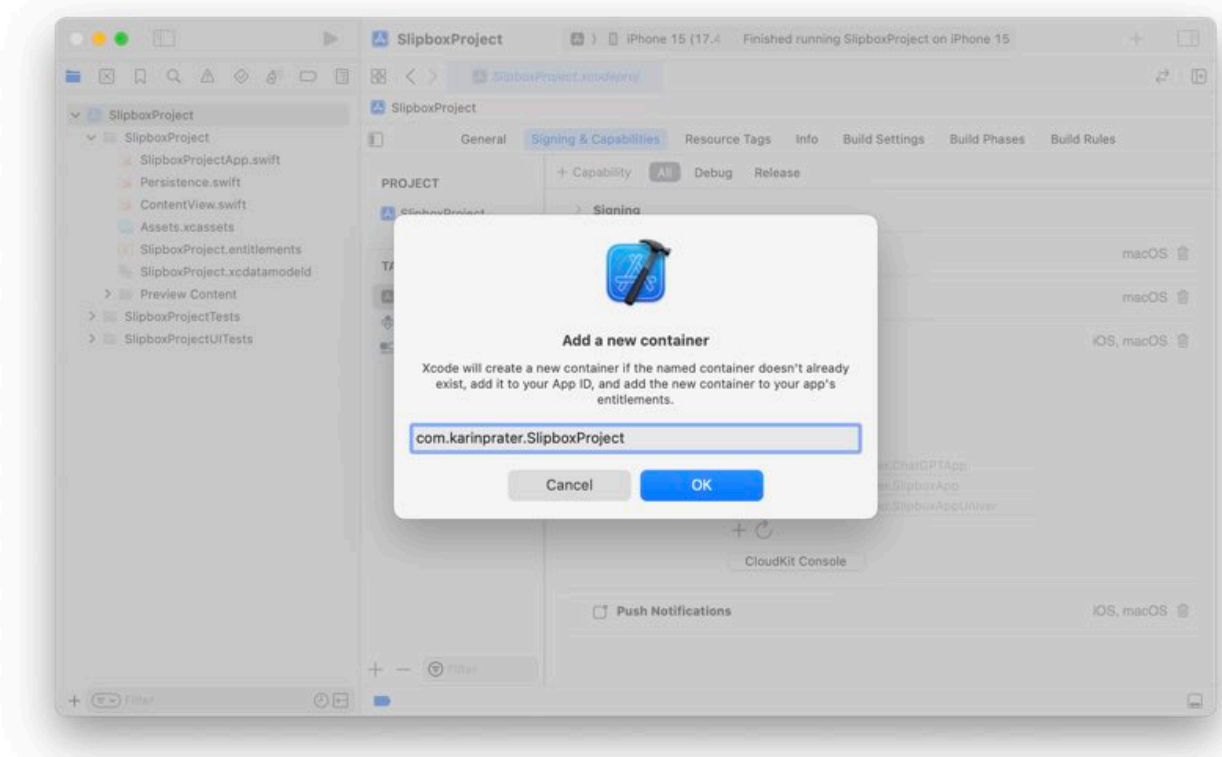


It’s important to note that most of the iCloud capabilities only work with a paid developer account. If you don’t have a paid account, you won’t see the iCloud option. Therefore, you may need to switch to a paid developer account to proceed.

Scroll down to the new section **“iCloud” (1)** and toggle the **CloudKit service (2)**:

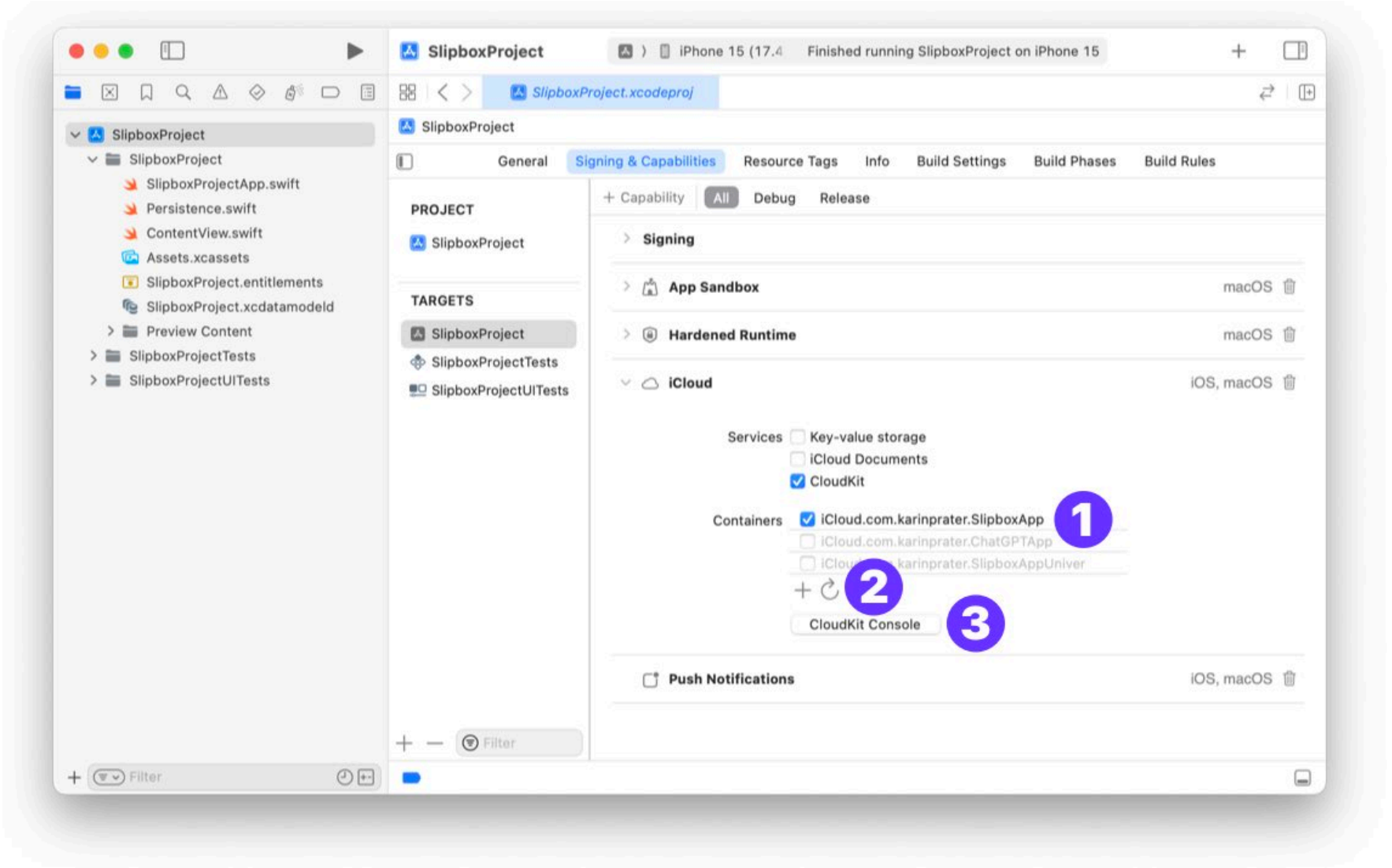


You need to add a new container. The container represents the server you want to use for iCloud sync. You can choose an existing container or create a new one. Press the **“+” button (3)** to create a new container.



It’s recommended to use the bundle identifier as the container name for simplicity and to maintain a clear association between the app ID and server ID.

You should now see you new container added (1). If it is red, try pressing the refresh button “2”, until Xcode resolves the issue:

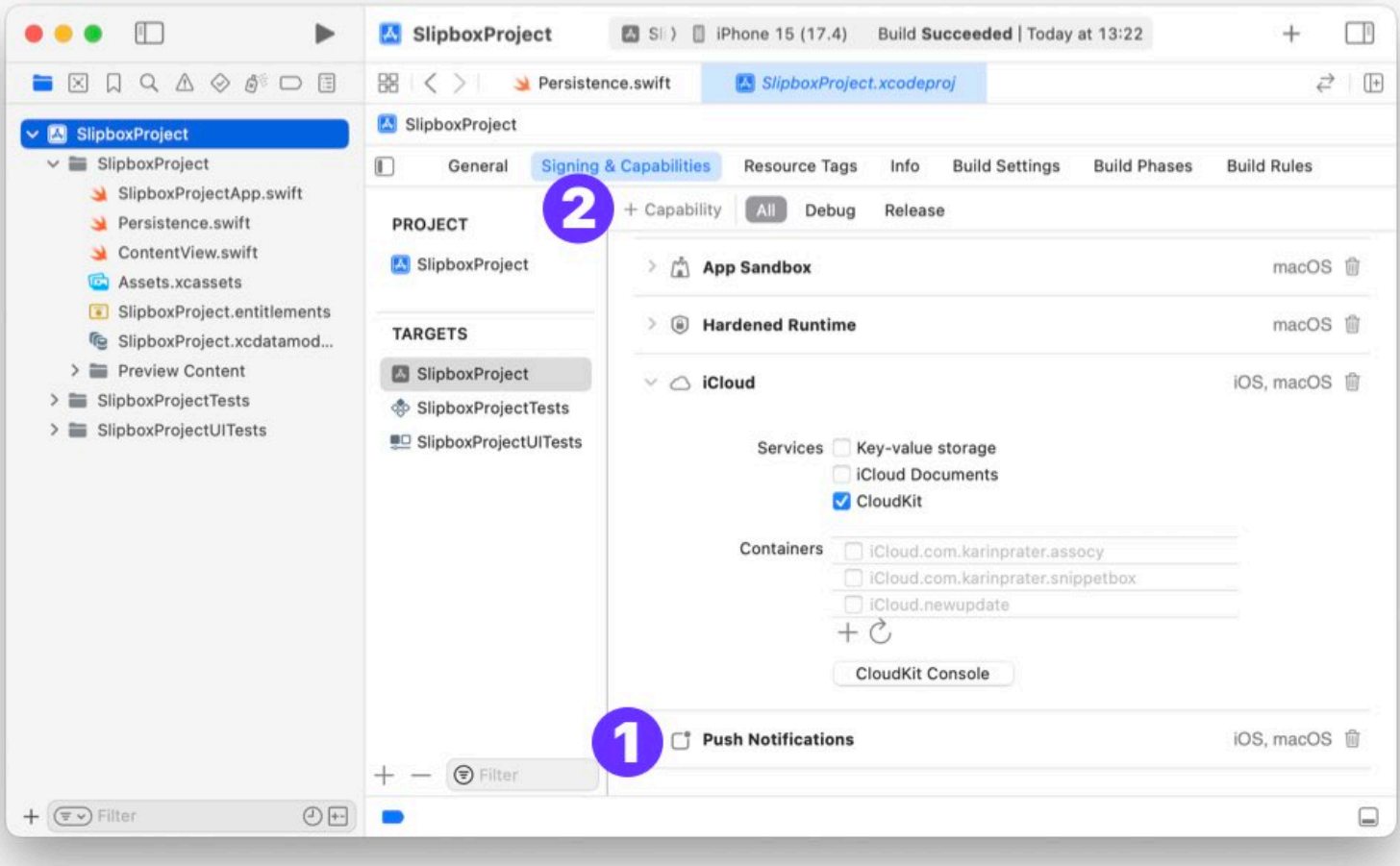


Once the container is added, you can verify its creation by clicking on the **“CloudKit Console” button (3)**. This will take you to the CloudKit dashboard, where you can manage your database and view different containers. Initially, you may not see any data in the dashboard, but don’t worry, we will propagate our schema to CloudKit in the next steps. First, we need to update the Persistence.swift file to use iCloud sync.

Additional Capabilities

To sync data, you’ll need to set up your Core Data stack to work with CloudKit, which includes enabling push notifications and background modes for remote notifications and background processing.

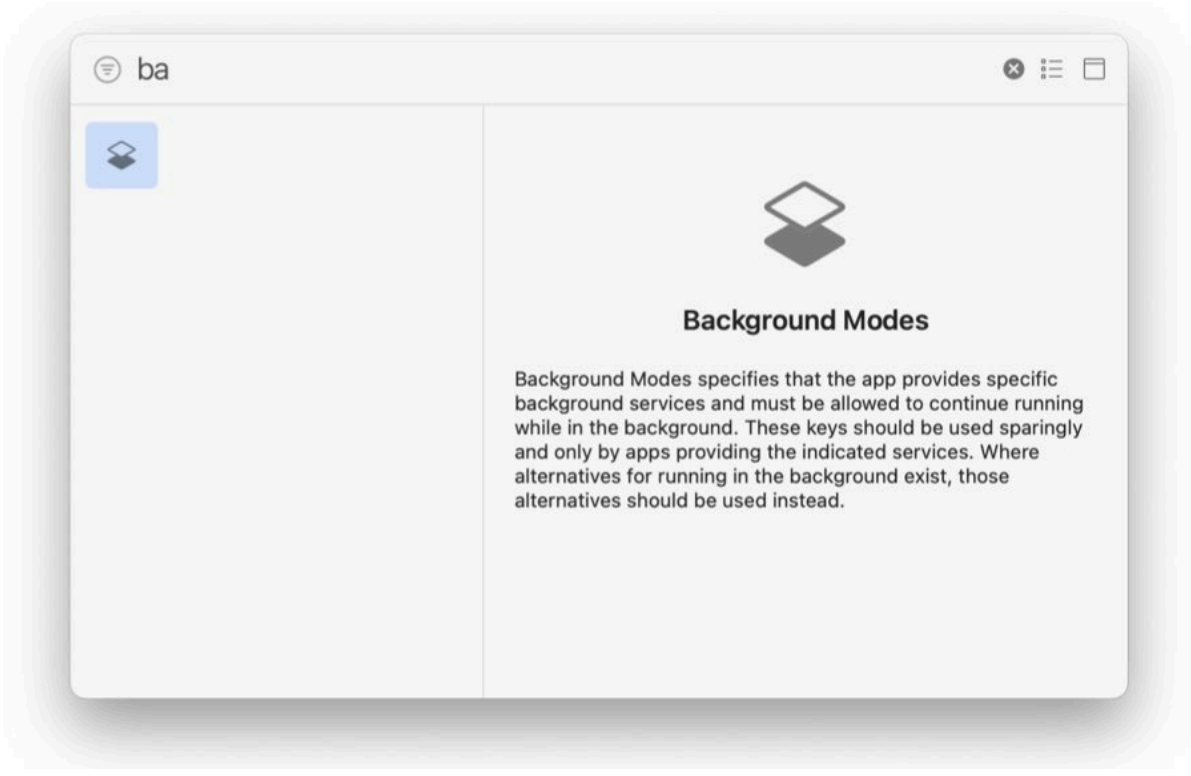
When you add iCloud to your project it also automatically adds **push notifications (1)**:



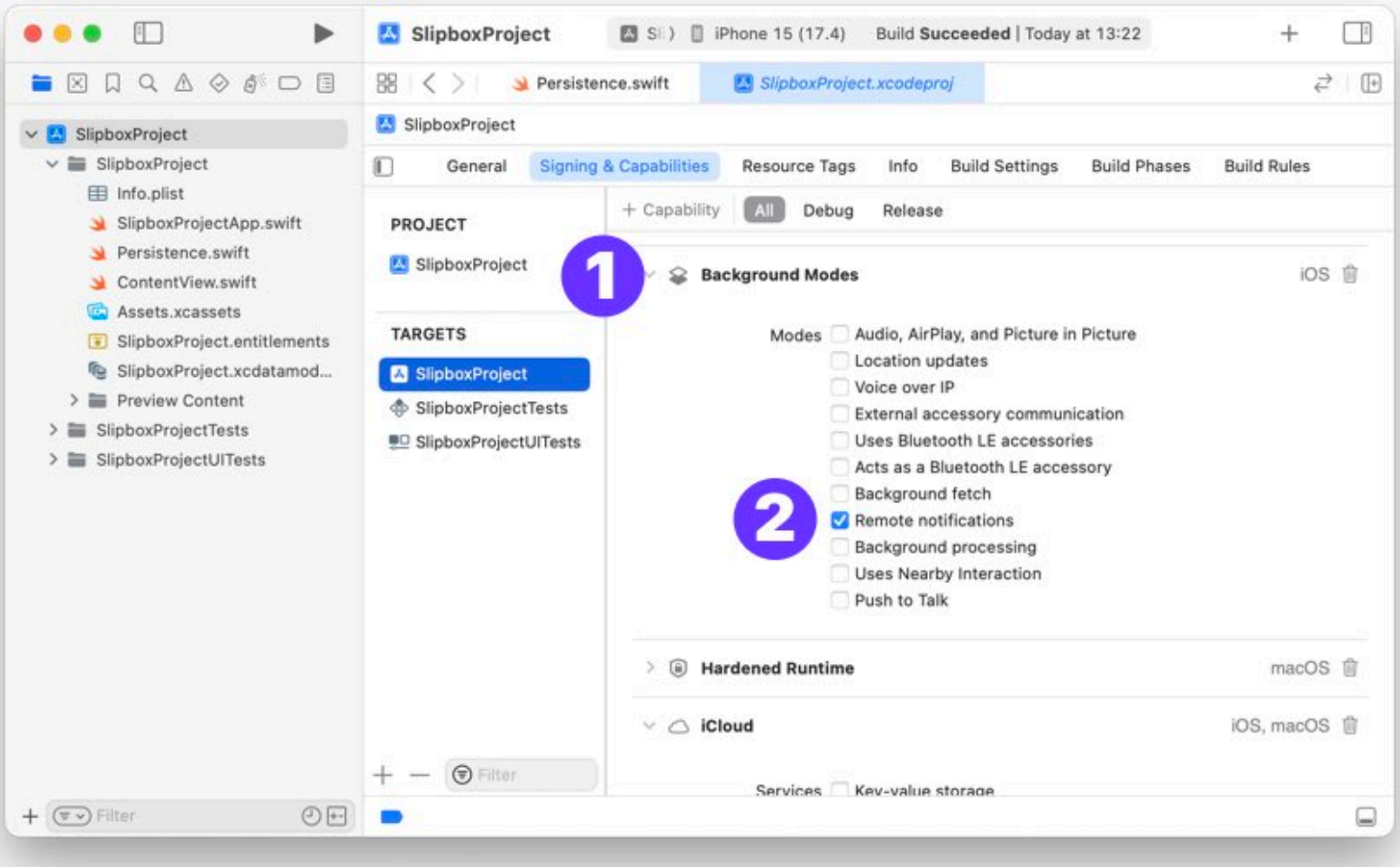
In order to sync your data, the container needs to perform tasks in the background. It receives silent notifications in the background when new data is available.

Therefore, you need to **add another capability (2)**.

Select **“Background Modes”** in the capabilities editor:



A new section for **“Background Modes” (1)** is added. Enable **“Remote notifications” (2):**



If you get the error **“CloudKit push notifications require the 'remote-notification' background mode in your info plist.”** In the debug area. You can change the info plist (Open As -> Source Code) and paste the dictionary entry for remote notification:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
    <key>UIBackgroundModes</key>
        <array>
            <string>remote-notification</string>
        </array>
    </dict>
</plist>
```

Persistent Container Setup

When initializing your persistent container, you'll want to add options to upload your schema to CloudKit. This is crucial for syncing your data model to iCloud.

Open **"Persistence.swift"** file and change the container to **NSPersistentCloudKitContainer**:

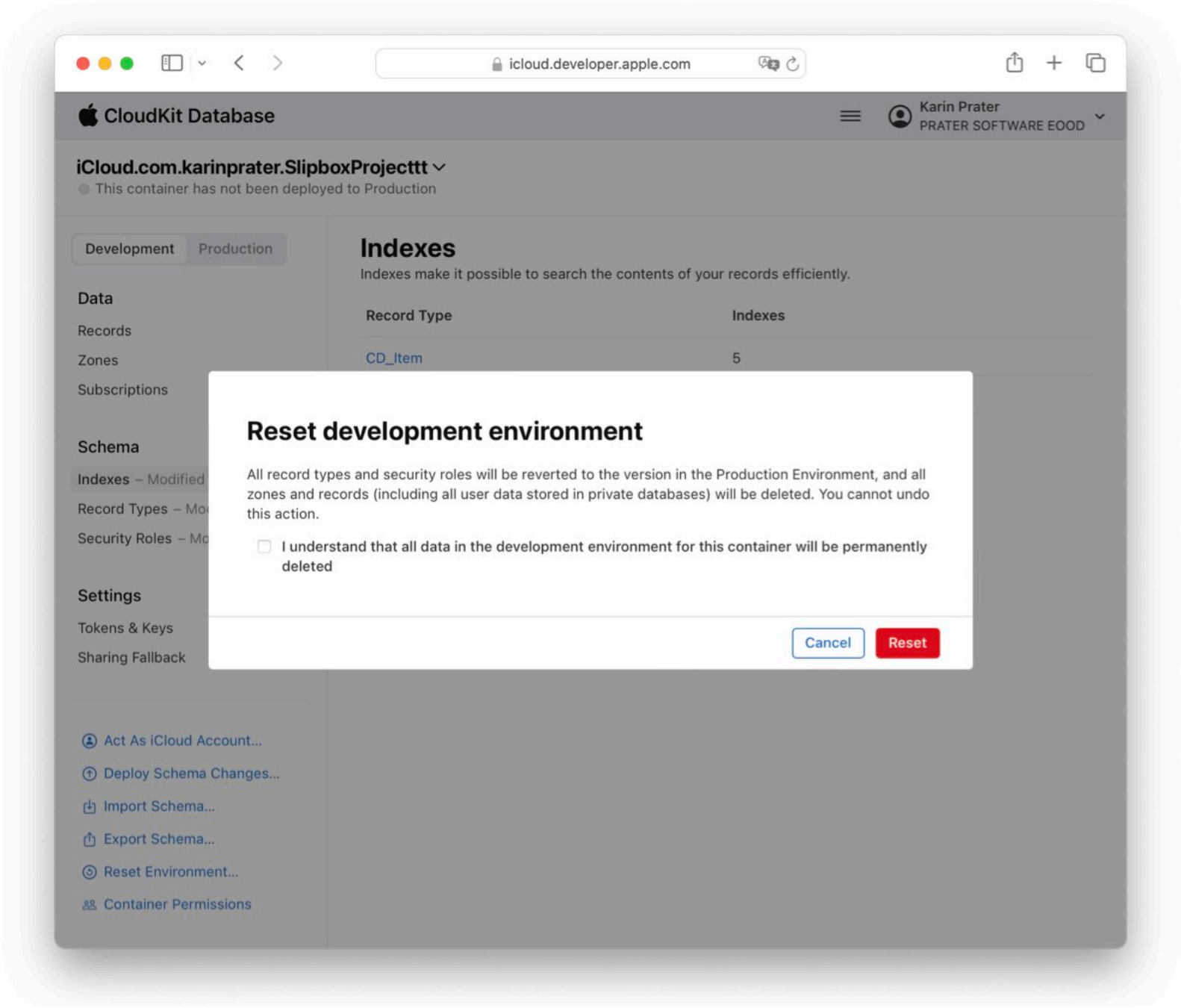
```
struct PersistenceController {

    static let shared = PersistenceController()

    let container: NSPersistentCloudKitContainer

    …

}
```

This container is set up in the init. Change the initialiser to use the new **NSPersistentCloudKitContainer** type:

```
init(inMemory: Bool = false) {
    container = NSPersistentCloudKitContainer(name: "SlipboxProject")

    if inMemory {
        container.persistentStoreDescriptions.first!.url =
                                                URL(fileURLWithPath: "/dev/null")
    }

    container.loadPersistentStores(completionHandler: { (storeDescription, error) in
        if let error = error as NSError? {
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    container.viewContext.automaticallyMergesChangesFromParent = true
}
```

Testing iCloud sync

As a test I am going to build and run the app on my real device iPhone and macBook. I will add new entries on my phone which should be seen on the Mac app too. Syncing can take some time to be



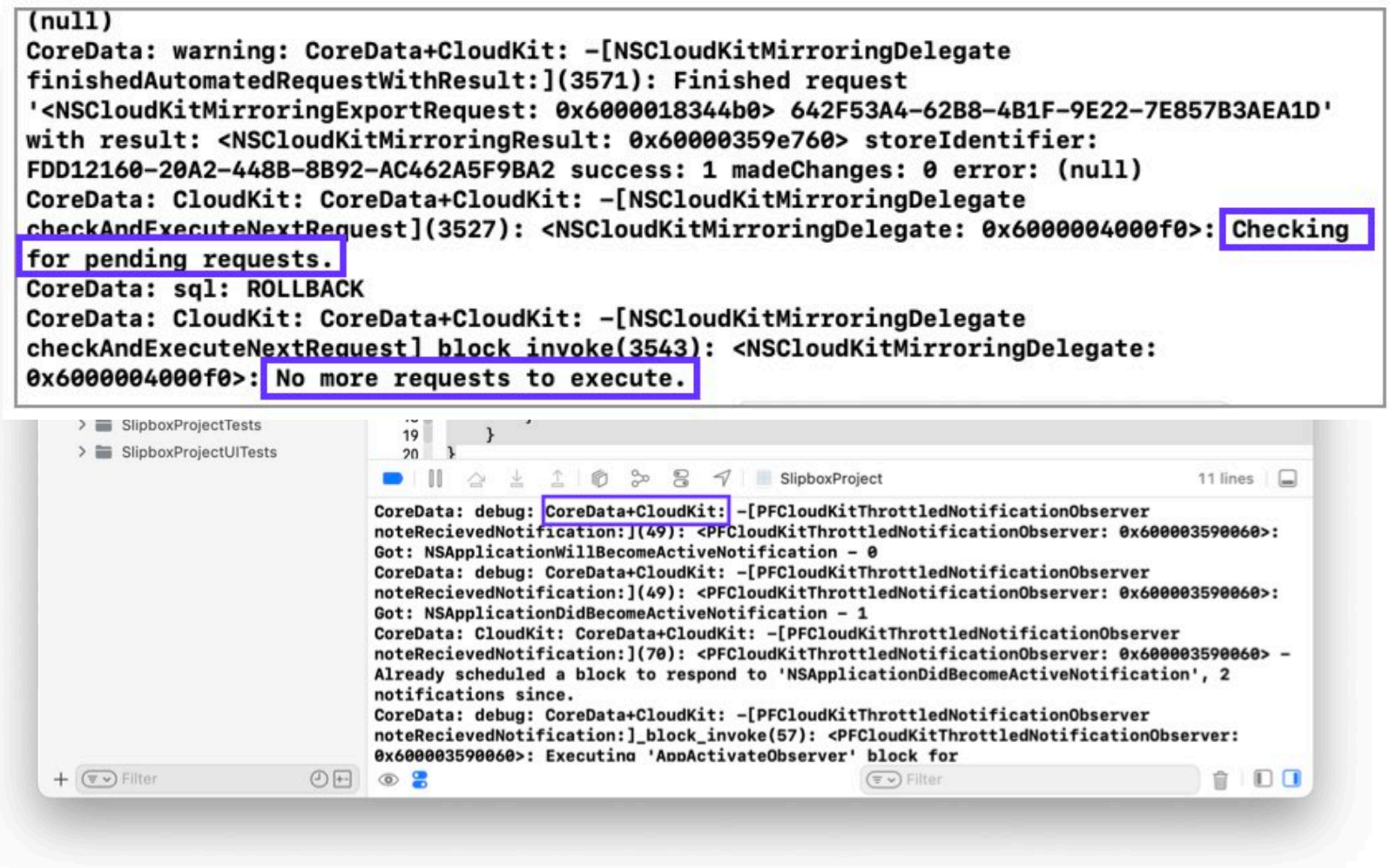
reflected. When the app launches it may take up to 30sec from my experience. Additionally, changes should be faster with ca. 2-5sec.

If you check the Xcode debug area, you will see much more information, including iCloud sync 🎉

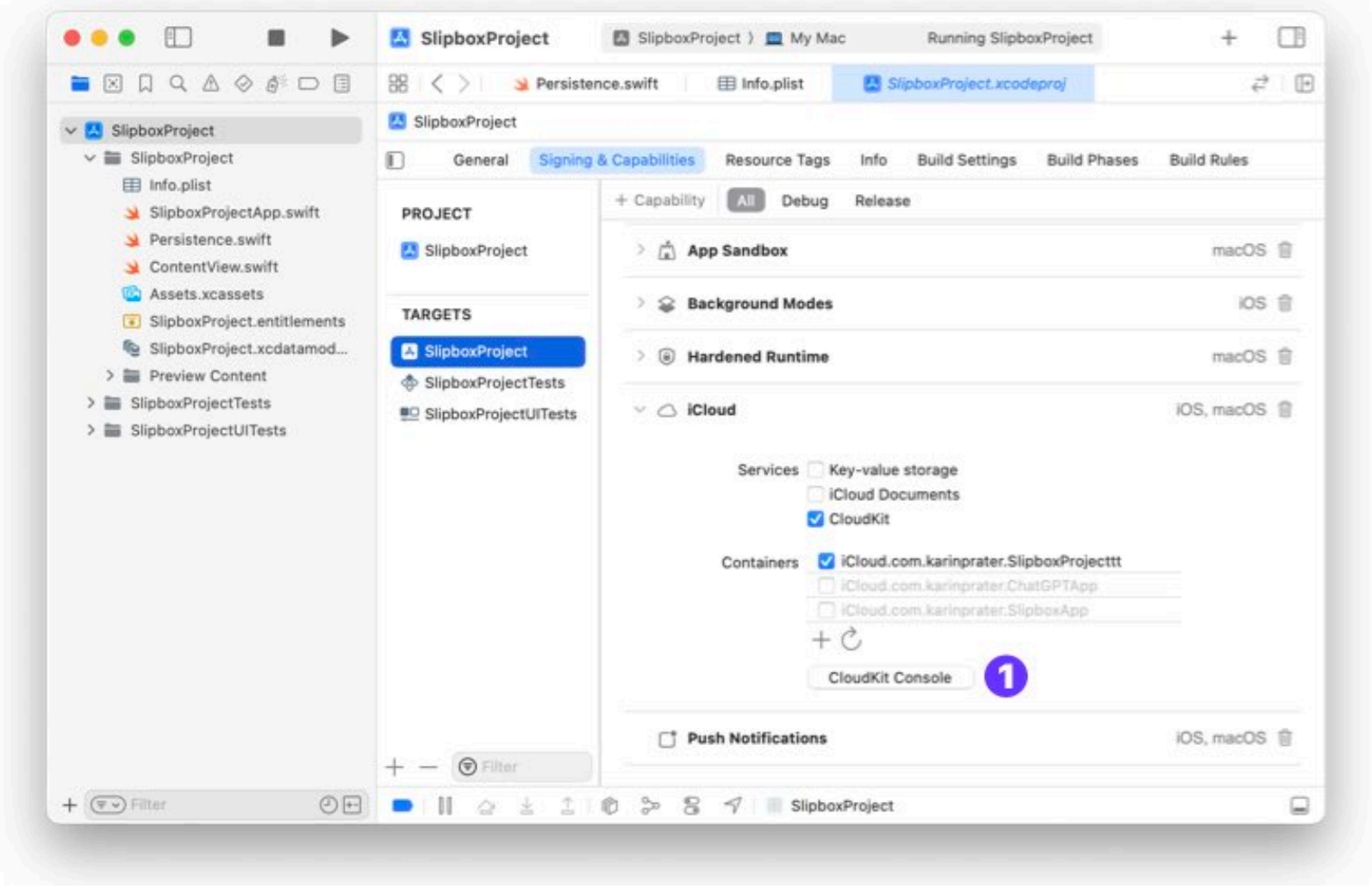
Scroll down to the end of the list. You should find something like “Checking for pending requests”. This indicates that the core data container communicates with iCloud. If no additional data is available a “No more requests to execute” message is shown:

It is important to note that your changes are only passed to iCloud if you call save on the context. Otherwise, changes are only local and temporary in memory.

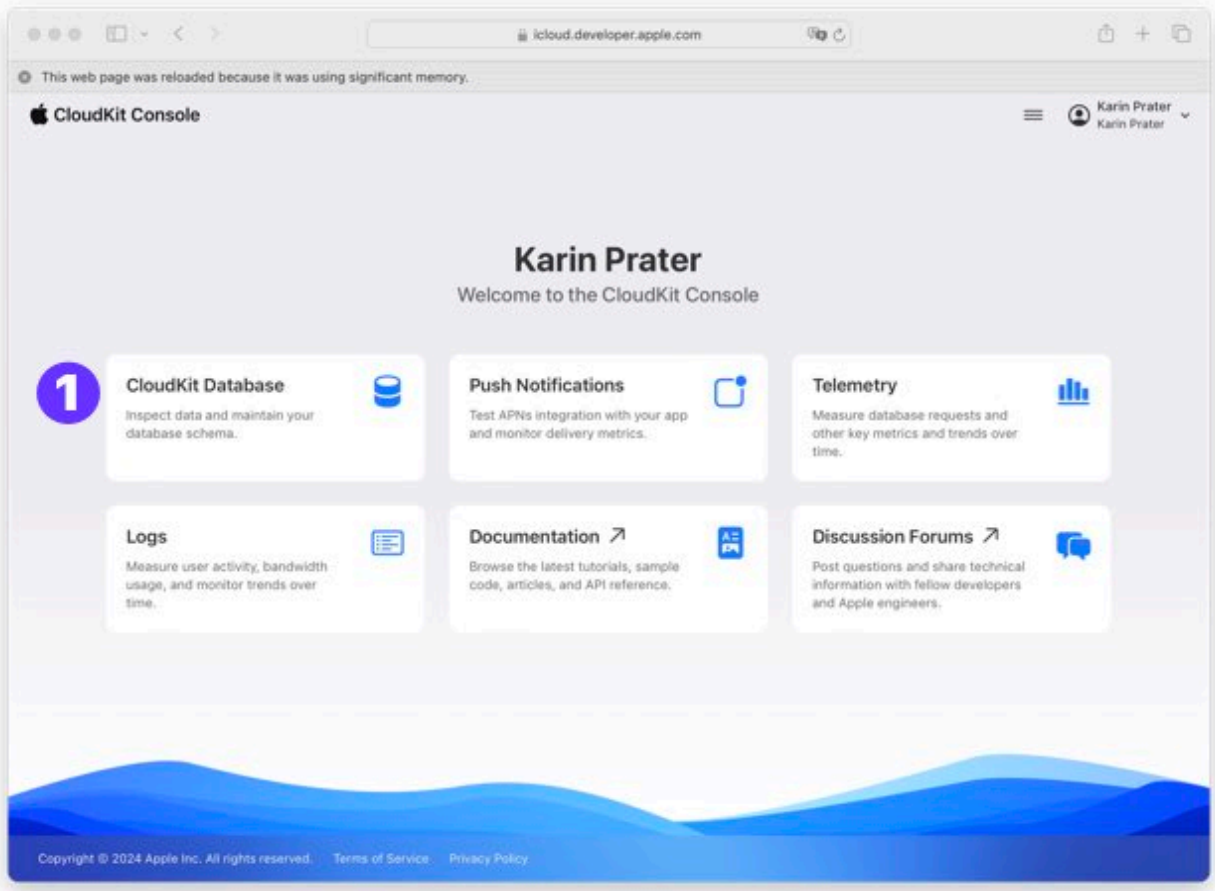
CloudKit Dashboard



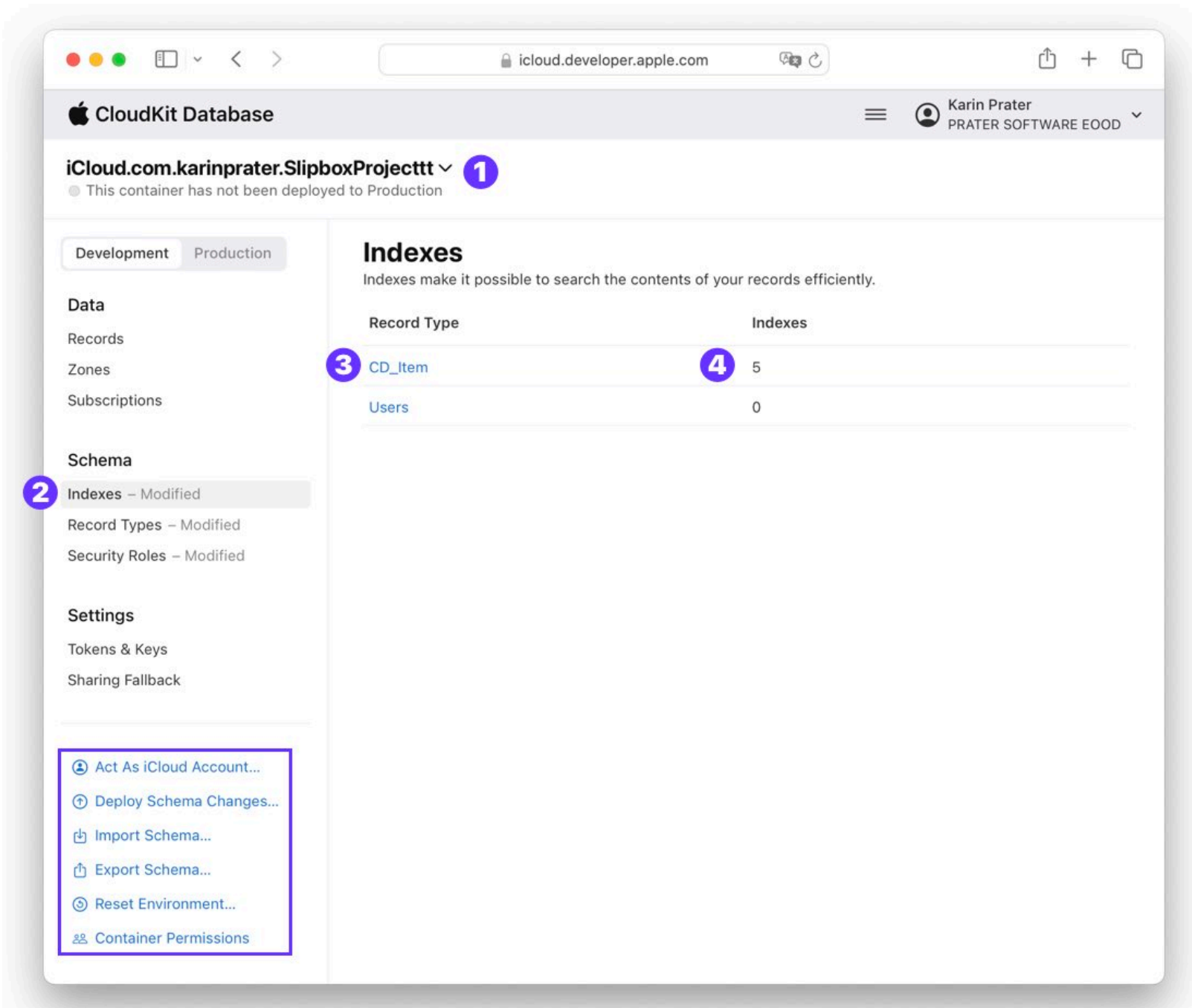
Now, that we have successfully run the app and send some data to CloudKit, the backend should be ready. Head back to your project capabilities and select **“CloudKit Console” (1)**



Your default browser e.g. Safari will open CloudKit Console. Open **“CloudKit Database” (1)**:



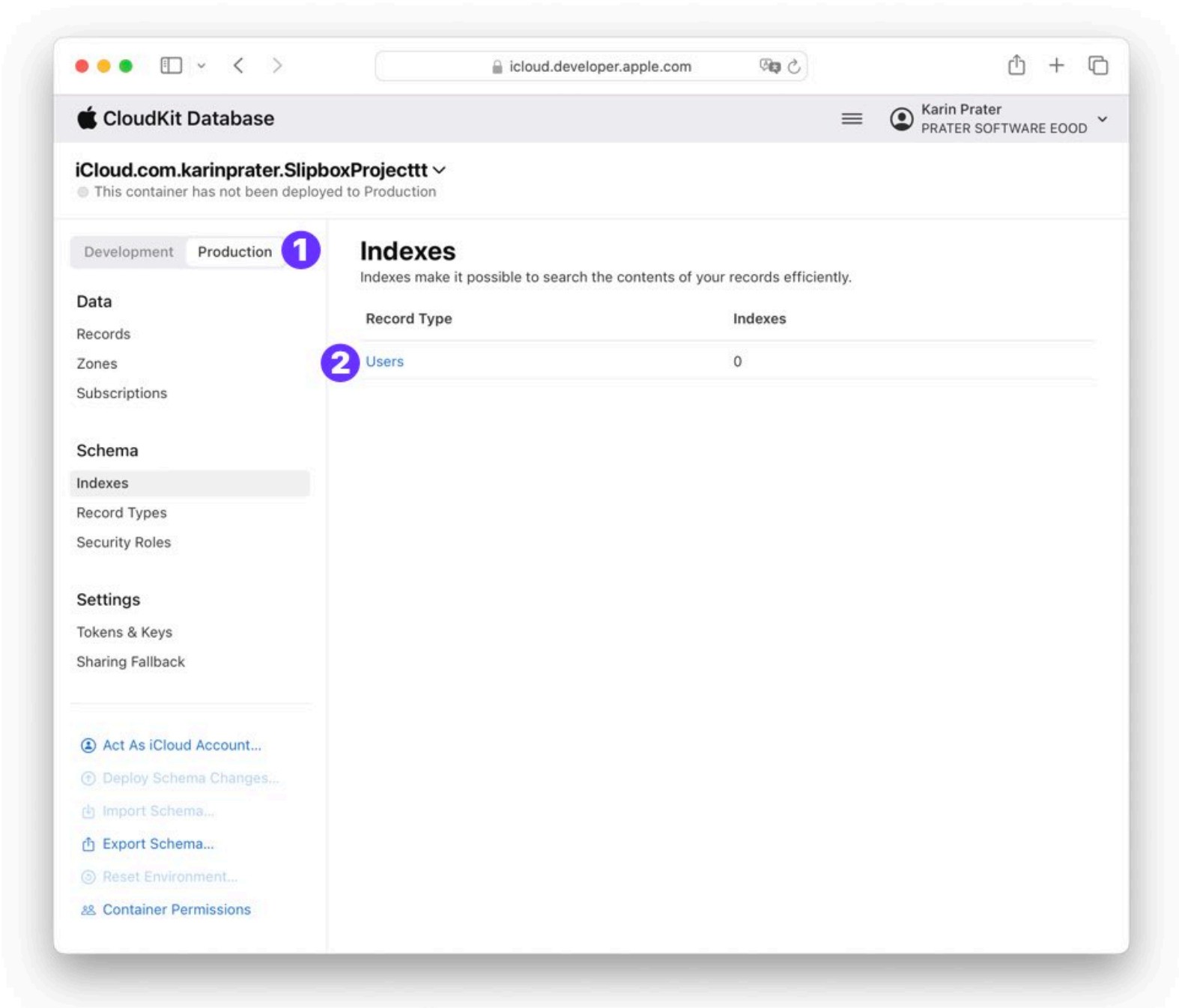
You should now see the CloudKit Database. First, select the **container identifier (1)** that you created in Xcode:



To see the schema, select Schema > Indexes (2). On the right, you will see a list of your types. The Xcode starter project uses an “Item” type. Because I run the app and create data, this data was sent to CloudKit and used to create a Schema type. The CloudKit types are mapped from the CoreData types with a “CD_” prefix. That is why the type we use in SwiftUI is “Item” and the one in the CloudKit dashboard is named **“CD_Item” (3)**. I created 5 entries when I tested this on the device (4).

In the right bottom corner are 6 buttons that help you manage your schema. Most of the time you want to test something, make changes, and revert changes. How does CloudKit handle Development and Production environments? In the above screenshot, I selected the development environment.

Switching to **Production (1)**:

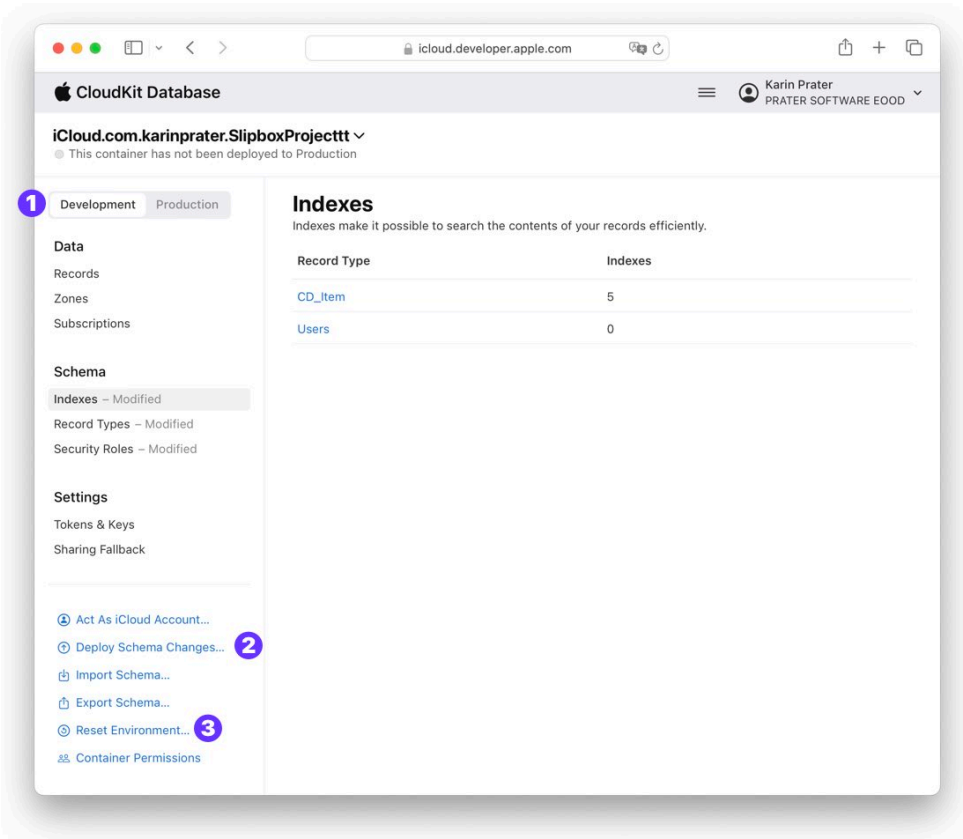


In the production environment, we only see the default “User” Record Type. The production environment is used for the final app that users see when they download the app from the app store. **Before you**

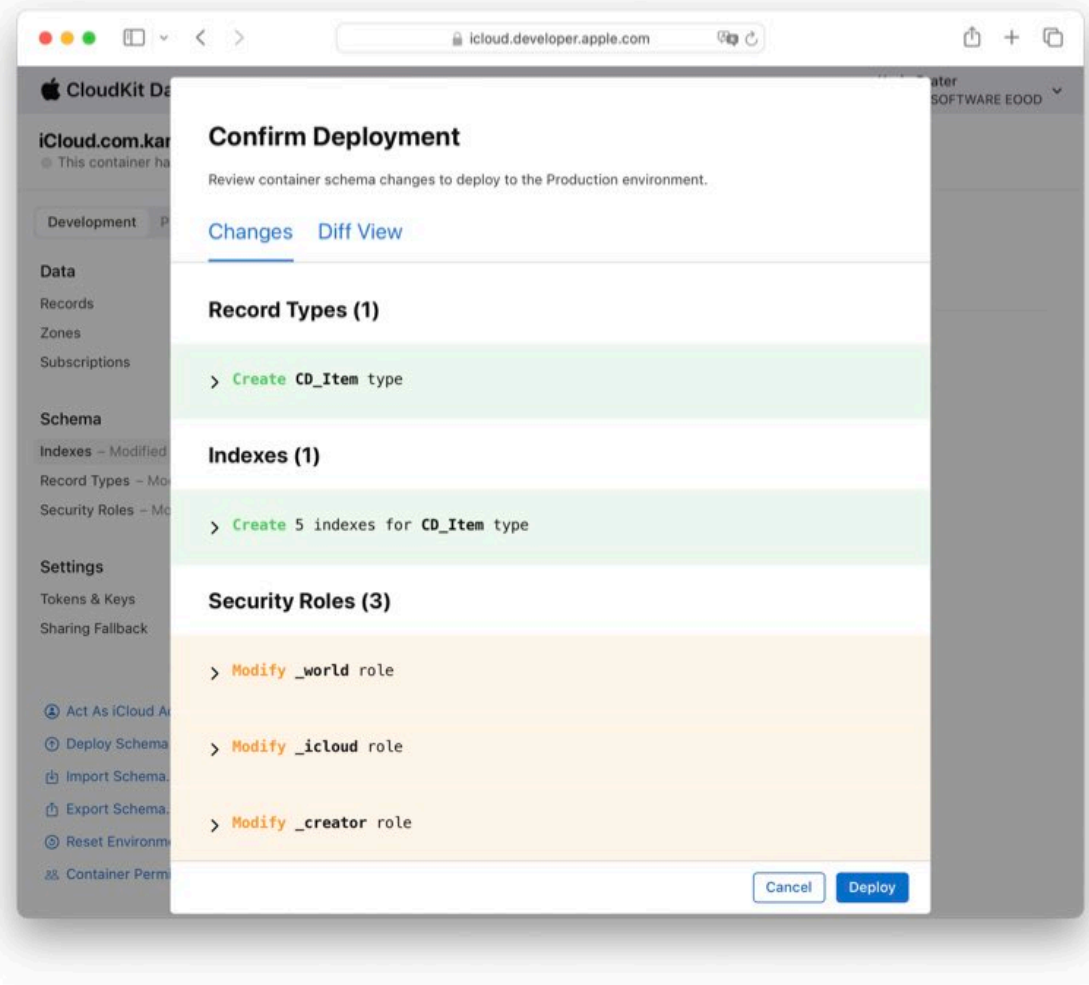
submit an app to the app store you must deploy your new schema otherwise the app will crash for end users.

Deploying Changes to Production

Go back to **Development (1)** and select **“Deploy Schema Changes” (2)** from the actions area. An editor will open, where you can choose to “Deploy”.



Once you deploy (apply Schema to Production), you cannot revert this. Migrating Schema with CloudKit is very limited. E.g. you cannot remove or rename attributes once they have been set. Be careful when doing this.



Additionally before publishing your app in the app store, test it with TestFlight, which uses the Production environment.

Reverting Changes

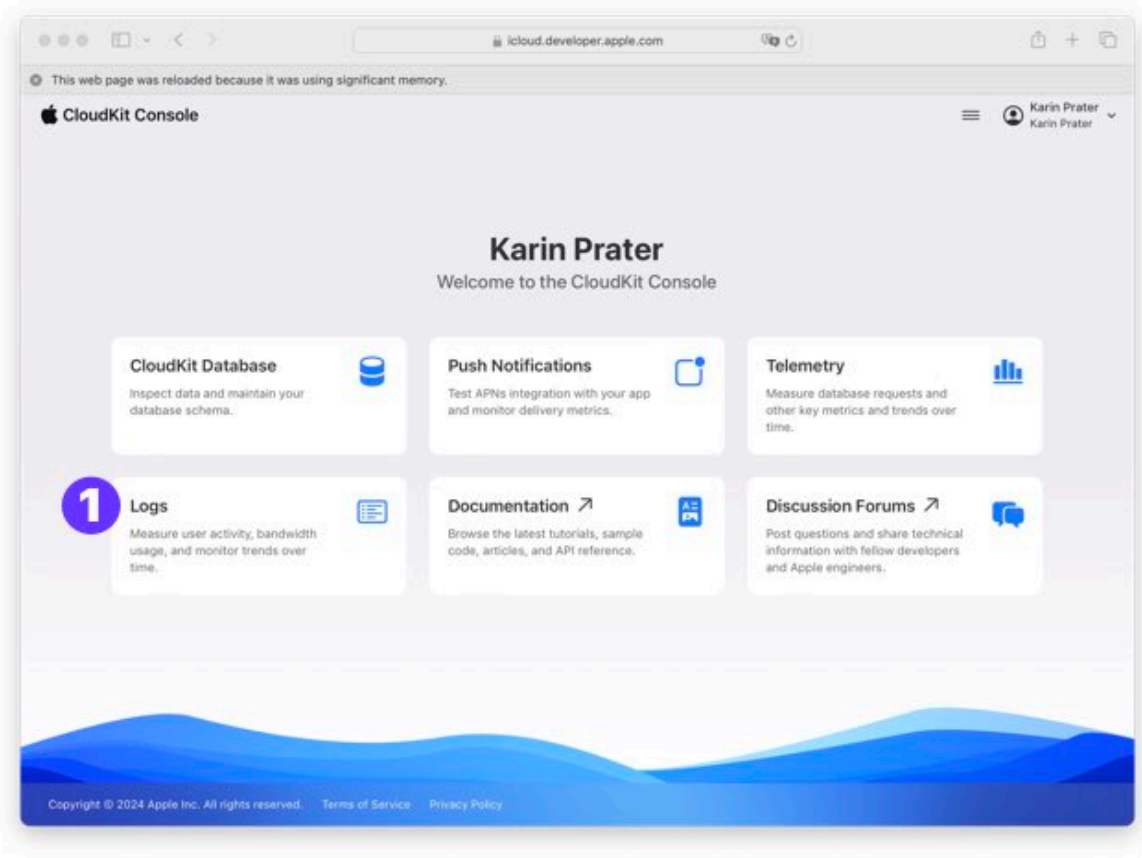
If you want to undo the changes to your schema and set it back to the production settings, choose **“Reset Environment”** from the bottom right corner:

You can reset the current schema because we will not use the “Item” type in the future. Once done, you should see only the “User” Record type.

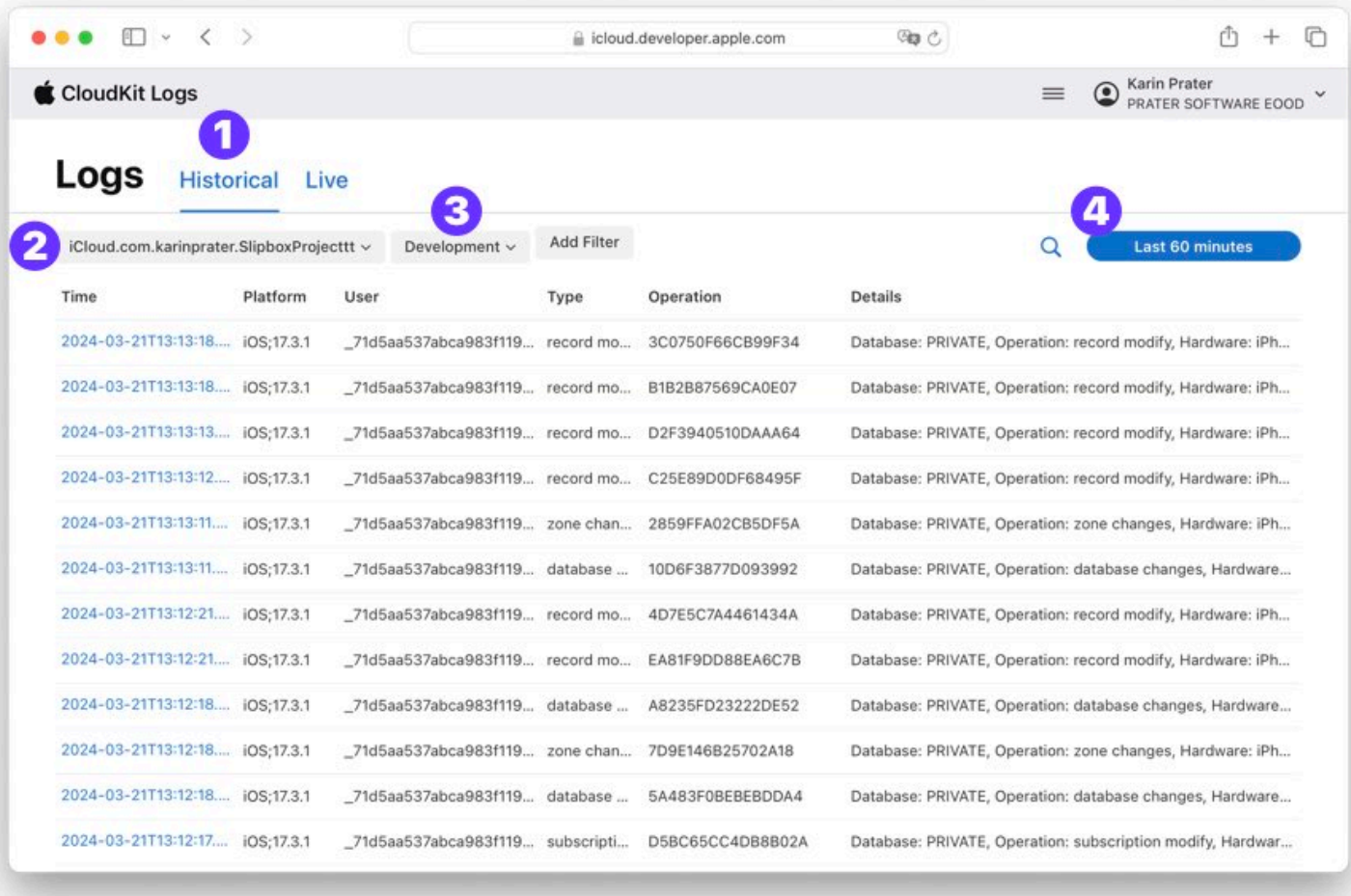
Dealing with CloudKit can be intimidating and overwhelming but it is doable and requires a little bit of testing and patience.

Checking CloudKit Logs

You can also see the data in CloudKt. Go to the first screen and select “Logs”



You can look at **historical (1)** or live data. Make sure to select the container identifier from the Xcode project (2). Select **Development Environment (3)** and press “**last 60 minutes”**. After a longer loading, you should see your data being displayed:



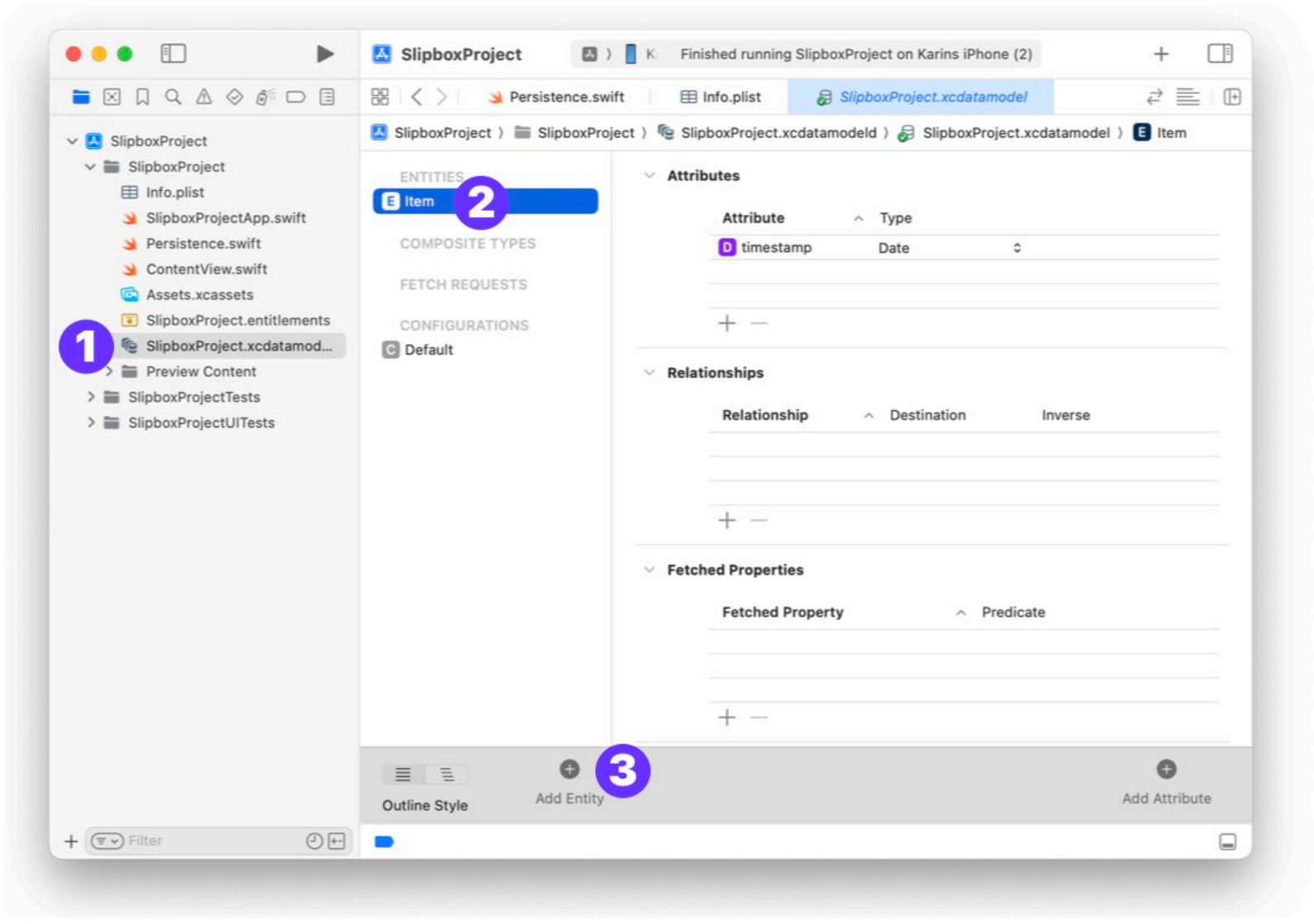
This is useful, to see if your data is passed through. You can also have a look at the “Telemetry” sections, where you see performance metrics. There is a cap on how much data can be shared with CloudKit and you should check it regularly to make sure your app is below the limits and functions as expected.

1.3 NOTE MODEL AND CRUD

In this lesson, we will start working on our model and make changes to our schema. We will create an entity called “Note” with attributes and use this new data in our SwiftUI views. To achieve this, we will need to implement CRUD operations, which include creating new Core Data objects, reading objects from the database, updating objects, and deleting objects.

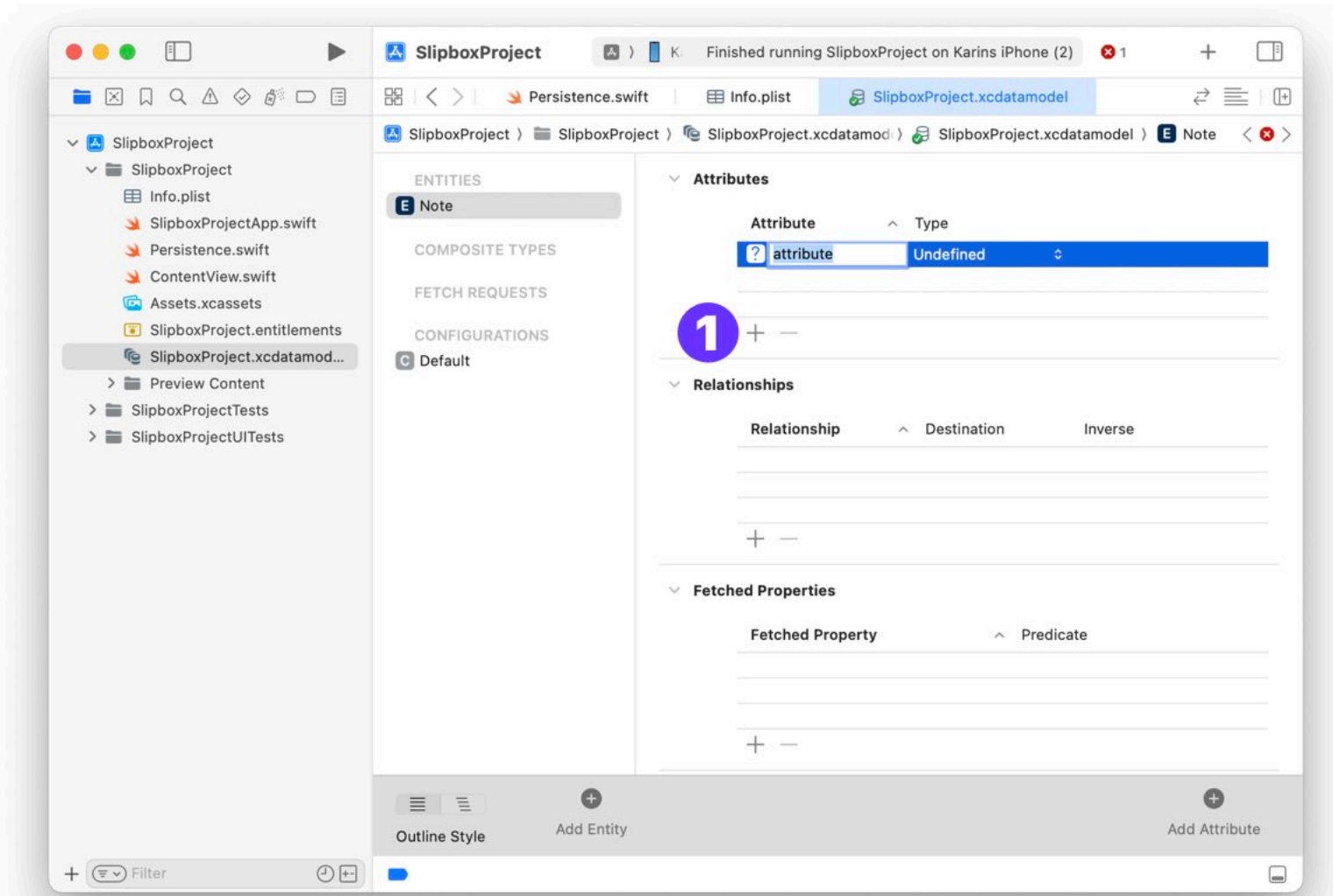
Creating the Note Entity

Let’s begin by examining the existing data model in the Slip Box project’s **.xcdatamodeld file (1)**. We have an entity called **“Item” (2)** which we don’t want to use. To delete it, we can simply select it and press the delete key.

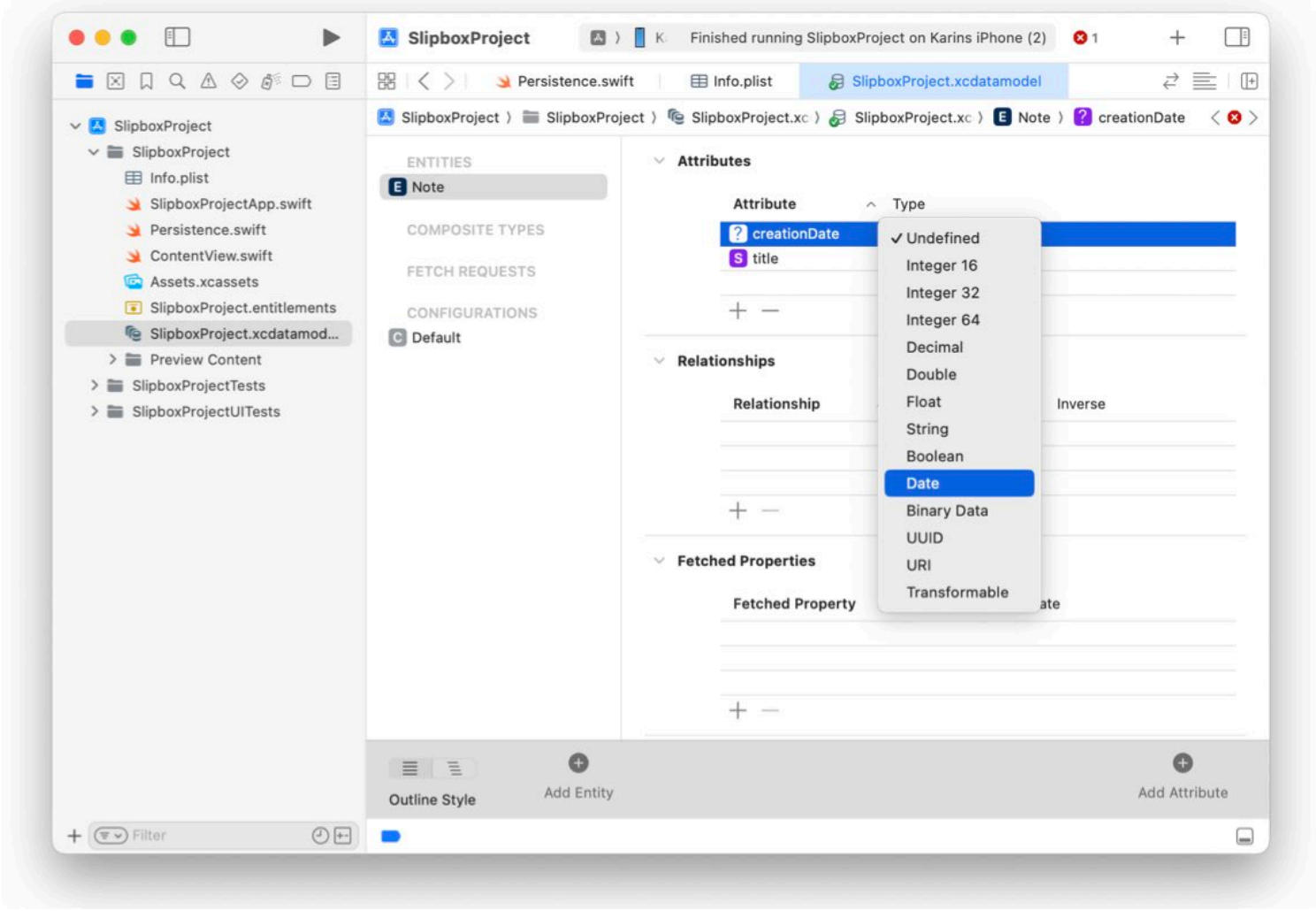


Next, we will create a new entity by clicking on the **“Add Entity” (3)** button at the bottom and renaming it to “Note”.

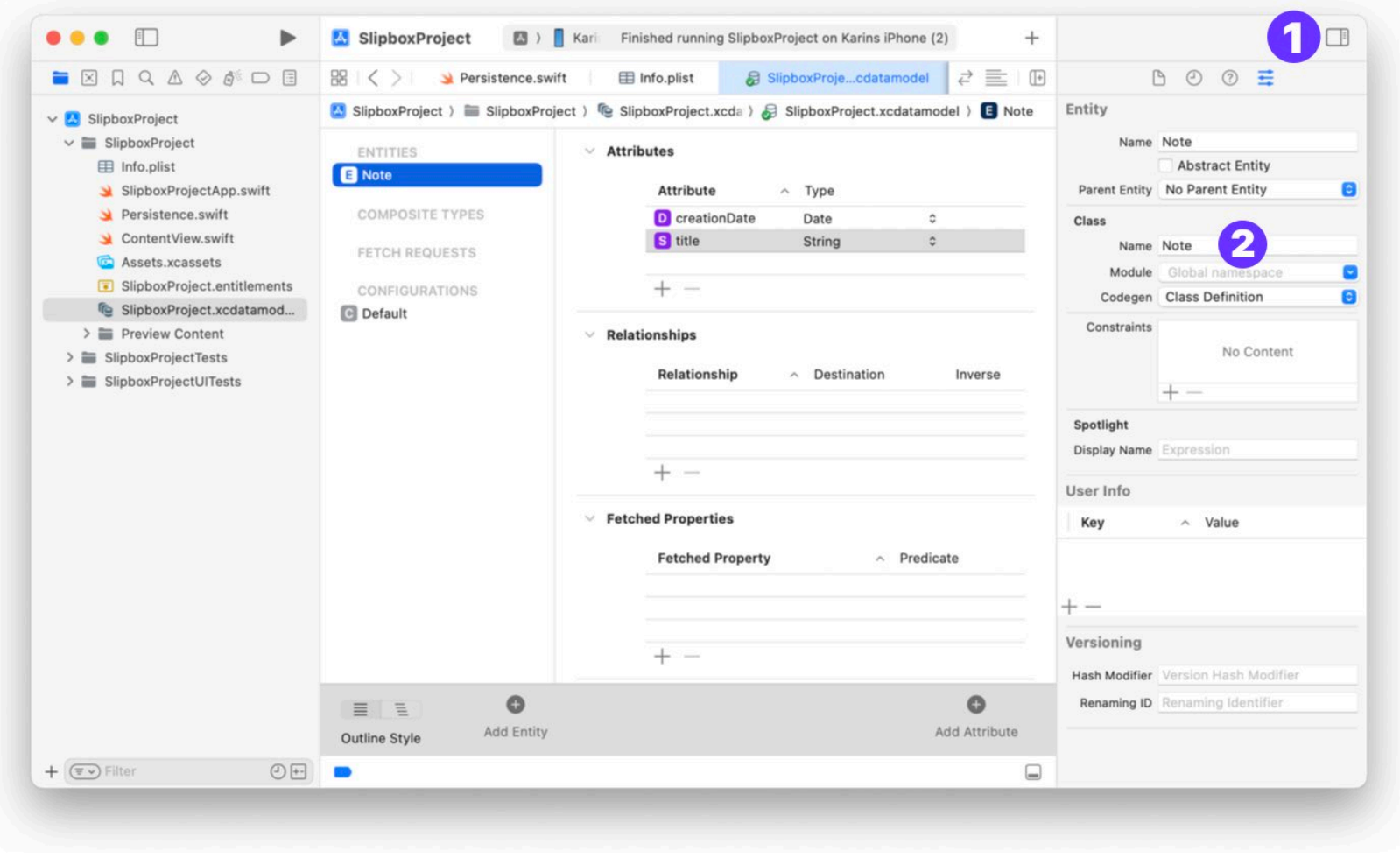
We can also add attributes to our entity, such as the title of the note. Press **“+” button (1)** to add to attributes. It’s important to define a type for each attribute to avoid errors. For example, if we leave the type undefined, we will receive an error message stating that the note title cannot have an attribute type of undefined.



Select String for the title and Date for the creationDate attribute:

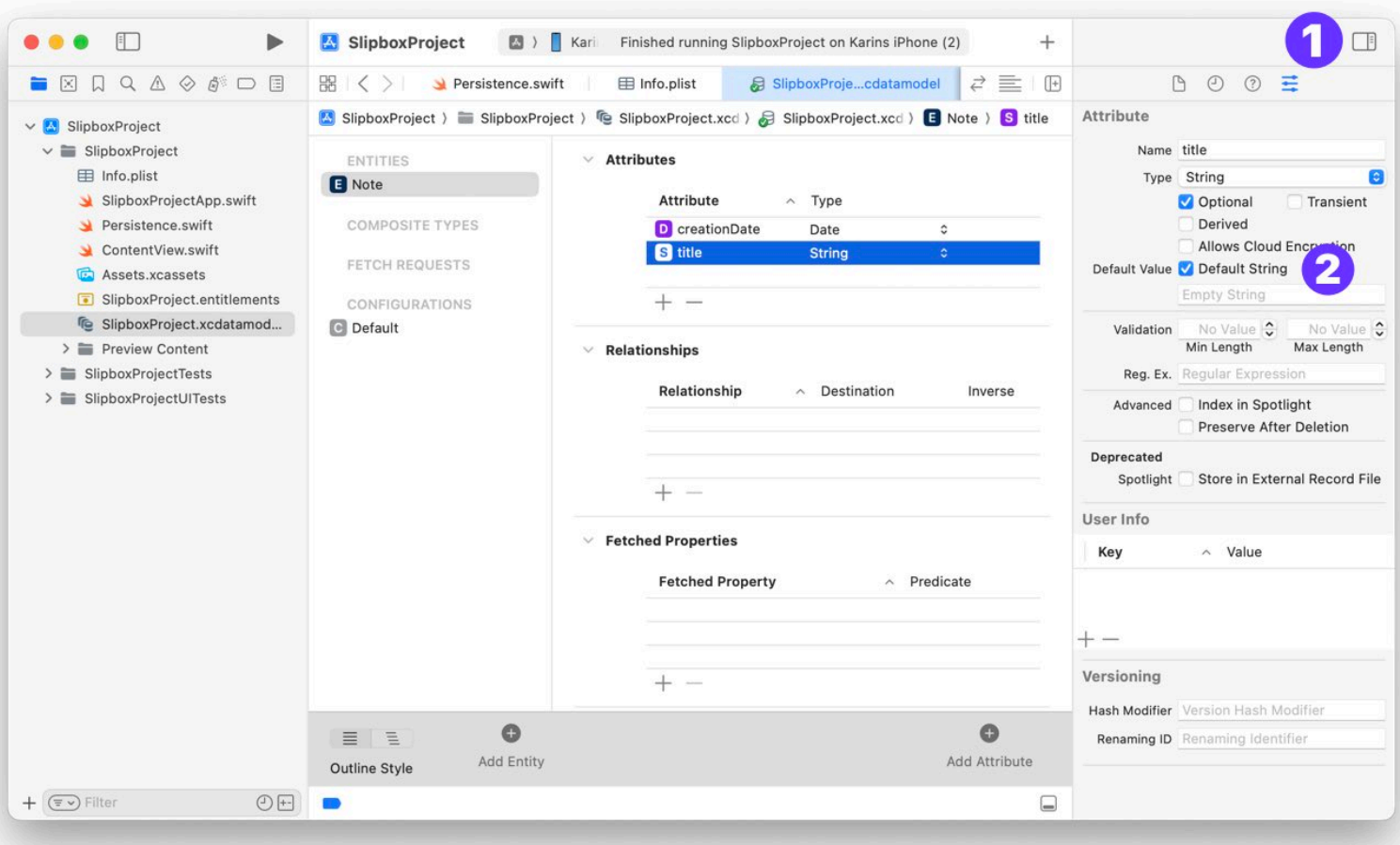


When we select the entity and **open the sidebar (1)**, we can see more information about it, including the name and parent entity. The class entities in Core Data are always represented as classes, and Xcode generates the necessary code for us based on the class definition in the code generation section. It's best to let Xcode handle the code generation to avoid complications.



We can examine the files that Xcode generates for us, such as the Note+CoreDataClass.swift file. However, it's important to note that these files are automatically generated and should not be edited directly. We can add additional functionality to the Note entity by creating an extension in a separate file that we manage ourselves.

You can also open the attribute inspector in the sidebar:



For example, I can set a default value for the title attribute and choose an empty string. This way, the title attribute is not optional and easier to handle in SwiftUI.

Additionally, we can specify whether the attribute is optional or provide a default value. For now, we won't be using any advanced features like storing external record files.

Error: You might see an error in Xcode. This is because you removed the Item type which is used throughout the project. We are going to fix it soon.

Adding Syntactic Sugar to Models

I am going to add an extension to Note where we can add convenience code. This type of "make my life easier" code is also referred to as syntactic sugar. We are adding sugar to our codebase to make life more sweet 🍭.

Create a new group called "Model" and add a new Swift file named "Note+Helper.swift".

In this file, we'll import Core Data since we'll be working with it extensively. Here, we can add convenience methods and properties to the Note entity by creating an extension.

```
import Foundation
import CoreData

extension Note {
    // write convenience code here
}
```

I want to create an initializer that takes a title string as a parameter and creates a new Note object with the provided title and the current date as the creation date:

```
import Foundation
import CoreData

extension Note {

    convenience init(title: String, context: NSManagedObjectContext) {
        self.init(context: context)
        self.title = title
    }

}
```

To create this convenience initializer, we'll use the class initializer init(context: NSManagedObjectContext) provided by Core Data. We'll pass the context parameter to the initializer to indicate that the new Note object belongs to the specified database.

To ensure that the creation date is always set, we can override the awakeFromInsert method in the Note entity. This method is called every time a new object is inserted into the database. By setting the creation

date in this method, we can guarantee that it will always have a value, even if the convenience initializer is not used.

```
import Foundation
import CoreData

extension Note {

    …

    public override func awakeFromInsert() {
        self.creationDate = Date()
    }

}
```

Updating Persistence.swift File

First, I want to remove the errors in Persistence.swift file. It is still using the Item type, which we just removed. Change the preview property to use Note instead:

```
import CoreData

struct PersistenceController {

    static let shared = PersistenceController()

    let container: NSPersistentCloudKitContainer


     …
    //MARK: - SwiftUI preview helper

    static var preview: PersistenceController = {
        let result = PersistenceController(inMemory: true)
        let viewContext = result.container.viewContext
        for index in 0..<10 {
            let newNote = Note(title: "note \(index)", context: viewContext)
            newNote.creationDate = Date() + TimeInterval(index)
        }

        return result
    }()
}
```

This will create 10 note entries that will be shown in the Xcode preview canvas.

Updating SwiftUI Views to use Note

Open “Content.swift” file. We need to remove all uses of Item type. By doing so, you will learn about CRUD. CRUD operations are fundamental when dealing with databases. Let’s break down each operation and see how it’s implemented in Core Data.

Reading Data

To read objects from the database, you can use the @FetchRequest property wrapper in SwiftUI views:

```
import SwiftUI
import CoreData

struct ContentView: View {

    @Environment(\.managedObjectContext) private var viewContext

    @FetchRequest(
        sortDescriptors: [NSSortDescriptor(keyPath: \Note.creationDate,
                                           ascending: true)],
        animation: .default)
    private var notes: FetchedResults<Note>

    var body: some View {
        …
    }
}
```

You need to specify “sortDescriptors” which defines how the objects are sorted. In the above example, they are sorted by the creation date in ascending order. The newest object will appear at the end of the list.

Next update the body of ContentView to show the notes:

```
struct ContentView: View {

     …

    var body: some View {
        NavigationView {
            List {
                ForEach(notes) { note in
                    NavigationLink {
                        Text("Item at \(note.creationDate!,
                             formatter: itemFormatter)")
                    } label: {
                        Text(note.creationDate!, formatter: itemFormatter)
                    }
                }
                .onDelete(perform: deleteItems)
            }
            …
        }
    }
}
```

Notice that I am force unwrapping the optional creationDate property. I will show you a much safer way to deal with optionals in section 3.

@FetchRequest can also be used to **filter what data is returned** e.g. you might only get notes that were created today. We define the filter with NSPredicates. To shorten the code, I am writing in my SwiftUI views, I will create a function in Notes extension that sets all these default sort descriptors and predicates:

```
extension Note {

   …

   static func fetch(_ predicate: NSPredicate = .all) -> NSFetchRequest<Note> {
        let request = NSFetchRequest<Note>(entityName: "Note")
        request.sortDescriptors = [NSSortDescriptor(keyPath: \Note.creationDate,
                                                    ascending: true)]
        request.predicate = predicate

        return request
    }
}
```

NSPredicate are a bit difficult to get used to and require exact usage. As a convenienc, I am adding a new file to the project named “NSPredicate+helper” where I define an extension to NSPredicate. I create to static properties, one for returning all objects and one for none:

```
import Foundation
extension NSPredicate {
    static let all = NSPredicate(format: "TRUEPREDICATE")
    static let none = NSPredicate(format: "FALSEPREDICATE")
}
```

Back in ContentView, I can use the Notes fetch function together with @FetchRequest. This is shorter and simpler. It also allows us to later test the fetch request separately from SwiftUI with Unit Tests.

```
@FetchRequest(fetchRequest: Note.fetch(.all)) private var notes: FetchedResults<Note>
```

Creating New Data

To create a new Note object, you must use a managed object context. This context indicates which database the object belongs to.

Update the addItem function in ContentView:

```
struct ContentView: View {

   @Environment(\.managedObjectContext) private var viewContext
   @FetchRequest(…) private var notes: FetchedResults<Note>
```

```
        var body: some View {
          …
        }

        private func addItem() {
             withAnimation {
                 let _ = Note(title: "New Note", context: viewContext)

                 do {
                     try viewContext.save()
                 } catch {
                     let nsError = error as NSError
                     fatalError("Unresolved error \(nsError), \(nsError.userInfo)")
                 }
             }
         }
    }
```

I am using the convenience initializer that we added to the Note extension in Note+Helper.swift file. The creationDate property is automatically set.

Notice that I only create a new CoreData object, I am not actually doing anything with it. Once the object is added to the context, the @FetchRequest property wrapper gets the newly updated data and displays the notes array including the newly created object.

This is a very neat automatically updating behavior. We don't have to worry about view updates. SwiftUI together with CoreData takes care of this for us 💪.

Deleting Data

To delete an object, inform the context to remove it. You can get the context to which the object belongs to by checking the managedObjectContext property:

```
    if let context = note.managedObjectContext {
        context.delete(note)
    }
```

In ContentView, swipe-on-delete is used that pass the offsets:

```
    List {
        ForEach(notes) { note in
            NavigationLink {
                Text("Item at \(note.creationDate!, formatter: itemFormatter)")
            } label: {
                Text(note.creationDate!, formatter: itemFormatter)
            }
        }
        .onDelete(perform: deleteItems)
    }
```

find the deleteItems function and update it to use the notes data now:

```
    private func deleteItems(offsets: IndexSet) {
        withAnimation {
            offsets.map { notes[$0] }.forEach(viewContext.delete)

            do {
                try viewContext.save()
            } catch {
                let nsError = error as NSError
                fatalError("Unresolved error \(nsError), \(nsError.userInfo)")
            }
        }
    }
```

The map and foreach operation iterates over all objects and calls viewContext.delete function for each of them.

Updating Data

Updating an existing Note object is as simple as modifying its properties:

```
note.title = "Updated Title"
```

SwiftUI views like TextFields and Pickers require a Binding. Let's see how this works together with CoreData. First, create a new SwiftUI view NoteDetailView and define a note property:

```
struct NoteDetailView: View {

    @ObservedObject var note: Note

    var body: some View {
        TextField("title", text: $note.title)
              .textFieldStyle(.roundedBorder)
    }
}
```

To ensure your SwiftUI views reflect these updates, you can mark your Note object as an @ObservedObject. CoreData uses objects and marks that as ObservableObject. Right-click on "Note.creationDate" and press "jump to definition". You should see something like this:

```
import Foundation
import CoreData

extension Note {

    @nonobjc public class func fetchRequest() -> NSFetchRequest<Note> {
        return NSFetchRequest<Note>(entityName: "Note")
    }

    @NSManaged public var creationDate: Date?
    @NSManaged public var title: String?

}
```

```
extension Note : Identifiable {

}
```

This is automatically generated code. As you can see Note conforms to Identifiable. Each CoreData object has an ObjectIdentifier that is used for Identifiable.

Handling Optional Attributes

We have defined 2 attributes in ".xcdatamodeld" that are shown here as properties. CoreData refers to "attributes", but you can basically see them as properties. Both **properties are optional**. You can set properties as non-optional in the ".xcdatamodeld" file, but that means in the database level non-optional. This is not equal to Swift's non-optional.

SwiftUI Textfield works with non-optional String and throughs an error. You can define a Binding property in body like so:

```
struct NoteDetailView: View {

   @ObservedObject var note: Note

   var body: some View {

        let textBinding = Binding(
            get: { note.title ?? "" },
            set: { note.title = $0 }
            )

            return TextField("title", text: textBinding)
                  .textFieldStyle(.roundedBorder)
    }
}
```

This works but is not very practical. I don't want to add this extra code in all my SwiftUI views. Instead, we are using again a little bit of syntactic sugar. First I am going to rename the title property in ".xcdatamodeld" to title_. The underbar indicates that this property is the optional property. Now I want to add a "title" property that is non-optional and uses "title_". It is a wrapper. In Note extension add a computed property with a getter and setter:

```
extension Note {

    var title: String {
        get {
            self.title_ ?? ""
        }
        set {
            self.title_ = newValue
        }
    }

    …
}
```

The getter is accessing the stored attribute “titel_”. If it is optional, it returns an empty string. When I set the title computed property, it changes the underlying “title_” attribute in the database.

Now with this changes, I can update NoteDetailView to use the new title property:

```
struct NoteDetailView: View {

   @ObservedObject var note: Note

   var body: some View {
        TextField("title", text: $note.title)
              .textFieldStyle(.roundedBorder)
    }
}
```

This is much better and in the style of SwiftUI. You will see that I use this property wrapper to handle optional CoreData attributes a lot in section 3. It works well and makes working with SwiftUI a lot easier. Although you need to do more setup work, it pays off in the long run. We handle the optional nil values gracefully and prevent app crashes.

Update ContentView to use NoteDetailView and test changing the title of the note:

```
struct ContentView: View {
    …

    var body: some View {
        NavigationView {
            List {
                ForEach(notes) { note in
                    NavigationLink {
                       NoteDetailView(note: note)
                    } label: {
                        Text(note.creationDate!, formatter: itemFormatter)
                    }
                }
                .onDelete(perform: deleteItems)
            }
        }
    }
}
```

SwiftUI Preview

The Preview in SwiftUI will require you to provide CoreData objects that belong to a view context. For example, you can use preview in NoteDetailView by creating the preview view context. You can then use it to create a new Note object and pass it to NoteDetailView like so:

```
#Preview {
   let context = PersistenceController.preview.container.viewContext

   return  NoteDetailView(note: Note(title: "test", context: context))
}
```

If you use @FetchRequest or access the context from the environment like in ContentView:

```
struct ContentView: View {

    @Environment(\.managedObjectContext) private var viewContext

    @FetchRequest(fetchRequest: Note.fetch(.all))
    private var notes: FetchedResults<Note>

    …
}
```

You also need to set it in the environment of the view:

```
#Preview {
    ContentView().environment(\.managedObjectContext,
                              PersistenceController.preview.container.viewContext)
}
```

Mastering CRUD operations with Core Data in SwiftUI is crucial for building dynamic and responsive applications. By creating a clear and structured data model, and understanding how to create, read, update, and delete data, you can efficiently manage your app's data layer. Remember to handle optional values gracefully, and use SwiftUI's property wrappers to keep your UI in sync with your data.

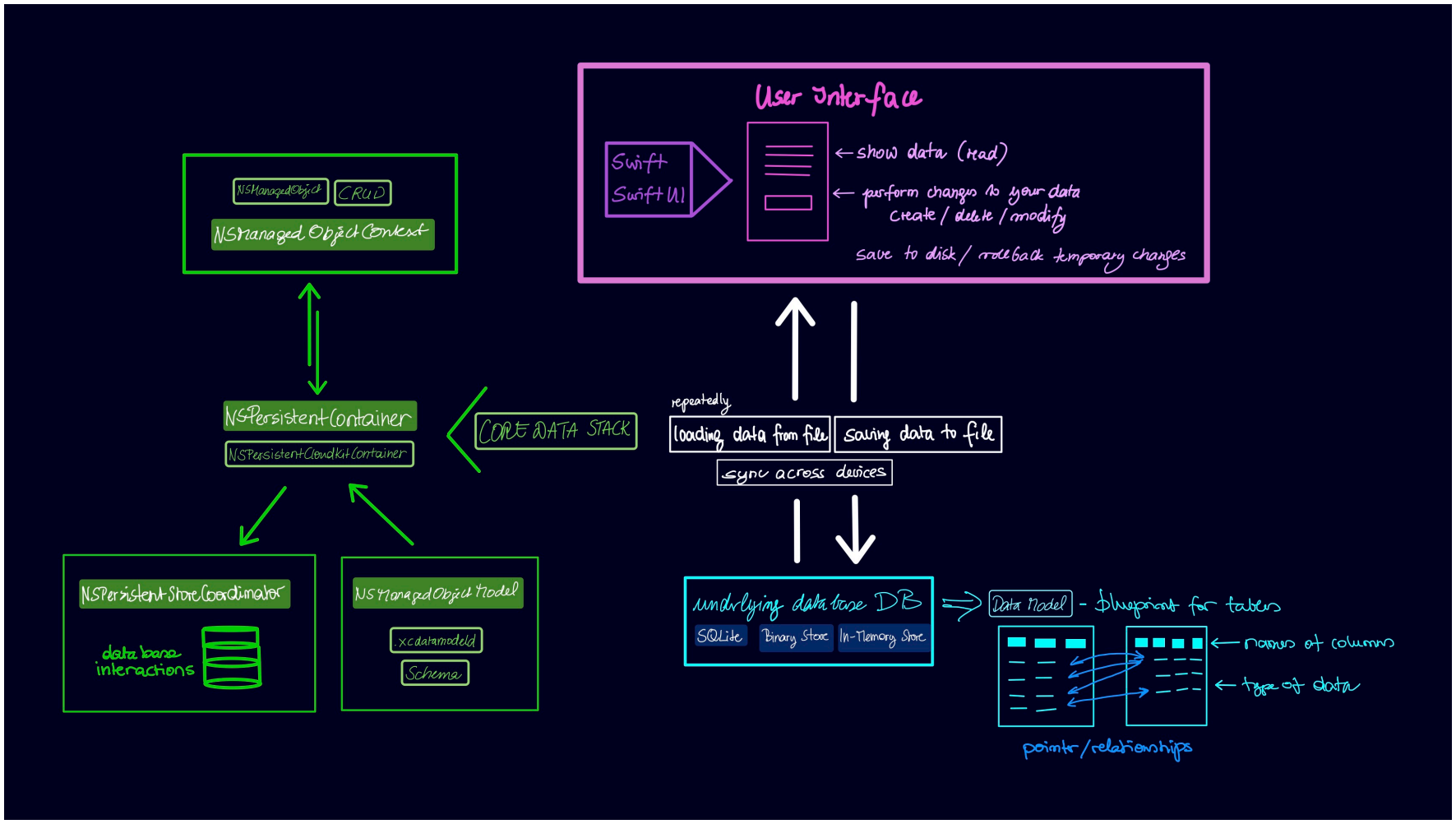
1.4 CORE DATA STACK

Apple's Core Data framework is a powerful tool for iOS developers to persist data within their applications. It includes several classes designed to simplify the process of data management, so you don't have to write extensive code from scratch. If you've looked at a template project, you've probably noticed that it contains very little code. I'm here to demystify what Apple has provided for us, particularly the Persistent CloudKit container and its components.

The Core Data Stack Components

The Core Data Stack comprises four main components that work in unison to manage your data and the underlying database on the file system:

1. Persistent Container
2. NSManagedObjectContext
3. NSManagedObjectModel
4. Persistent Store Coordinator



Let's break down each component and understand its role.

Persistent Container

The Persistent Container is the central piece of the Core Data Stack. Think of it as the team leader. When you instantiate this container in your app, it automatically sets up the other three components. Here's how you create a Persistent Container in your app:

```
    let container = NSPersistentContainer(name: "ModelName")
    container.loadPersistentStores { (storeDescription, error) in
        if let error = error as NSError? {
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    }
```

Replace "ModelName" with the name of your data model file.

NSManagedObjectModel

The NSManagedObjectModel represents the data model schema you've defined in your **.xcdatamodeld** file. It describes the entities and their relationships in your app. You don't typically interact with it directly. An instance of NSManagedObjectModel is created by the container.

Persistent Store Coordinator

The Persistent Store Coordinator is responsible for the actual file management and communicates with the data store. It handles all the low-level SQL operations, so you don't have to worry about writing SQL code.

The Persistent container will create a coordinator automatically. You will not interact with the coordinator directly. Just know that it is there for you to do the heavy lifting.

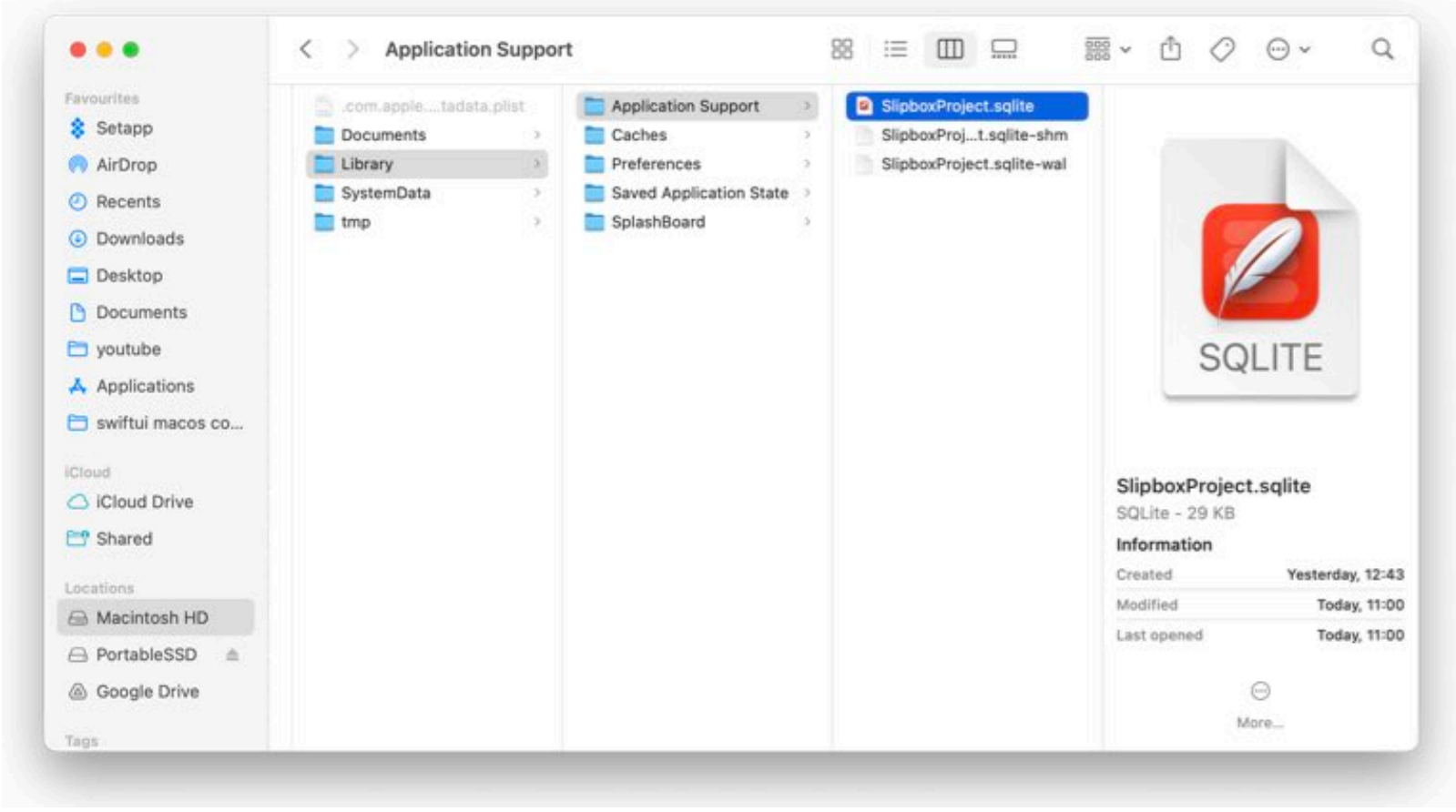
As an example, we will use it to get the location of the SQLite file. Change PersistencController to print the file URL on launch:

```
struct PersistenceController {

     …

    init(inMemory: Bool = false) {
        container = NSPersistentCloudKitContainer(name: "SlipboxProject")
         …

        if let url = container.persistentStoreCoordinator.persistentStores.first?.url  {
            print("url:" + urlL.absoluteString)
        }
    }
}
```

Run the app. You should see the URL in the debug area. Here is an example:

```
url:file:///Users/karinprater/Library/Developer/CoreSimulator/Devices/C09B976D-95A2-41EA-B92E-7504F3C01302/data/Containers/Data/Application/AC1C60E9-0AE8-4737-84E0-49907BFEA3E8/Library/Application%20Support/SlipboxProject.sqlite
```

Open Finder and use Go >> Go To Folder to open the file location:



Since the coordinator is responsible for the database on file. It holds the information about what stores we use and where they are located.

In section 3 after you will have added more data to the model, you will open the SQLite file and look how the data is stored.

NSManagedObjectContext

The NSManagedObjectContext is what you'll interact with most frequently. It is also created automatically from the container. You can access like so:

```
let context = container.viewContext
```

It's like a staging area for working with your managed objects. You fetch, create, update, and delete objects through the context. Here's a simple example of how to create a new entity:

```
let newEntity = Entity(context: context)
```

Saving to File

With Core Data, you can focus on Swift and let Core Data handle the complexity of persistent storage. As long as you use the view context in SwiftUI to update or delete objects, these changes are temporary and in memory. They are not written to the database file.

In order to save the changes to file, you need to call save on the view context like so:

```
    let context = container.viewContext
    guard context.hasChanges else { return }

    do {
        try viewContext.save()
    } catch {
        let nsError = error as NSError
        fatalError("Unresolved error \(nsError), \(nsError.userInfo)")
    }
```

This code checks if there are any changes in the context and saves them. The container will then pass the save request to the coordinator who handles saving and syncing with iCloud.

We will go into more detail about when and how to save in your SwiftUI project in the next section.

Advanced Core Data: Multiple Contexts

For advanced operations, such as performing tasks in the background, you can create multiple contexts. However, remember that contexts are not thread-safe. You should never pass managed objects between contexts directly. Instead, pass the object's identifier and fetch it again from the other context.

1.5 SAVING YOUR USERS DATA CORRECTLY

In this section, I'm going to walk you through the right way to save your users' data using Core Data in a SwiftUI project. I've noticed some issues with the current saving mechanism, and I want to clarify how to handle data persistence properly.

Let's dive into the problem. You might recall that in our notes app, we could create and modify note titles. Although new notes were being saved and persisted across app launches, updates to existing note titles weren't being retained. This discrepancy is because Core Data works with a context that temporarily caches changes—they aren't saved to the actual database until you explicitly tell Core Data to do so.

Understanding Core Data Context

When you make changes to your managed objects, Core Data stores these changes in a managed object context. It's like a scratchpad. These changes are not permanent until you call the save() method on the context. Here's a snippet showing how you might have used save() when creating a new note:

```
let newNote = Note(context: managedObjectContext)
newNote.title = "New Note Title"
do {
    try managedObjectContext.save()
} catch {
    // Handle the error appropriately in your app
}
```

In the above example, when you tap the "plus" button to add a new note, it's directly saved, and that's why it's persisted. However, when you change the title of an existing note, you need to call save() again to persist that change.

Efficient Saving Strategy

You don't want to save after every single keystroke or change. That could lead to performance issues and could be inefficient. Instead, think about the moments in your app's lifecycle when it's essential to save—typically when the app is going into the background.

Lifecycle Event Handling in SwiftUI

In SwiftUI, you can detect when your app moves to the background using the scenePhase environment value. Here's how you can observe the scene phase changes:

```
@main
struct SlipboxProjectApp: App {

    let persistenceController = PersistenceController.shared
```

```
        @Environment(\.scenePhase) var scenePhase

        var body: some Scene {
            WindowGroup {
                ContentView()
                    .environment(\.managedObjectContext,
                                 persistenceController.container.viewContext)
            }
            .onChange(of: scenePhase) { newScenePhase in
                if newScenePhase == .background {
                    persistenceController.save()
                }
            }
        }
    }
```

The onChange modifier is called every time the scene phase changes. By checking for .background, you can save your context right before the user leaves the app.

Adding a Convenience Save Function

To avoid repetitive code, add a convenience save() function to your PersistentController. This function should only save if there are changes:

```
    import CoreData

    struct PersistenceController {

        static let shared = PersistenceController()

        let container: NSPersistentCloudKitContainer

        init(inMemory: Bool = false) {
            …
        }

        func save() {
            let context = container.viewContext
            guard context.hasChanges else { return }

            do {
                try context.save()
            } catch {
                let nsError = error as NSError
                fatalError("Unresolved error \(nsError), \(nsError.userInfo)")
            }
        }
    }
```

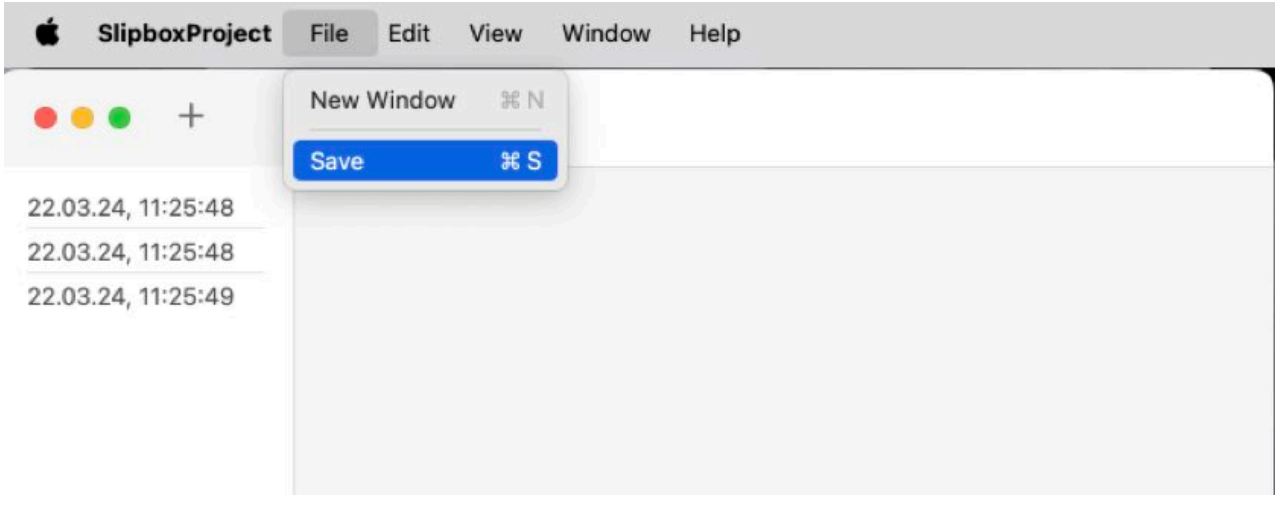
Now you can call PersistentController.shared.save() wherever necessary, like when the app enters the background or when a view is about to disappear.

```
struct NoteDetailView: View {

    @ObservedObject var note: Note

    var body: some View {
       …
      .onDisappear {
          PersistenceController.shared.save()
      }

    }
}
```

Handling Saves on macOS

For macOS apps, you can add save commands in the menu bar, providing users with a familiar way to trigger saves. Here's how you can add a "Save" command with a keyboard shortcut:

```
@main
struct SlipboxProjectApp: App {

    let persistenceController = PersistenceController.shared

    @Environment(\.scenePhase) var scenePhase

    var body: some Scene {
        WindowGroup {
            ContentView()
                ..
        }
        .onChange(of: scenePhase) { … }
        .commands {
            CommandGroup(replacing: .saveItem) {
                Button("Save") {
                    persistenceController.save()
                }
                .keyboardShortcut("S", modifiers: [.command])
            }
        }
    }
}
```



Cloud Sync Considerations

If you're using iCloud sync, remember that calling save() pushes changes to the underlying database, which can then sync across devices. If you want a live sync experience, you might need to call save() more frequently, but always be mindful of the performance implications.

Saving your users' data correctly is crucial for any app. By understanding Core Data's context and lifecycle events in SwiftUI, you can ensure that data is persisted at the right moments. Use the provided code snippets as a guide to implement an efficient and user-friendly saving mechanism in your app. Remember, the goal is to create a smooth and reliable experience for your users, so they trust your app with their valuable data.

2. UNIT TESTS

2.1 INTRODUCTION TO UNIT TESTING

In this section, I want to dive into the world of unit tests and test-driven development (TDD). You're going to learn how to apply these concepts specifically to Core Data within a SwiftUI project.

Unit testing is crucial when it comes to verifying the business logic of your application, especially for actions like creating new Core Data objects. You'll be asking questions such as:

- Did I set all my properties correctly?
- Am I handling deletions as expected?
- Does my fetch request return objects in the correct sort order?

These tests are great for getting extra practice with Core Data and ensuring your app's stability as it grows.

Writing Tests

You'll write these tests outside of your main application code, in a separate test target. This ensures that your test code doesn't mingle with your production code and can be managed independently.

Principles of Testing

When I write tests, I follow **FIRST**:

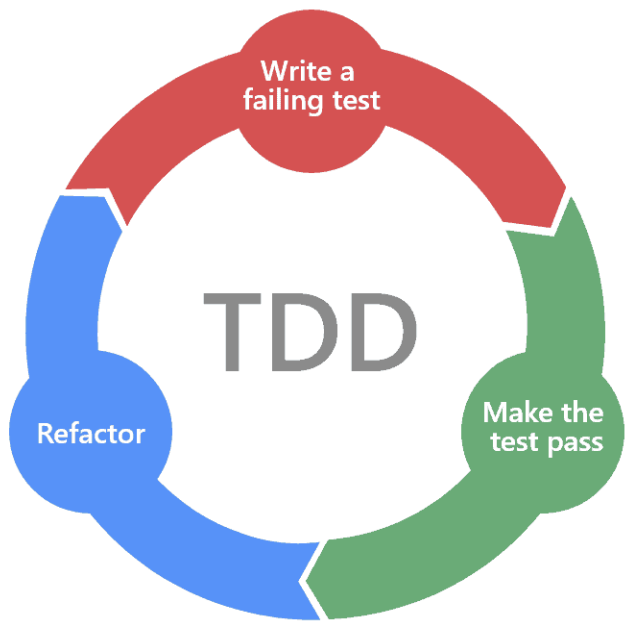
1. **Fast Execution**: Tests should run quickly. If you're waiting minutes to run your tests every time, it's inefficient. Fast tests encourage frequent testing, which is vital as your app scales and complexity increases.
2. **Isolation**: Each test must be independent. If tests affect each other, the results can be unreliable. Isolation ensures that the outcome of one test won't impact another.
3. **Repeatability**: Tests should produce the same results every time. Consistent results are a sign of reliable tests.
4. **Self-Verification**: Tests should automatically assert whether they've passed or failed. You shouldn't have to manually inspect console logs or outputs to determine if the test was successful.
5. **Timely**: Write your tests first so they can act as a blueprint for the functionality you add.

Test-Driven Development (TDD)

If you follow TDD, you'll begin by writing a test that fails because the functionality it's testing doesn't exist yet. Then, you'll write the code to make the test pass. This cycle of write-fail-refactor-pass ensures that your codebase is thoroughly tested and functions as expected.

Benefits of TDD

- **Documentation**: Your tests serve as a guide on how to use your code.
- **Stability**: Regular testing leads to fewer bugs and a more stable application.
- **Collaboration**: Tests help other developers understand your code and prevent them from introducing errors.



Moving Forward

Throughout this book, I'll provide examples of unit tests for the Core Data model. As we progress and make changes to the schema, I'll demonstrate how to incorporate TDD into these updates. However, due to time constraints, I won't cover every possible test scenario. I encourage you to write additional tests to solidify your understanding and to ensure the robustness of your Core Data implementation.

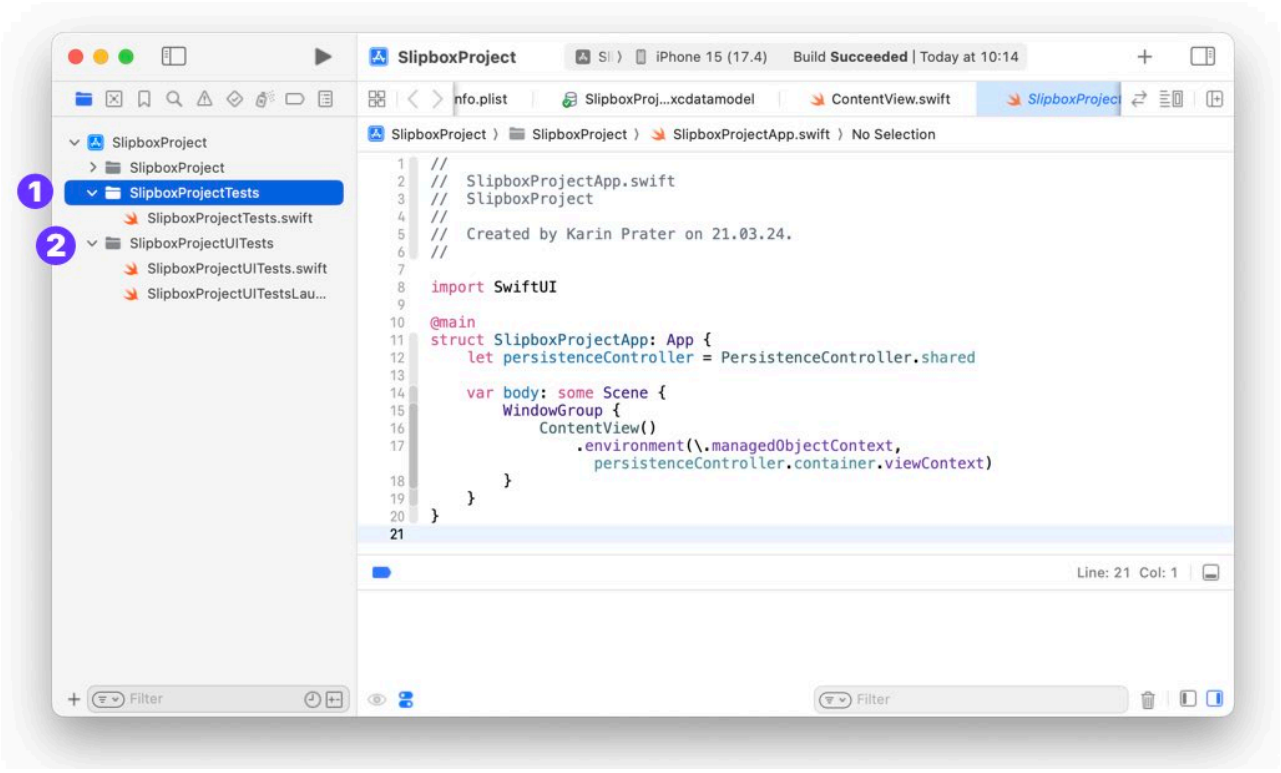
Remember, unit testing is like a playground for Core Data. It's separate from your UI code, allowing you to focus on the data layer of your app without distractions.

2.2 WRITE YOUR FIRST UNIT TEST FOR CORE DATA

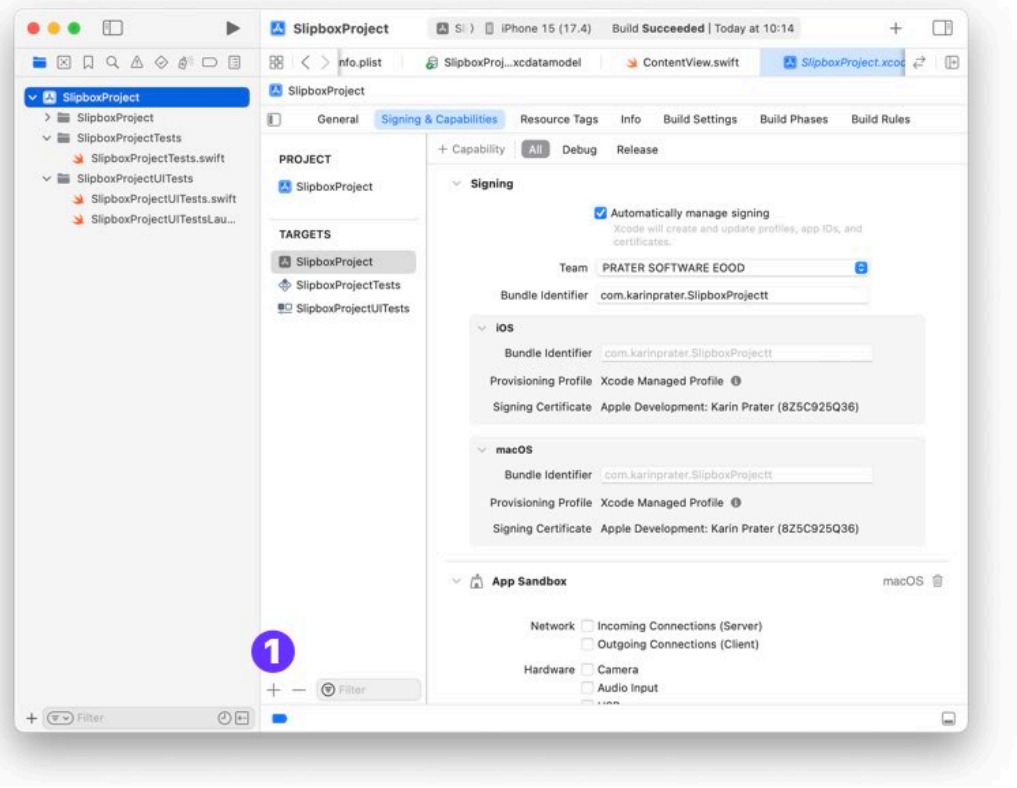
Unit testing is an essential aspect of iOS development, allowing you to verify that your Core Data code and business logic work as expected. In this section, I'll guide you through the process of writing your first unit test for Core Data within a SwiftUI project.

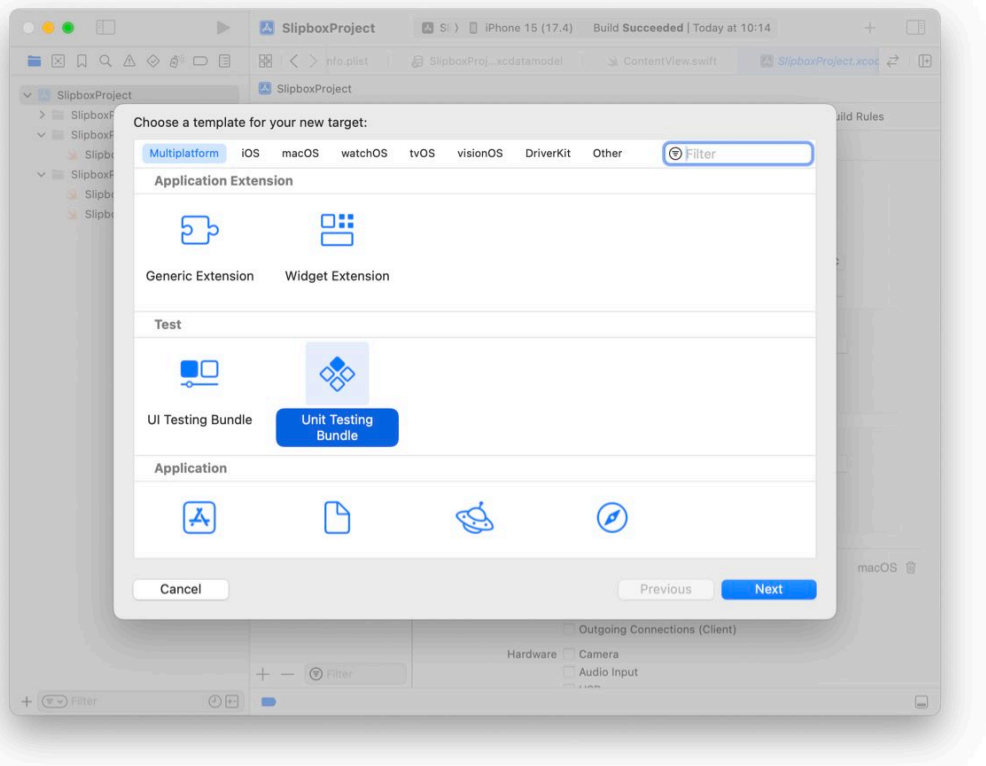
Setting Up the Test Environment

When you create a new project in Xcode and include a test target, you'll find a couple of test folders in your project structure: one for **(1) unit tests** (YourProjectTests) and another for **(2) UI tests** (YourProjectUITests).



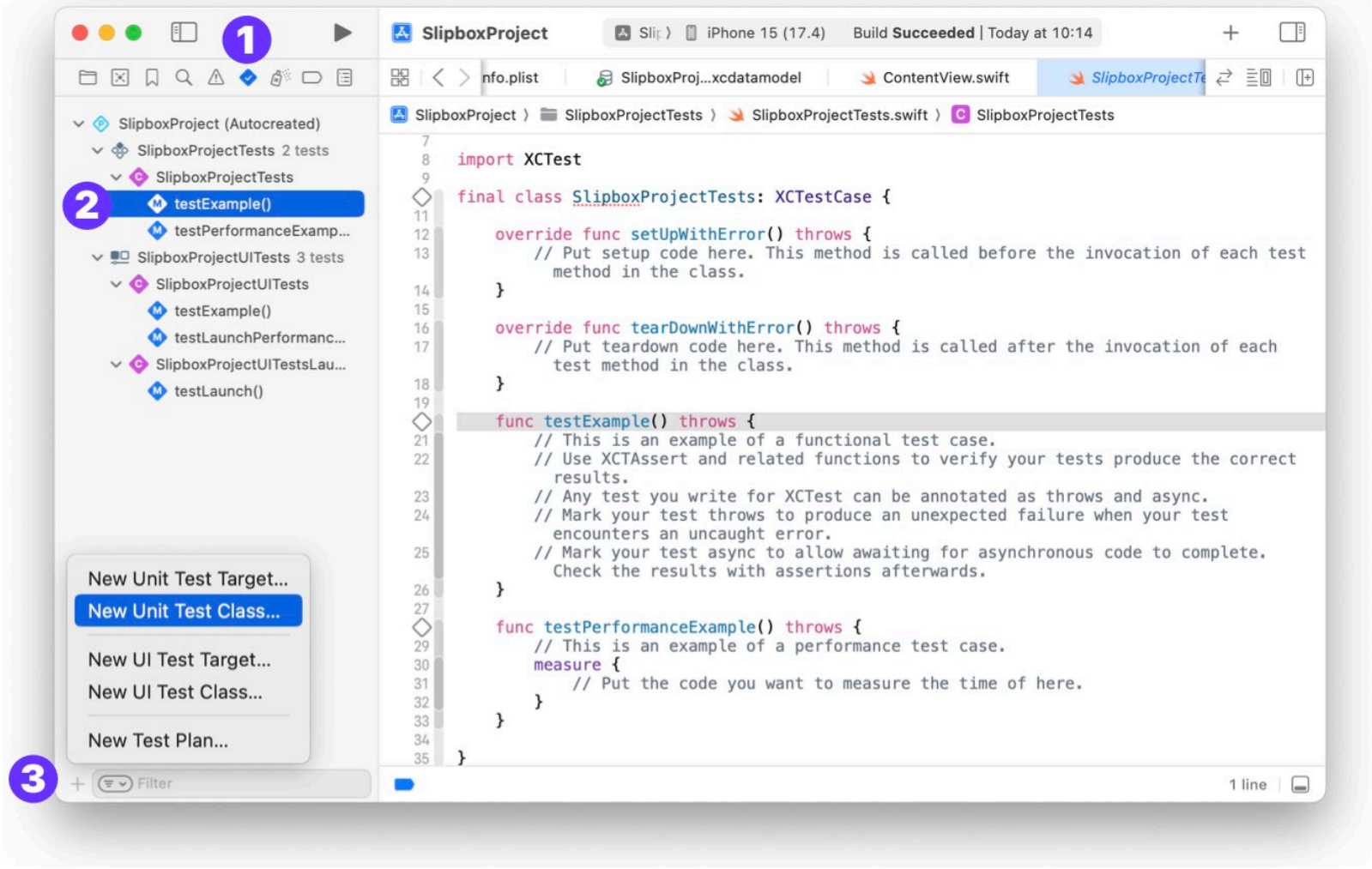
If you didn't add unit tests when setting up your project, don't worry. You can add them at any time. Tap on the "+" button in the project target area, select "New Unit Test Target," and Xcode will set up the necessary files for you.





Open the **testing navigator area (1)**. You will see all test bundles. Each bundle can have multiple test classes. Each test class in the test target can focus on a different aspect of your code. For instance, you might have one class for testing note functionality and another for testing folder relationships.

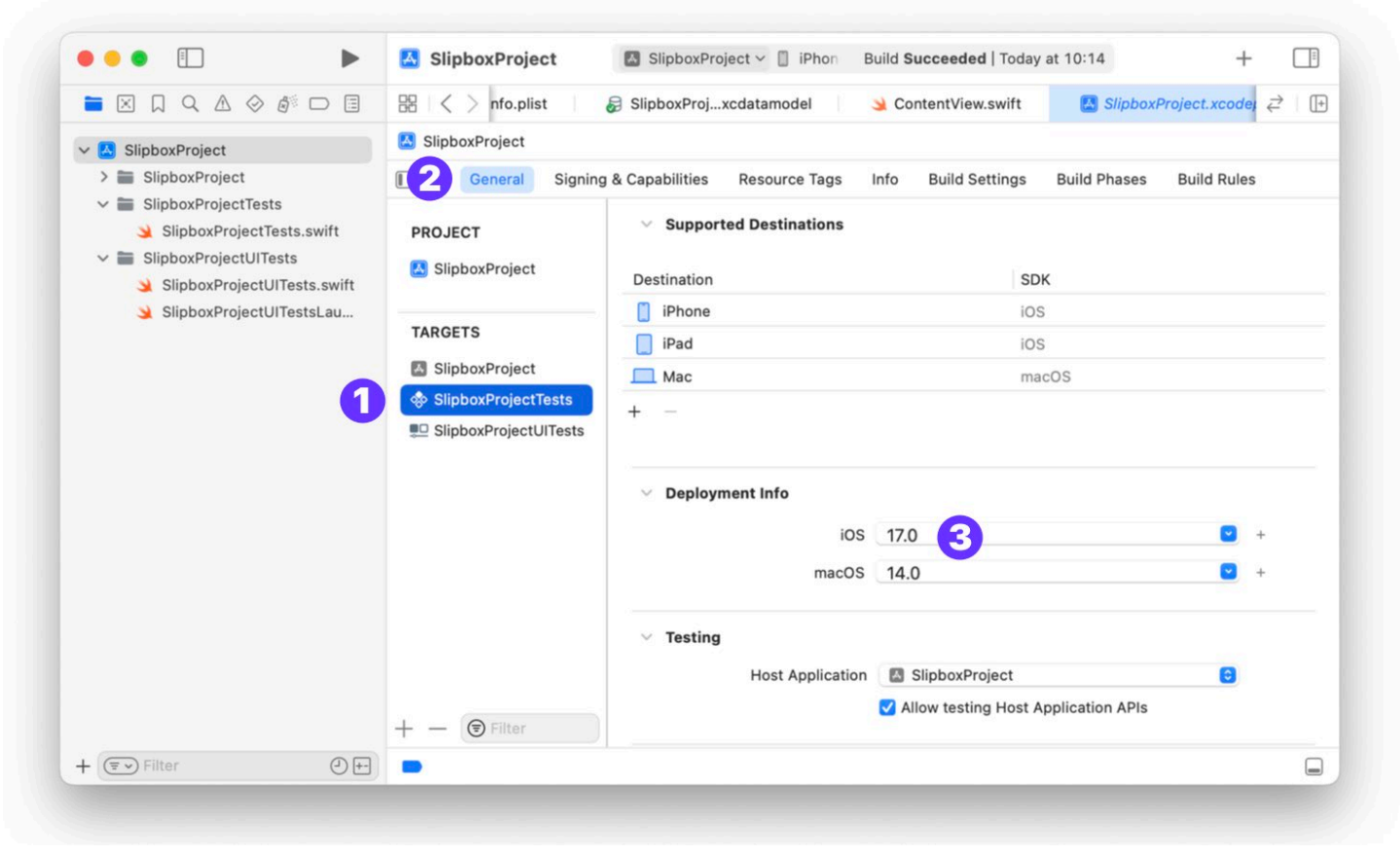
Xcode automatically added a template test class “SlipboxProjectTests” with 2 test functions **“testExamples” and “testPerformanceExample” (2):**



You can also generate a new test class by pressing the **“+” button in the bottom left corner (3)** and choosing “New Unit Test class”. Create a new class and name it “NoteTests”.

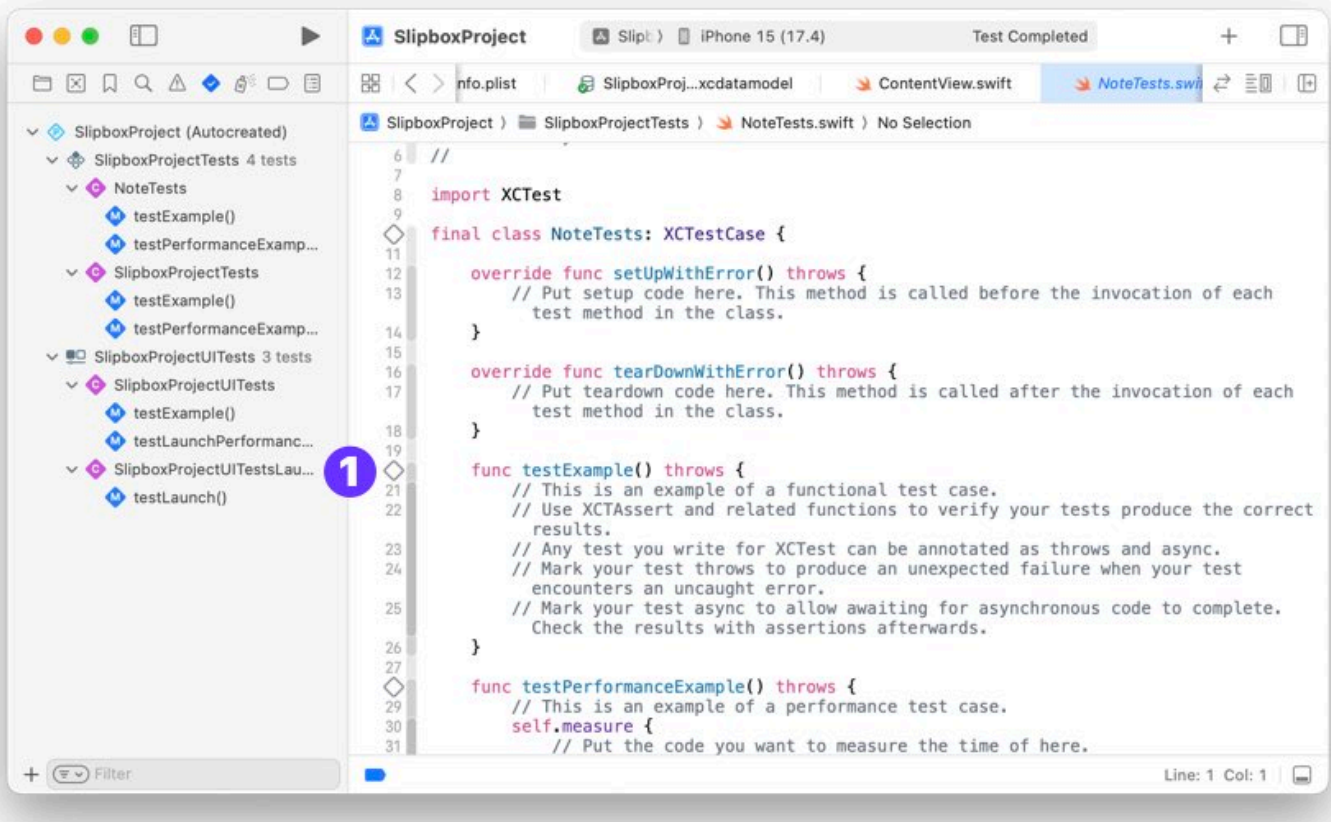
Configuring the Test Target

Before writing tests, ensure your test target's deployment info matches your simulator's iOS version. You can check this in the project settings under the test target.



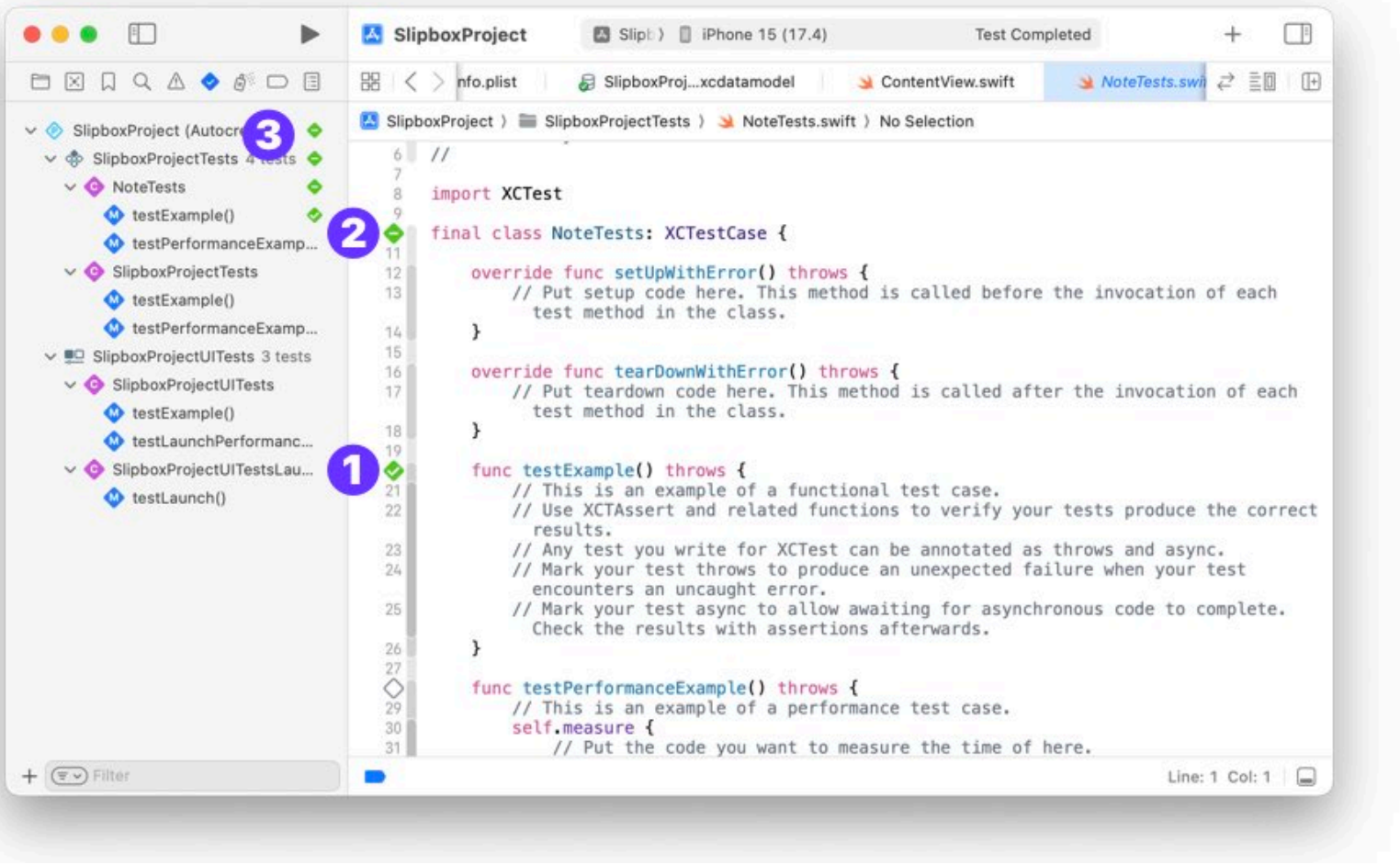
Writing a Unit Test

In your test class, you'll notice small diamonds to the left of each test function. Clicking on these will run the tests. Each test function must start with the word "test", for Xcode to recognize them as a test function. Open NoteTests test class:



You will see the diamonds icon next to the test functions. Hover over these diamonds icon to see a play button, which you can click to run individual tests. Press the one next to “testExample” (1). Xcode will build and run the simulator.

After the test execution, you will see the results. In the following, the test passes and is **marked green (1):**



If all test functions in a class pass the class is also marked with a **green marker (2)**. In the **navigator area (3)** on the right you can also see these markers for all test classes and functions that were executed.

The template function is not doing anything. So let’s write unit tests with CoreData.

Setup and Teardown

Each test class has a setup and teardown function. I want to use a testing CoreData store. First, let’s create a function in PersistenceController that initializes a new in-memory store:

```
struct PersistenceController {

    static let shared = PersistenceController()

    init(inMemory: Bool = false) {
       …
    }

    static func createEmpty() -> PersistenceController {
        return PersistenceController(inMemory: true)
    }
}
```

Next, I will go to the “NotesTestClass” and import CoreData and my project:

```
import XCTest
@testable import SlipboxProject
import CoreData

final class NotesTests: XCTestCase {
    // add test functions here
}
```

I then change then create a new property controller that holds the in-memory controller. In the setup function, I am initializing a new controller. This function is called when I press the play button for a test.

In the teardown, I am setting the controller to nil. This function is executed after all test functions are finished for this class:

```
final class NotesTests: XCTestCase {

    var controller: PersistenceController!

    var context: NSManagedObjectContext {
        controller.container.viewContext
    }

    override func setUpWithError() throws {
        self.controller = PersistenceController.createEmpty()
    }

    override func tearDownWithError() throws {
        self.controller = nil
    }

    // add test functions here
}
```

I also added a computed property “context” that gets the context from the controller. This is a shortcut. We will be using the context for our tests, so making it easier and shorter to access it will help in the following test functions.

Writing a Test Function

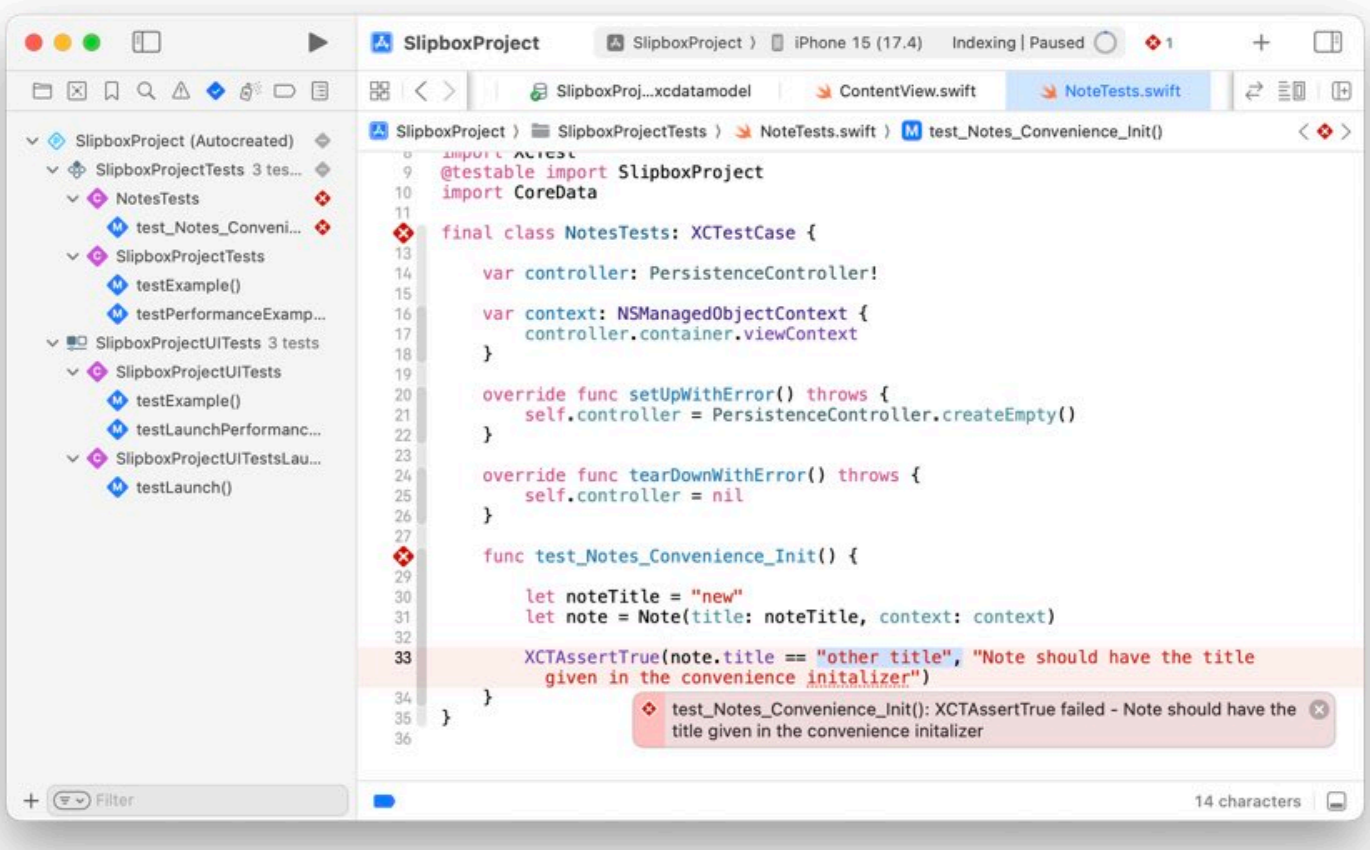
Now, let's write a test to check if a note is created with the correct title and a creation date. This is the code we added earlier and that we want to test now:

```
extension Note {
    convenience init(title: String, context: NSManagedObjectContext) {
        self.init(context: context)
        self.title = title
    }

    public override func awakeFromInsert() {
        self.creationDate = Date()
    }
}
```

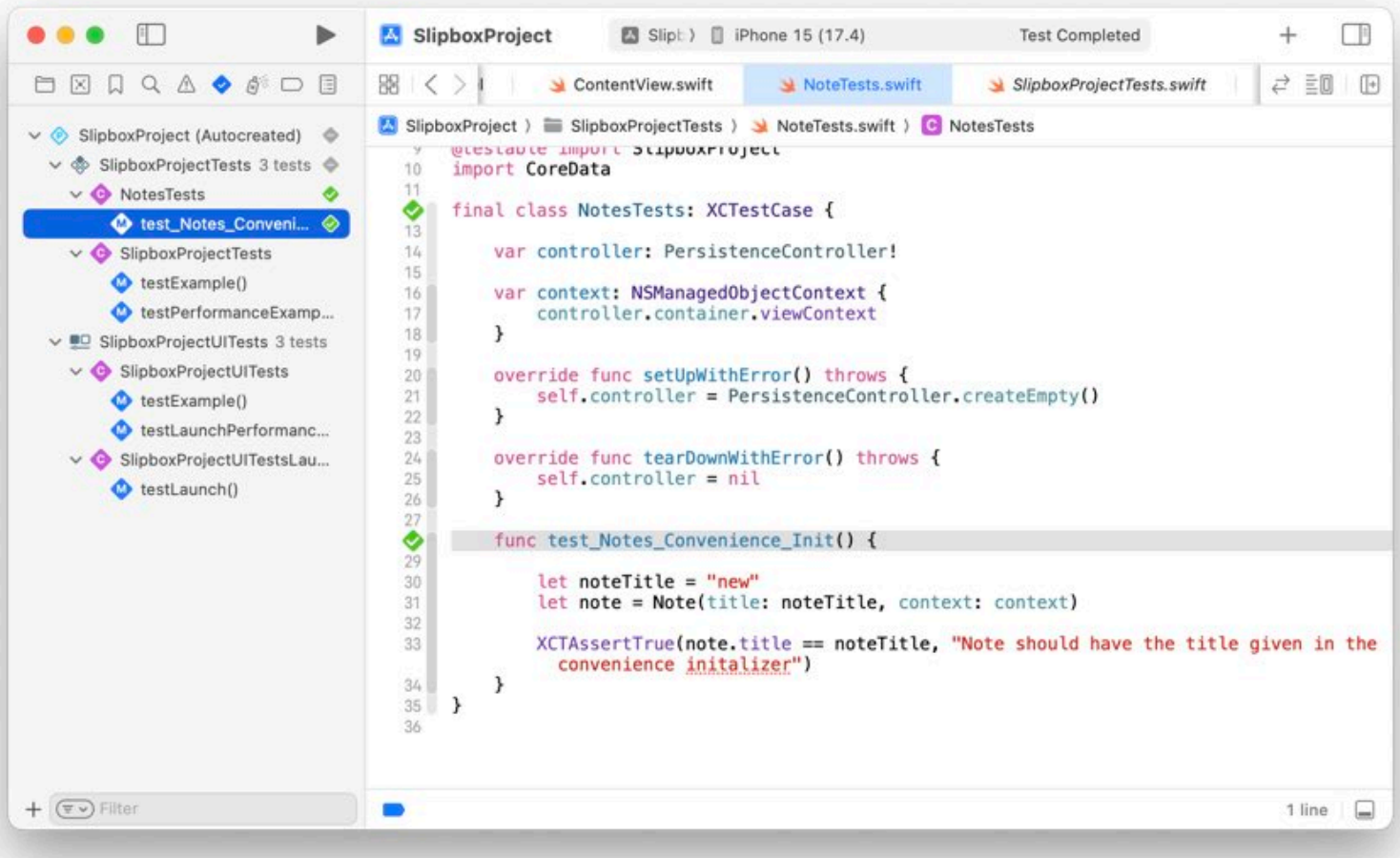
First, I want to test if my convenience initialiser sets the title property. I am adding a test, create a new note with the setup context:

```
    func test_Notes_Convenience_Init() {

        let noteTitle = "new"
        let note = Note(title: noteTitle, context: context)

        XCTAssertTrue(note.title == noteTitle, "Note should have the title given
                                                in the convenience initalizer")
    }
```

XCTAssertTrue checks if the note's title matches what we set. If it fails, the error message that I added will be shown in Xcode. In the following, I changed the check on purpose to fail and the test failed:



Using the correct test condition will result in the test passing:



Now add a test function to see if the creationDate property is set:

```
    func test_Notes_CreationDate() {

        let note = Note(context: context)

        let noteConvenient = Note(title: "new", context: context)

        XCTAssertNotNil(note.creationDate, "notes should have creationDate property")
        XCTAssertNotNil(noteConvenient.creationDate)
    }
```

I created to note objects with the default init(context:) and the convenience init(title:context:)

XCTAssertNotNil confirms that both note objects have a non-nil creationDate. Run the test and you should see it passing.

Remember to make your tests fail intentionally by changing conditions to ensure they are working correctly. For example, comment out the line that sets the creationDate and run the test again. It should fail, indicating the test is effectively checking the creation date's existence.

Unit testing is crucial for maintaining a stable codebase, especially when dealing with Core Data. By following the steps outlined above, you've written tests to verify that your Core Data managed objects are created correctly. As you continue developing your app, keep writing tests for new functionalities to ensure everything works as expected. Remember, the key is to make your tests clear, concise, and focused on one aspect of your code at a time.

2.3 WRITE A UNIT TEST TO CHECK FETCHREQUEST FOR OUR NOTE'S OBJECTS

When working with Core Data and SwiftUI, it's crucial to ensure that your data updates correctly. I've had moments of doubt—wondering if a property didn't update due to a mistake in my Core Data handling or an issue with SwiftUI's views. To clear up such uncertainties, I write unit tests.

Testing Property Updates

First, let's write a test to confirm that updating the properties of a Note object works as expected. I'll create a Note with an initial title and then change it. The test will verify that the update is successful.

```
    func test_Notes_Updating_Title() {

        let note = Note(title: "old", context: context)

        // Update the Note's title
        note.title = "new"

        XCTAssertTrue(note.title == "new")
    }
```

Running this test with the correct title "New Title" should pass. If I mistakenly check for "Old Title", the test will fail, which is expected behavior.

Testing Core Data Functionality

It might seem redundant to test Core Data's default behavior, but it's not about testing Core Data itself. I focus on the logic I've written. For instance, I don't need to test the basic fetch functionality, but I should test the custom fetch requests I've implemented, especially since predicates can be error-prone.

Testing Fetch Requests with Predicates

I created a fetch function in the Note extension to generate a NSFetchRequest. Let's now test this function together with the Predicates for "none" and "all" that we defined in section 1.4:

```
    extension Note {

       …

       static func fetch(_ predicate: NSPredicate = .all) -> NSFetchRequest<Note> {
            let request = NSFetchRequest<Note>(entityName: "Note")
            request.sortDescriptors = [NSSortDescriptor(keyPath: \Note.creationDate,
                                                        ascending: true)]
            request.predicate = predicate

            return request
        }
    }
```

To test fetching, I need something in the database. I'll insert a Note and then create a fetch request to retrieve it. You can use the fetch request together with the view context and fetch all objects:

```
    func test_Fetch_All_Predicate() {
        _ = Note(title: "default note", context: context)
        let fetch = Note.fetch(NSPredicate.all)

        let fetchedNotes = try? context.fetch(fetch)

        XCTAssertNotNil(fetchedNotes)
        XCTAssertTrue(fetchedNotes!.count > 0,
                      "Predicate of all should fetch at least 1 object")
    }
```

To check the test results I am using XCTAssertNotNil. If the fetched notes property is nil, I did not access anything. In the above test, I am setting the predicate to all and should get all objects of type Note. If fetchedNotes is nil, the test fails. You could also check how many objects are returned. E.g. I am checking if the count of the fechtedNotes is larger than 0.

In this test, I've used a forced unwrap (!) because I've already confirmed that the array isn't nil. If there's an issue, the test will fail, which is what I want.

The opposite of fetching all is fetching none. In the below test, I am creating a fetch request with the none predicate. The test passes when the fetchedNotes count is 0:

```
    func test_Fetch_None_Predicate() {

        _ = Note(title: "default note", context: context)
        let fetch = Note.fetch(NSPredicate.none)

        let fetchedNotes = try? context.fetch(fetch)

        XCTAssertNotNil(fetchedNotes)
        XCTAssertTrue(fetchedNotes!.count == 0,
                      "Predicate of none should not fetch any objects")
    }
```

Handling Optional Results

In production code, you'd handle optionals more carefully, but in unit tests, it's acceptable to force unwrap when you know it won't be nil. This approach simplifies the test and makes failures clear and actionable.

Writing Good Tests

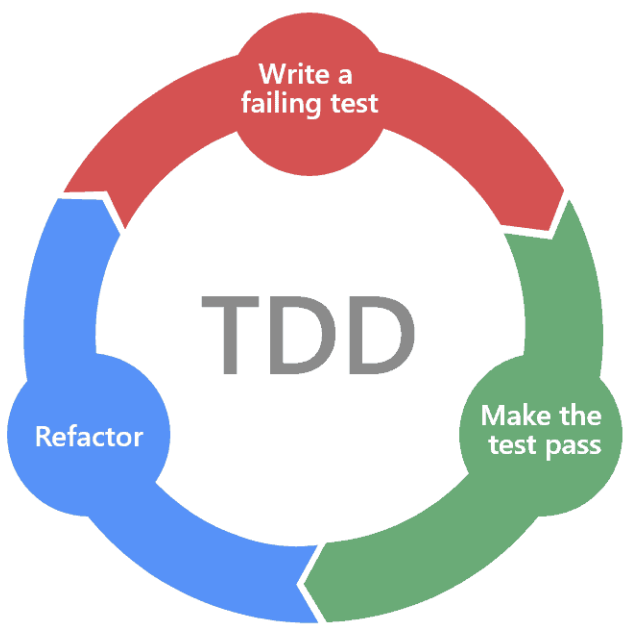
Writing good unit tests is a skill. I aim to isolate each test to check one aspect at a time. For example, if a test fails, I want to know exactly what went wrong—was it the predicate or the fetch request setup?

Remember to name your tests descriptively. For instance, testFetchAllPredicate and testFetchNonePredicate clearly indicate what each test is checking.

2.4 PRACTICING TDD

In the previous lessons, I approached testing in a traditional way: writing tests after the business logic was in place. However, Test-Driven Development (TDD) flips this process on its head. You start with the test, and then you write the logic, functions, and code necessary to pass that test.

Let's practice TDD with a common task: deleting a note from the database.



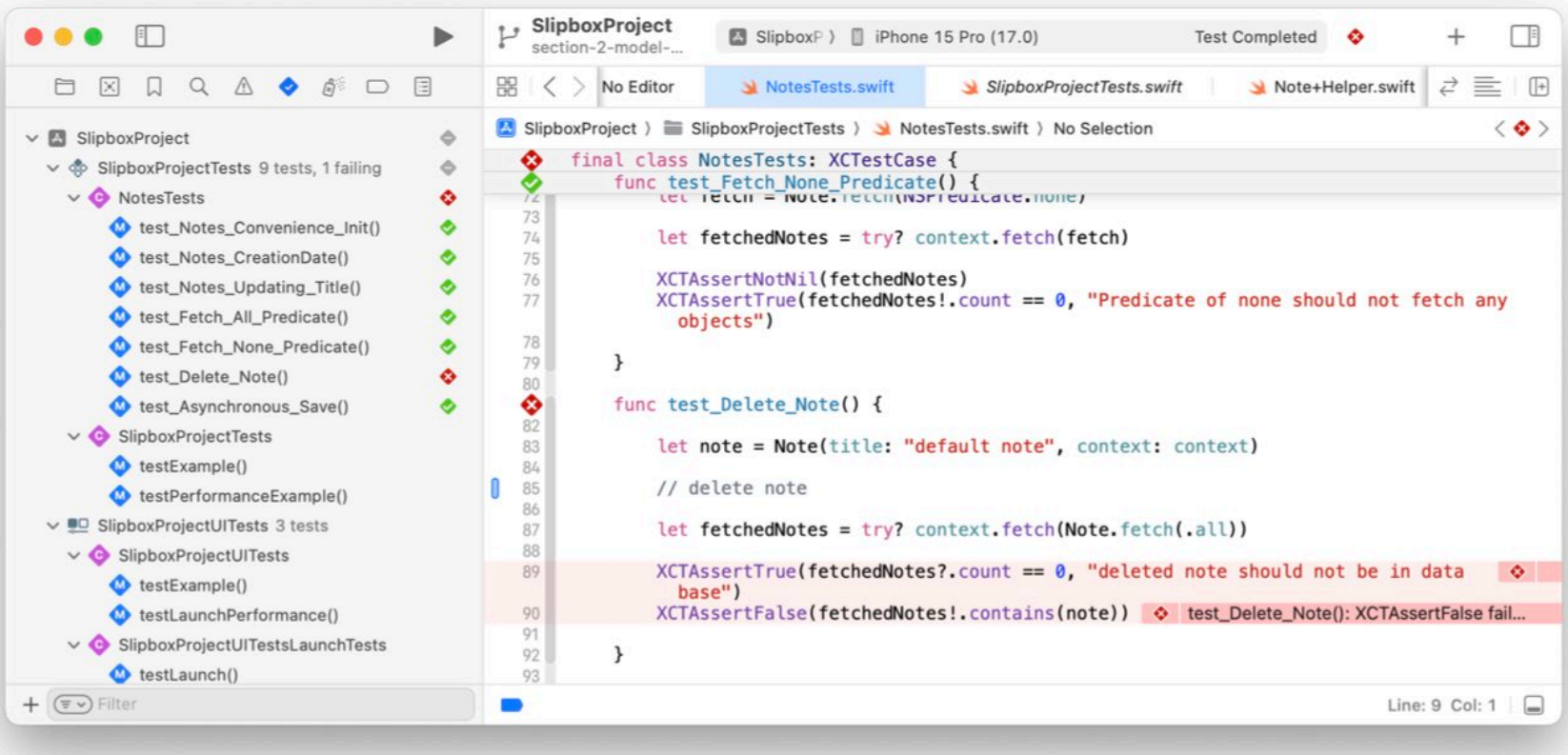
Step 1: Write a Test That Fails

I'll demonstrate by writing a test for the delete functionality. The test function is named testDeleteNote. To delete a note, I first need to create one, because to delete something, it has to exist. So here's what I do:

```
    func test_Delete_Note() {
        let note = Note(title: "default note", context: context)

        // delete note here

        let fetchedNotes = try? context.fetch(Note.fetch(.all))

        XCTAssertTrue(fetchedNotes?.count == 0,
                      "deleted note should not be in database")
        XCTAssertFalse(fetchedNotes!.contains(note))
    }
```

Notice that I use XCTAssertEqual to check if the note count is zero after deletion, and XCTAssertFalse to confirm the deleted note is not contained in the fetched results.

Run the test now. It should fail. That's expected in TDD. Next, you will work on implementing the delete action and passing this test.



Step 2: Create a Delete Function

I am going to implement the delete function in the extension of the Note class:

```
import Foundation
import CoreData

extension Note {

     …

    static func delete(note: Note) {
        guard let context = note.managedObjectContext else { return }
        context.delete(note)
    }
}
```

Step 3: Passing the Test

Now that the delete function is implemented, I can use it in the test function:

```
func test_Delete_Note() {
    let note = Note(title: "default note", context: context)

    Note.delete(note: note)

    let fetchedNotes = try? context.fetch(Note.fetch(.all))

    XCTAssertTrue(fetchedNotes?.count == 0,
                  "deleted note should not be in database")
    XCTAssertFalse(fetchedNotes!.contains(note))
}
```

I run the test again. This time, it should pass, confirming that the delete operation works as expected. If it passes, I know that I'm not retrieving any notes from the database, and the specific note I intended to delete is indeed gone.

Step 4: Refactor (if necessary)

The final step in TDD is to refactor the code if needed. But before refactoring, I ensure that I have a failing test for the current implementation. Then, after making changes, I write the necessary code to make the test pass again.

Remember, every time you write a new test, you need to check for s**ituations where it passes and fails**. Practicing with variations and edge cases ensures that your code behaves as expected without interference from other factors.

By following these steps, you can practice TDD and ensure that your Core Data operations are reliable and bug-free. Keep experimenting with different scenarios to strengthen your understanding of TDD and Core Data within your SwiftUI applications.

2.5 HOW TO WRITE UNIT TESTS FOR ASYNCHRONOUS CODE

When you're building an app with Core Data and SwiftUI, you'll often perform asynchronous operations. These operations are executed in the background, which means they don't block your main UI thread and will be completed at a later time. Testing asynchronous code requires a different approach than testing synchronous code, and I'm going to show you how to handle it.

Understanding Asynchronous Operations

Let's take a common asynchronous operation: saving data to Core Data. You typically call context.save() to persist changes to your managed object context. This task is executed in the background because you don't want to block the main UI. It's okay if it takes a bit longer; you don't really mind as long as it doesn't interfere with the user experience.

```
struct PersistenceController {
    …

    func save() {
        let context = container.viewContext
        guard context.hasChanges else { return }

        do {
            try context.save()
        } catch {
            let nsError = error as NSError
            fatalError("Unresolved error \(nsError), \(nsError.userInfo)")
        }
    }
}
```

Writing a Unit Test for Asynchronous Save

Testing an asynchronous save operation involves checking whether the save actually happens. Since you don't get a direct result back from the save method, you need to listen for a notification that indicates the operation is complete.

To do this in a unit test, you use XCTestExpectation. Expectations tell your test what to wait for. In this case, you're waiting for a notification that indicates the managed object context has been saved.

Here's how you can write an expectation for an asynchronous save operation:

```
    func test_Asynchronous_Save() {

     expectation(forNotification: .NSManagedObjectContextDidSave, object: context) { _ in
            return true
       }

        let note = Note(title: "default note", context: context)
        controller.save()

        waitForExpectations(timeout: 2.0) { error in
            XCTAssertNil(error, "saving did not complete")
        }
    }
```

In this example, you create an expectation and then add an observer for the NSManagedObjectContextDidSave notification. When the context saves successfully, the notification is posted, and your expectation is fulfilled.

The waitForExpectations(timeout:handler:) method is crucial. It tells the test to wait for a specified duration for the expectation to be fulfilled. If the timeout is reached without the expectation being fulfilled, the test fails.

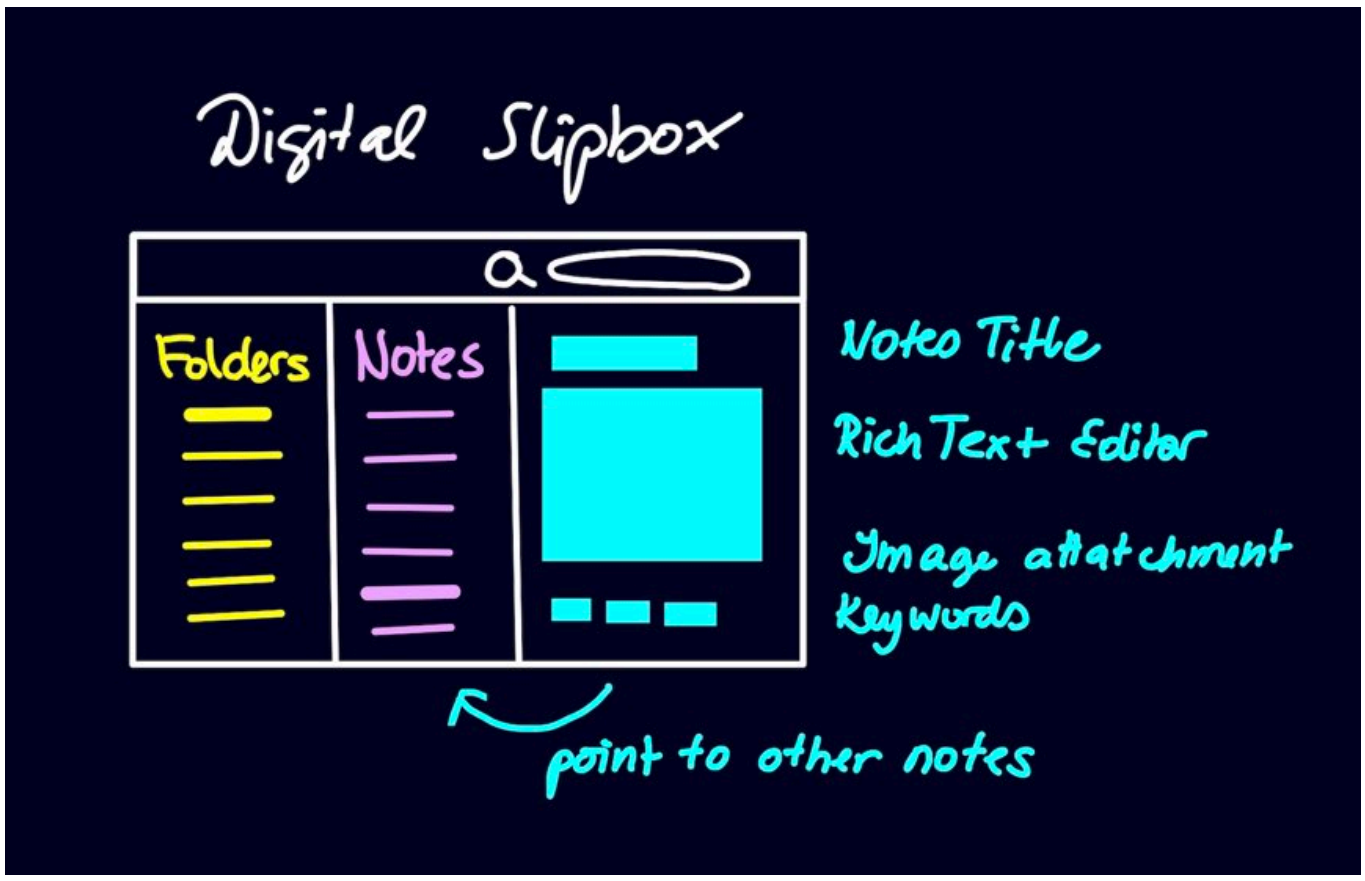
Testing asynchronous code can be more complex than testing synchronous code, but it's essential, especially for operations like database performance, data migrations, or schema changes. By using XCTestExpectation and being mindful of both positive and negative test cases, you can ensure that your asynchronous Core Data operations behave as expected. Remember to test thoroughly and handle all possible scenarios to maintain a stable and reliable app.

3. SCHEMA ATTRIBUTES

In this chapter, we delve into the intricacies of the Core Data schema. You'll learn how to effectively map your app's information to Core Data, ensuring that your data model is robust and well-structured.

3.1 SCHEMA FOR NOTES

When working on your model schema, it is best to look at the app first and determine what properties we need:



In the right column, we will show the note detail view. First, we will show the notes title property, then the content of the note in a rich text editor which should be of type NSAttributedString. We will show a list of image attachments. In the middle column, we will show the notes in an ordered list.

Let's summarise and list all **attributes that the Note entity should have:**

- **title**: String
- **creationDate**: Date
- bodyText: NSAttributedString
- Image: Data
- isFavorite: Bool
- status: Enum with draft, review, and archived cases

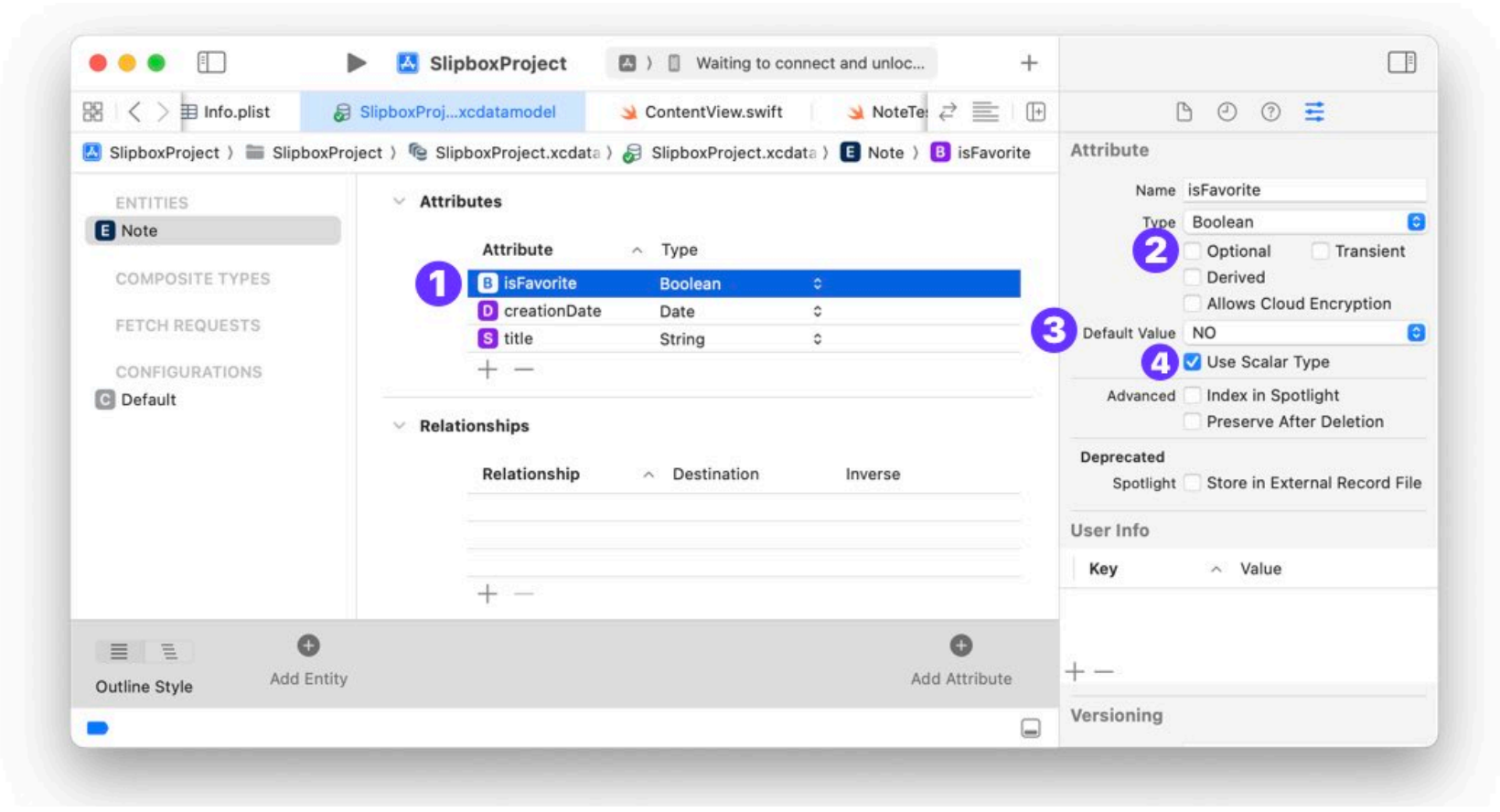
In section 1, we already added the attributes for title and creationDate. In this section, we will work on the rest including how to store enum, boolean, and images.

Additionally, we will create another entity Tag, which we will later use to tag the notes objects. We are going to store a custom type Color for each Tag.

In the following, I will give you examples of primitive types like Boolean, String, and Numbers. In the following section, you will learn how to save more complex data like enums and images.

Storing Boolean Values

For features like marking a note as a favorite, we use a Boolean value. Here's how you define a **Boolean attribute (1)** in your Core Data model:



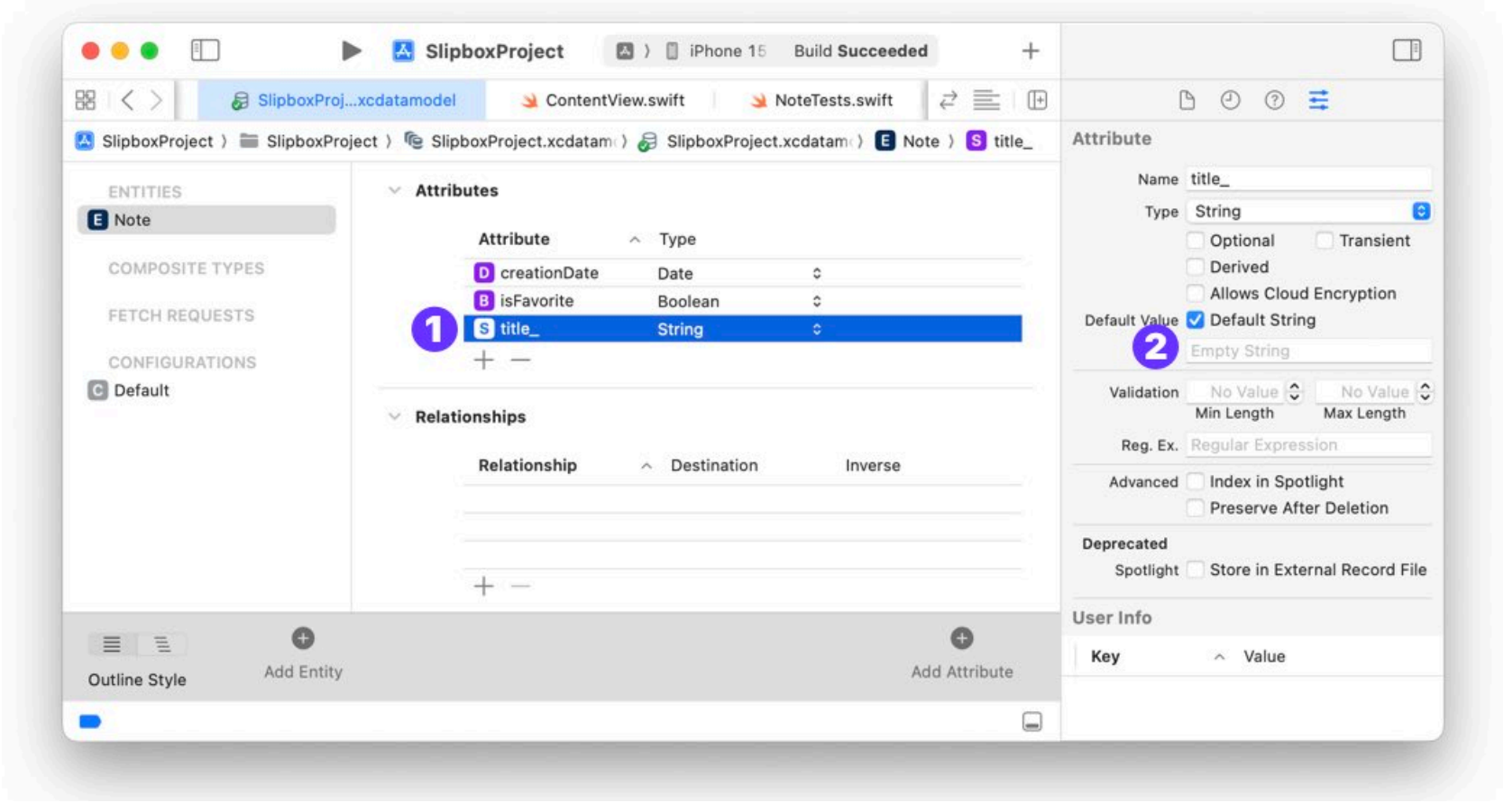
We can set this attribute to **non optional (2)** and set a **default value (3)** of "NO". Make sure the **"Use Scalar Type" (4) toggle** is enabled. Like this the attribute is non-optional. When you create a new Note object, it is per default set as false.

You can write a test to check how the isFavorite property is set:

```
func test_IsFavorite_Default_Value() {
    let note = Note(title: "default note", context: context)

    XCTAssertFalse(note.isFavorite, "note is per default not favorite")
}
```

Storing String Values and Handling Optionals

The option for **“use scalar type” (2)** is not available for String types. Which means CoreData will handle String as optional. To work around this I introduced to you the wrapper title_ property.



Together with the computed property “title” in the note extension:

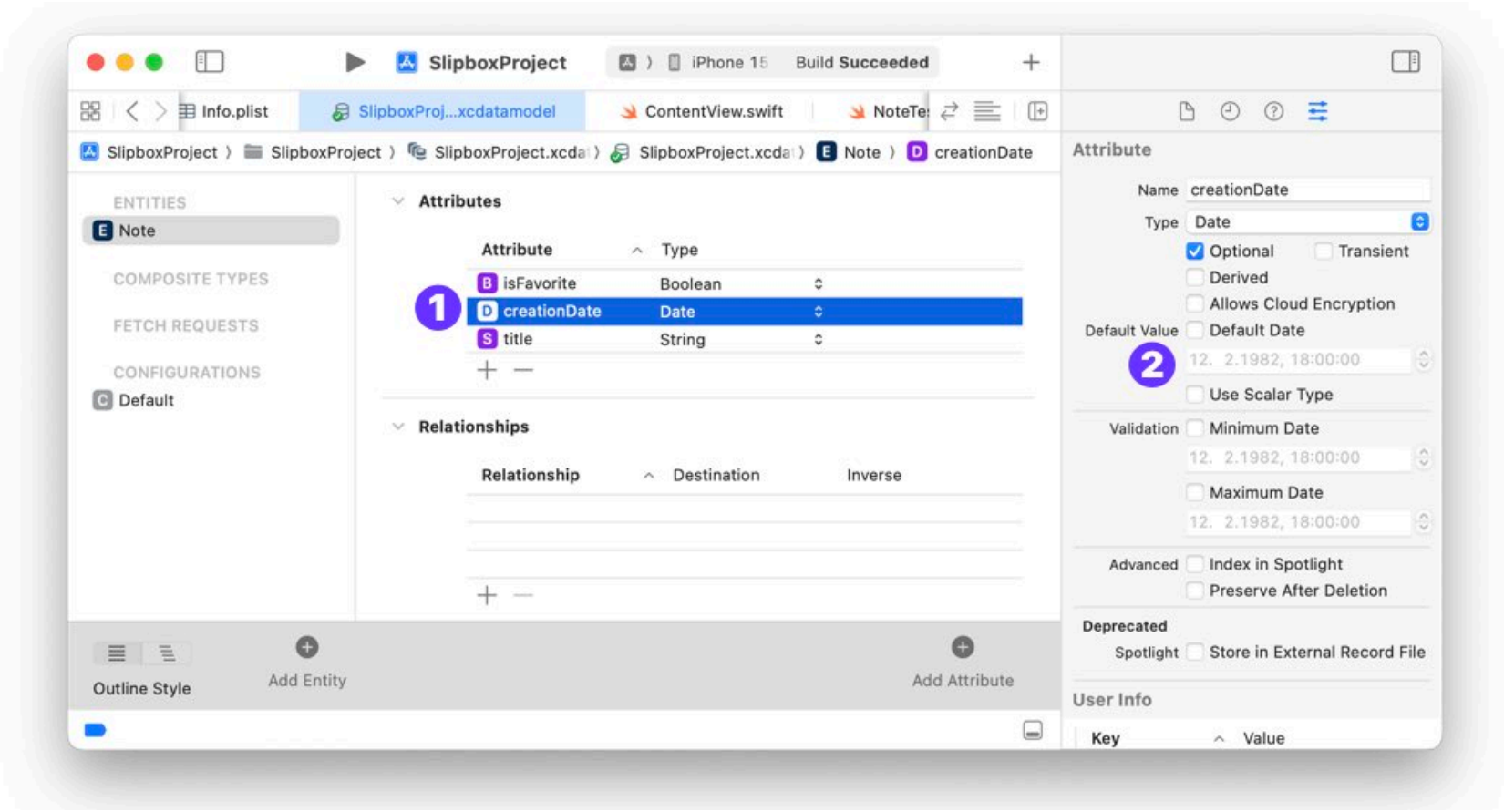
```
extension Note {
    …

    var title: String {
        get { self.title_ ?? "" }
        set { self.title_ = newValue  }
    }
}
```

This makes it possible to use the non-optional property title in SwiftUI. For example, in NoteDetailView I add a TextField to change the title of the note:

```
struct NoteDetailView: View {

   @ObservedObject var note: Note

   var body: some View {
        TextField("title", text: $note.title)
              .textFieldStyle(.roundedBorder)
    }
}
```

Storing Date Values

I already added a **(1) “creation date”** attribute of type Date. Every time I am creating a new Note instance, I need to set this property to the current time. You could use the **attribute inspector (2)** to set a default value. However you can only set one predefined value. For the creationDate property this is different for each Note instance. So using the attribute inspector in the xcdatamodeld file is not an option.



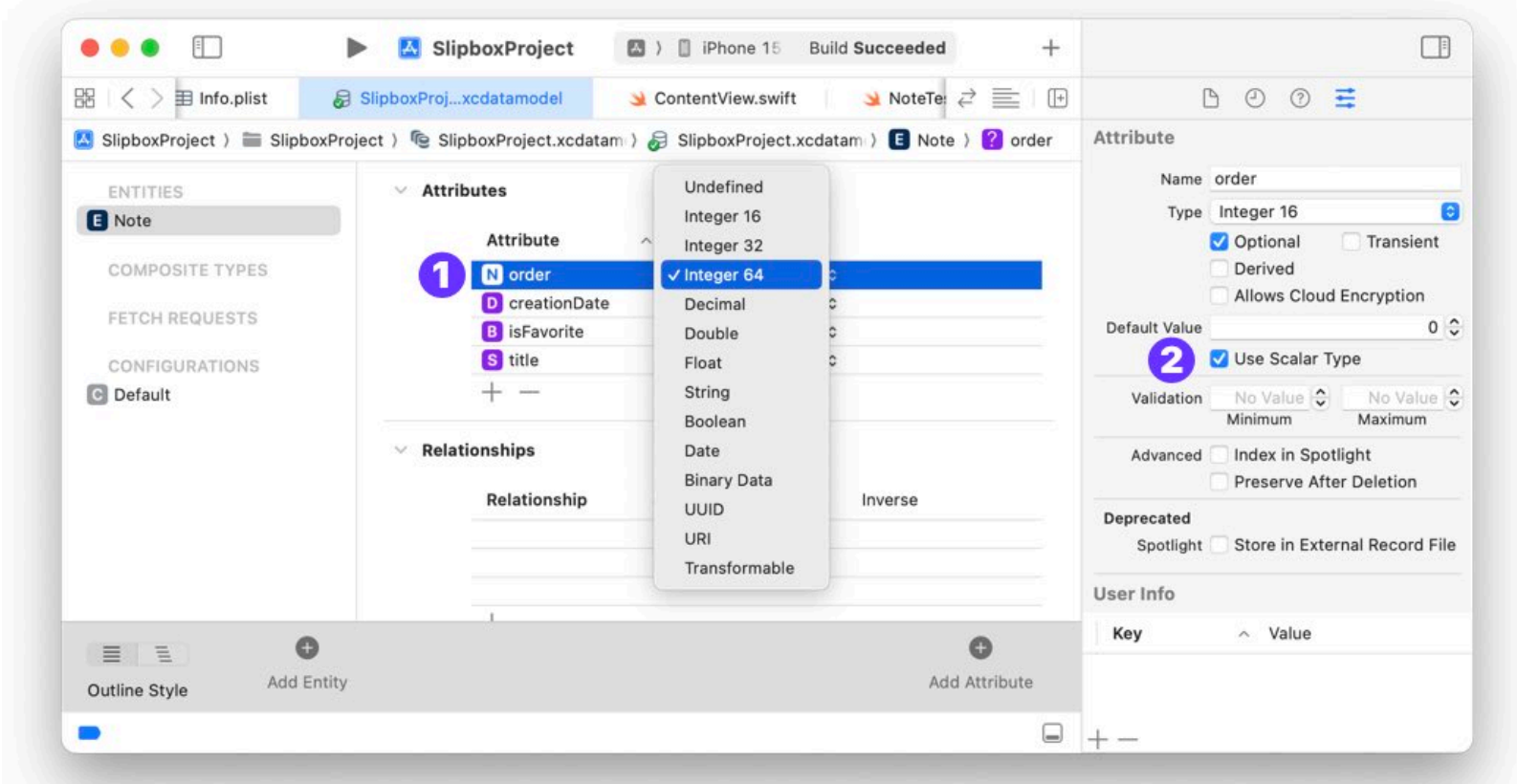
Instead, I am setting the creationDate property in the code. Each CoreData entity gets default initializers. I already added a convenience initializer. I don’t want to set this property in the convenience initializer because I might also use the default initializers somewhere and for these, the creationDate attribute would not be set. For this situation, you can use the **awakeFromInsert** function that is called for all objects when they are created. No matter with which initializer. Here is how you can set the creationDate attribute with awakeFromInsert:

```
extension Note {
    …

    public override func awakeFromInsert() {
        super.awakeFromInsert()
        self.creationDate = Date()
    }
}
```

This is a very useful function. You can use it to set other properties programmatically, e.g. uuids.

Storing Number Values

For attributes like the order of notes in a list, you can use integers that describe the sort order. Here's an example of how you define an integer attribute:



You can see options for Int16, Int32, and Int64. They correspond to integer data types in Swift, and you can choose them based on the range of values you need to store.

Int16

- **Range**: -32,768 to 32,767
- **Use Case**: You use Int16 when you want to store integer values that fall within the small range specified. It's a good choice when you know you will not exceed these limits, as it saves storage space compared to larger integer types.

Int64

- **Range**: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- **Use Case**: You choose Int64 when you need to store very large integer values. It's suitable for cases where Int16 or Int32 wouldn't provide a sufficient range, like identifiers that might exceed billions in value or when you are dealing with large counts or calculations.

When you're defining your Core Data model in the Xcode model editor, you can select these types from the attribute type dropdown menu for each attribute you create.

You can make number attributes non-optional and set a **default value (2**). In the attribute inspector set a default value e.g. “0”. Toggle the “Use Scalar Type”.

Note that you don’t have an option for “Use Scalar Type” or similar for String types. Therefore String attributes are optional in Swift.

Mapping Complex Data to Store in CoreData

For the sort order, an integer is sufficient. But you can also choose **Double or Float** types. For example, in a drawing app where you need to save the size and position of the shape. You could store the width of the shape as a Float type:

```
extension CDSape {
    @NSManaged public var positionX: Float
    @NSManaged public var positionY: Float
    @NSManaged public var width: Float
    @NSManaged public var height: Float
}
```

In your SwiftUI views you could then convert this value from Float to CGFloat and set it with the frame(width) modifier:

```
shape.drawing
   .frame(width: CGFloat(shape.width),
          height: CGFloat(shape.height))
   .position(x: CGFloat(shape.positionX),
             y: CGFloat(shape.positionY))
```

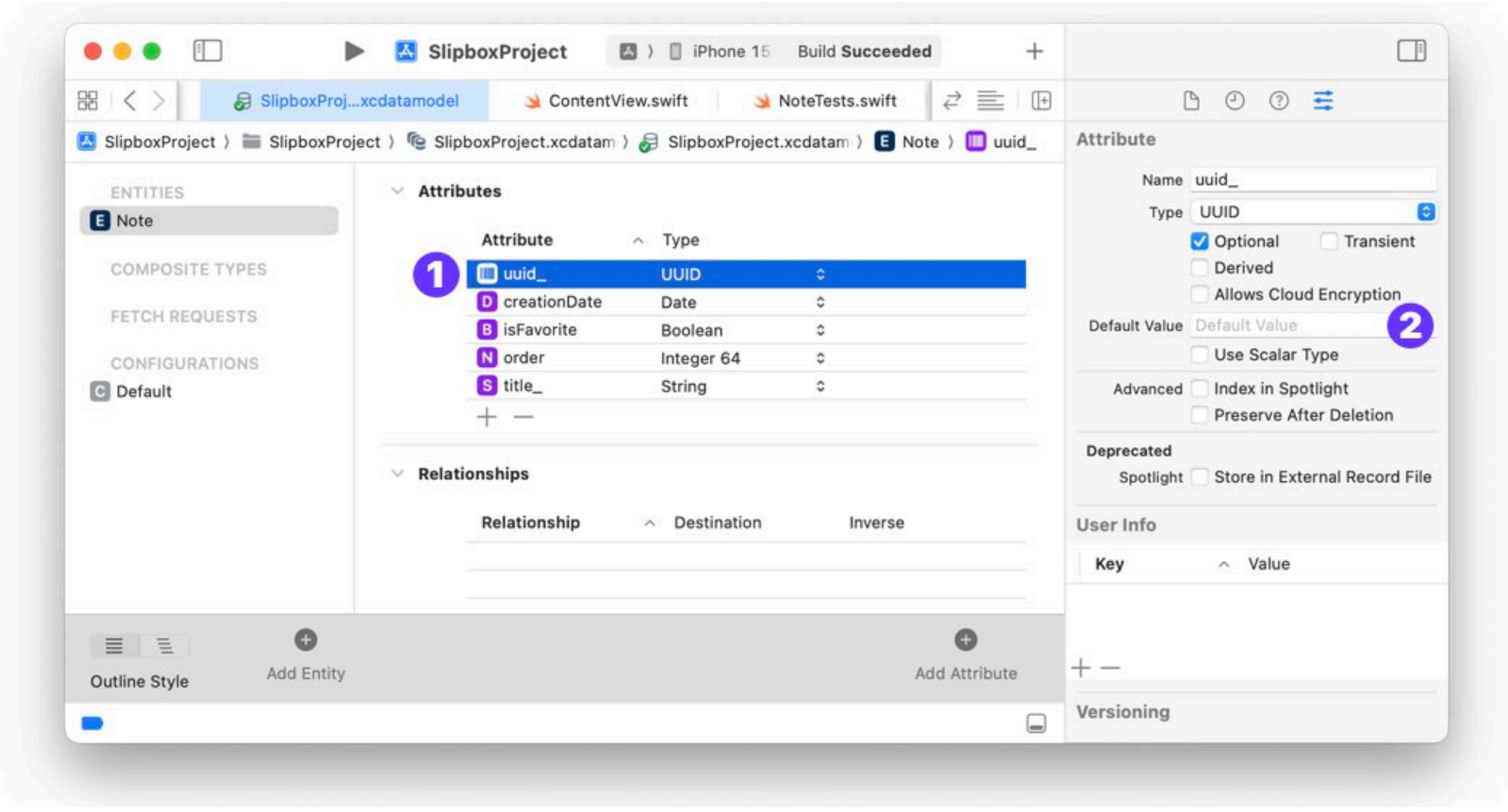
As you see, I broke down the complex information of size and position into atomic pieces of positionX, positionY, width, and height. I store these individual pieces in CoreData as individual attributes. When I retrieve them to use in SwiftUI, I get all these properties and piece them back together. You can make this even more convenient by adding computed properties. For example, I create a property that composes the size of the shape:

```
extension CDSape {
    var size: CGSize {
        CGSize(width: CGFloat(width), height: CGFloat(height))
    }
}
```

This principle of **breaking down complex data into small bit-sized pieces** that you can store with primitive type in CoreData is a very common and useful pattern. You will see it used a lot when mapping your data in code.

Storing Unique Identifiers of Type UUID

Core Data generates identifiers for all Entities of type ObjectIdentifier. I oftentimes find them difficult or near impossible to work with. A more convenience type is UUID. In addition to the already existing id, I am adding a “uuid" property of type UUID:



For UUID type attributes you can set a default value and make them non-optional by enabling **“Use Scalar Type” (2).** However, I need to set a different identifier for each instance. If I set an uuid in the attribute inspector all instances will have the same identifier. This is definitely not what I want. Instead, I use the same approach as for the “creationDate” and use the “awakeForInsert “function:

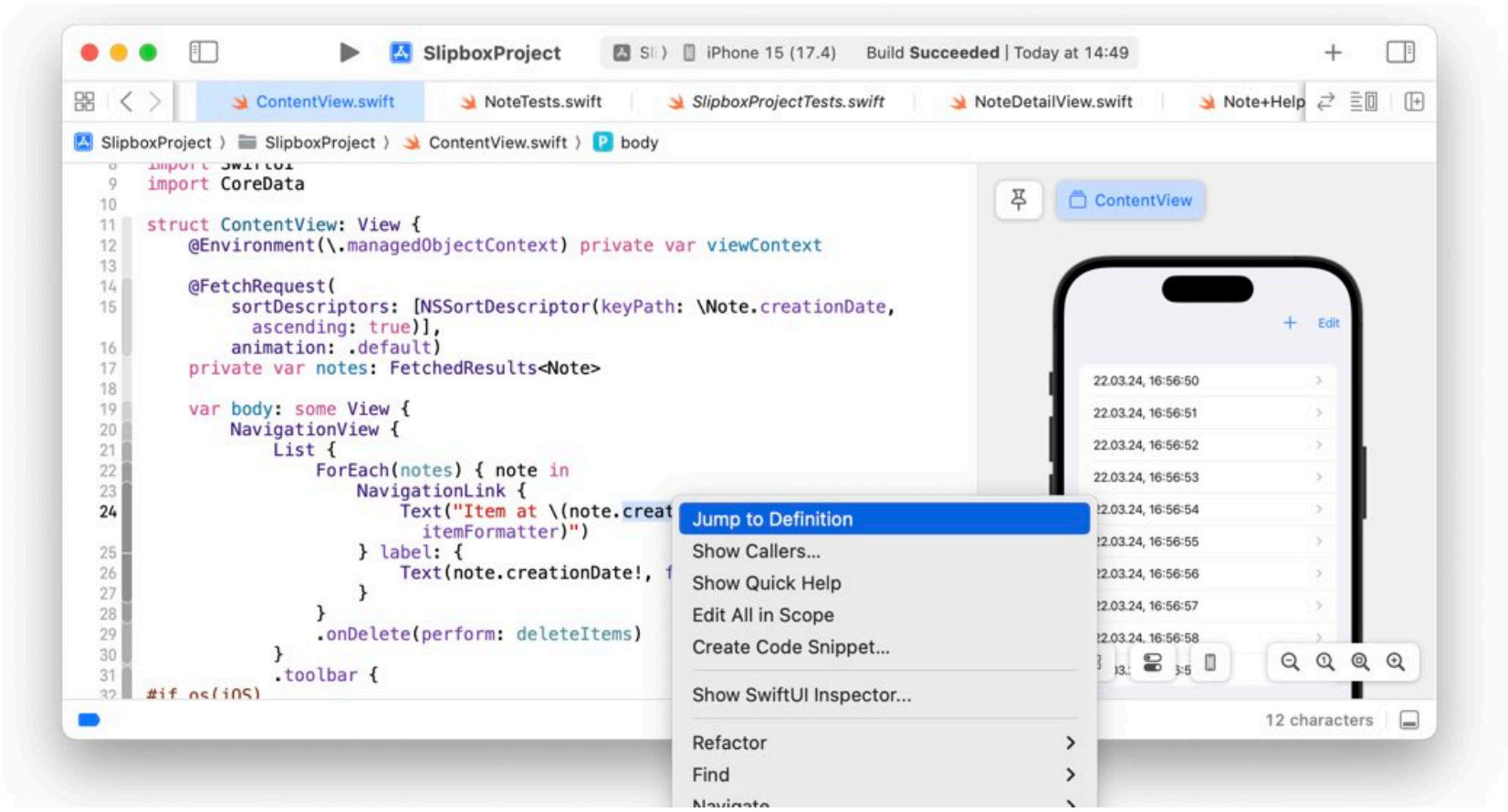
```
extension Note {

    …

    var uuid: UUID {
        get { uuid_ ?? UUID() }
    }


    public override func awakeFromInsert() {
        super.awakeFromInsert()
        self.creationDate = Date()
        self.uuid_ = UUID()
    }
}
```

Since I don’t want to deal with the optional handling, I use again a wrapper type “uuid_” and handle the optional in a computed property “uuid”.

Checking your Schema in Code

When you work with the “xcdatamodeld” file, you might not be sure if you set all properties correctly. You can do an additional check. Go to “ContentView”. Right click on an attribute of Note and choose “Jump to definition”.

Xcode will open a autogenerated file, that is normally hidden from you.



Here is an example of what you might see:

```
import Foundation
import CoreData

extension Note {
    @nonobjc public class func fetchRequest() -> NSFetchRequest<Note> {
        return NSFetchRequest<Note>(entityName: "Note")
    }

    @NSManaged public var uuid_: UUID?
    @NSManaged public var creationDate: Date?
    @NSManaged public var isFavorite: Bool
    @NSManaged public var order: Int64
    @NSManaged public var title_: String?
}

extension Note : Identifiable {
}
```

I have the 5 properties that I defined in the “xcdatamodeld” file. This code automatically updates when you change the schema. It is protected and cannot be changed by you.

As you can see 3 properties are optional. I set “isFavorite” and “order” as non-optionals in Swift.

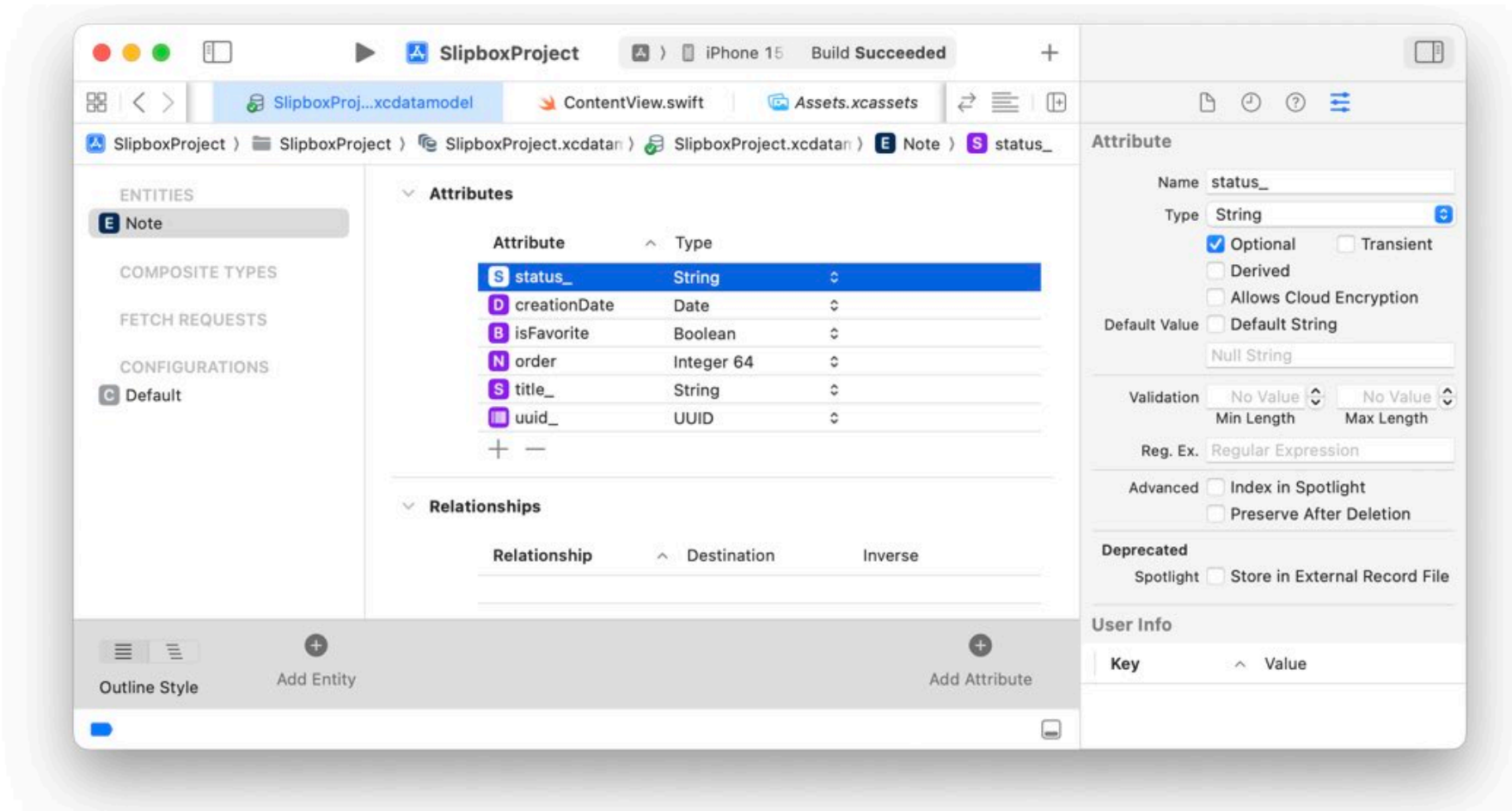
3.2 HOW TO SAVE ENUM IN CORE DATA

When working with Core Data and enums in Swift, you might wonder how to store enum values persistently. I'll walk you through the process step by step.

First, let's define an enum. Suppose we have a Status enum representing the status of a note with three cases: .draft, .review, and .archive. Here's how you might declare it in a Swift file within your models' group in Xcode:

```
enum Status: String, Identifiable, CaseIterable {

    case draft = "Draft"
    case review = "Review"
    case archived = "Archived"

    var id: Self { return self }
}
```

Notice that I've made the enum conform to Identifiable and CaseIterable. The Identifiable protocol requires an id property. The rawValue of "Status" is String, which I will use to store it in CoreData. Add a "status_" attribute to the "Note" entity and make it of type String:



I am using the primitive String type to create a representation for the enum. Because it is of type String, I can easily store it in CoreData.

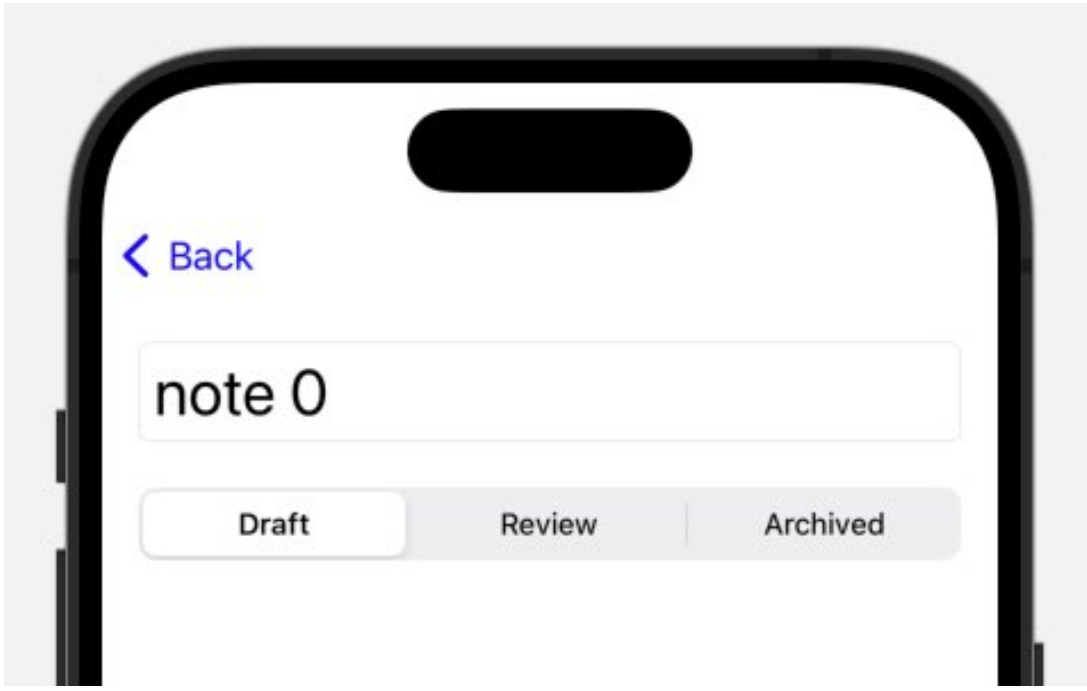
For my SwiftUI views, I want to have a convenience way of getting a status property with my custom enum type. In the Note extension, I am adding another computed property that handles the conversion from primitive CoreData type String to my custom enum type:

```
extension Note {
    …

    var status: Status {
        get {
            if let rawStatus = status_,
              let status = Status(rawValue: rawStatus){
                return status
            } else {
                return Status.draft
            }
        }
        set {
            status_ = newValue.rawValue
        }
    }
}
```

The setter takes the new Status enum and uses it to set the underlying database attribute “status_”. In the getter, I first to the optional handling. If “status_” is nil, I return the “draft” status. If the “status_” attribute has a value, I use it to generate an enum Status type.

This looks a little bit clunky but makes my SwiftUI code simpler. For all SwiftUI views, I will use the computed property “status”. The underbar property “status_” is the one in the database. This naming convention with an underbar, helps me distinguish and remember which property I should use where.

Let’s use the enum attribute in the NoteDetailView. I am showing a picker with a segmented style:



```swift
struct NoteDetailView: View {

    @ObservedObject var note: Note

    var body: some View {
        VStack(spacing: 20) {
            TextField("title", text: $note.title)
                .textFieldStyle(.roundedBorder)
                .font(.title)

            Picker(selection: $note.status) {
                ForEach(Status.allCases) { status in
                    Text(status.rawValue)
                }
            } label: {
                Text("Note´s status")
            }
            .pickerStyle(.segmented)
        }
        .padding()
    }
}
```

If you see only the SwiftUI view, you see a nice Swift enum type. The additional work of converting the enum to a primitive CoreData type is hidden in the Note extension. With this, you can easily work with enums in SwiftUI.

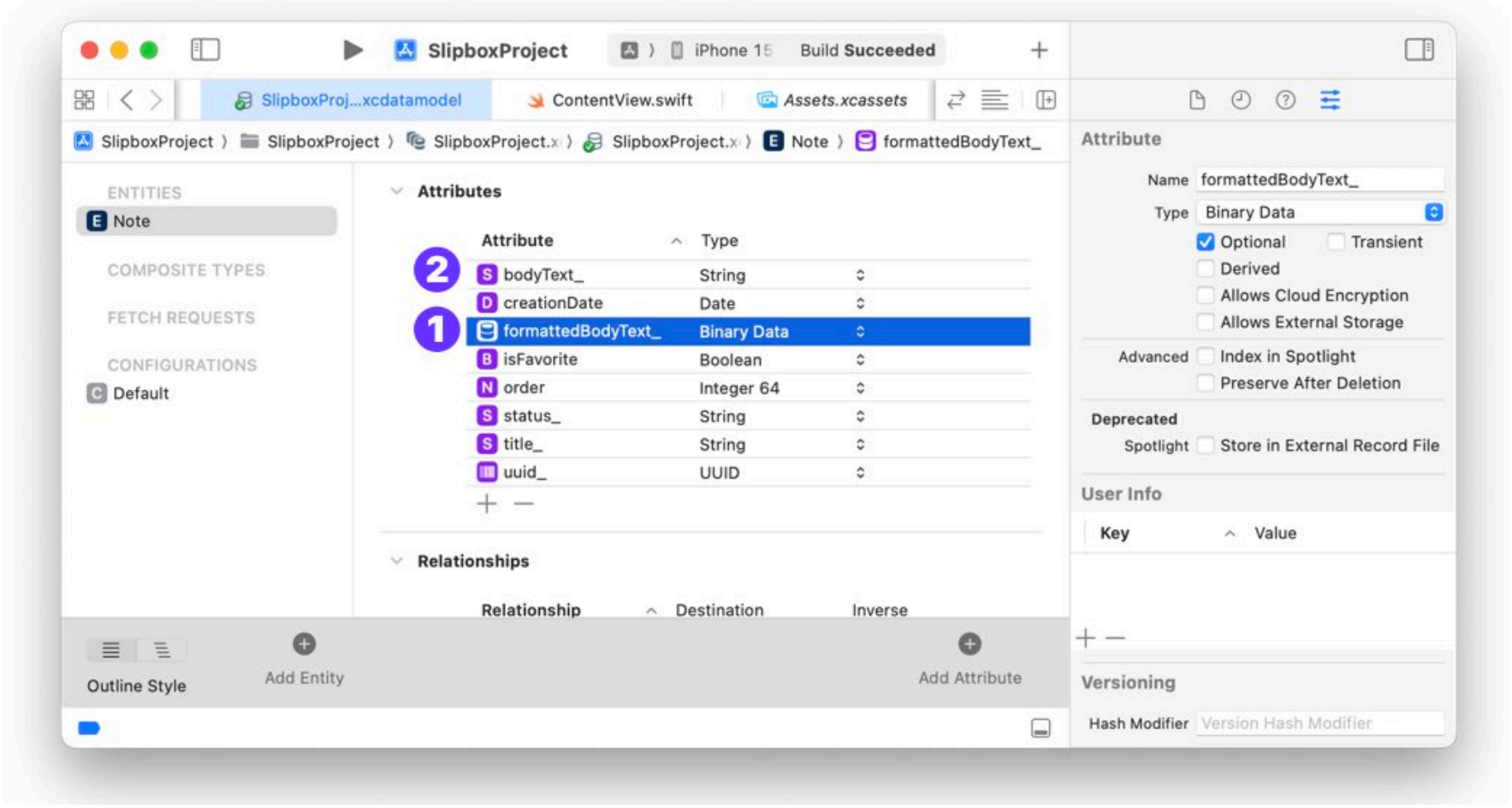
To summarize, **storing an enum in Core Data involves:**

1. Defining the enum with a raw value type that Core Data can store (such as String or Int).
2. Create an attribute in “xcdatamodeld" file with the enum raw type (String)
3. Creating a computed property on your Core Data model object that translates between the stored value and the enum.
4. Using the computed property to bind the enum to your SwiftUI views.

Remember, this approach works well for simple enums. If your enum has associated values or is more complex, you’ll need a more advanced strategy to store and retrieve the values in Core Data.

3.3 RICH TEXT EDITOR AND SAVING NSATTRIBUTEDSTRING IN CORE DATA

Now, let's say you want to add rich text editing capabilities to your notes. This involves storing attributed strings, which are not a standard attribute type in Core Data. You'll need to use a binary data type for this:



Create a new attribute named "formattedBodyText_" and set its type to Binary Data. In section 4, I will show you how to search notes with a search text field. If you use Binary Data types, they are not searchable with NSPredicate, which only works with primitive types.

As a workaround, I am adding an additional attribute "bodyText_" of type String. This attribute represents the "plain" information of "formattedBodyText_". We will use it to implement the search feature later.

With this 2 attribute approach, you need to make sure both attributes are in sync.

Converting Data to NSAttributedString

Next, I am adding convenience code that makes it easier to convert the CoreData "Binary Data" to NSAttributedString, which we will use for the rich text editor in SwiftUI.

Add a new file **"NSAttributedString+Data+Helper.swift**" and past the following:

```
import Foundation

extension NSAttributedString {

    func toData() -> Data? {
        let options: [NSAttributedString.DocumentAttributeKey: Any] =
```

```
                [.documentType: NSAttributedString.DocumentType.rtf,
                .characterEncoding: String.Encoding.utf8]

            let range = NSRange(location: 0, length: length)
            let result = try? data(from: range, documentAttributes: options)
            return result
        }

    }

    extension Data {
        func toAttributedString() -> NSAttributedString? {

            let options: [NSAttributedString.DocumentReadingOptionKey: Any] =
                [.documentType: NSAttributedString.DocumentType.rtf,
                  .characterEncoding: String.Encoding.utf8]

            let result = try? NSAttributedString(data: self,
                                                 options: options,
                                                 documentAttributes: nil)
            return result
        }
    }
```

The **toData** method converts the NSAttributedString to Data using the RTF document type, which is a rich text format. The **toAttributedString** static method does the reverse, creating an NSAttributedString from Data.

Integrating with Core Data

With these extensions in place, you can easily integrate with Core Data by storing the Data representation of your attributed string. Here's how you might use these methods in your Core Data model:

```
    extension Note {
        …

        var formattedBodyText: NSAttributedString {
            get {
               formattedBodyText_?.toAttributedString() ?? NSAttributedString(string: "")
            }
            set {
                formattedBodyText_ =  newValue.toData()
                bodyText_ = newValue.string.lowercased() // storing the plain text
            }
        }

        var bodyText: String {
            get { bodyText_ ?? "" }
        }
    }
```

I created a "formattedBodyText" computed property that does the data conversion. In the getter, I use the "toAttributedString" function to convert from "Data" to "NSAttributedString". In the setter, I use the

“toData” function to take the new value of type “NSAttributedString”, convert it to “Data” type and store it in CoreData in the “formattedBodyText_” attribute.

Additionally, I use the setter to create a string version from the new rich text value “NSAttributedString” and set it to “bodyText_”. This generates a “plain” copy that is always in sync with “formattedBodyText_”.

Showing a Rich Text Editor

In SwiftUI, we don’t have a native rich text editor component. To integrate a rich text editor into a SwiftUI app, you need to bridge UIKit and AppKit components using UIViewRepresentable and NSViewRepresentable protocols. I’ll guide you through the process of creating wrappers for UITextView on iOS and NSTextView on macOS.

Creating the iOS Rich Text Editor with UIViewRepresentable

Firstly, you need to define a SwiftUI view that conforms to UIViewRepresentable. This protocol requires you to implement two methods: makeUIView(context:) and updateUIView(_:context:).

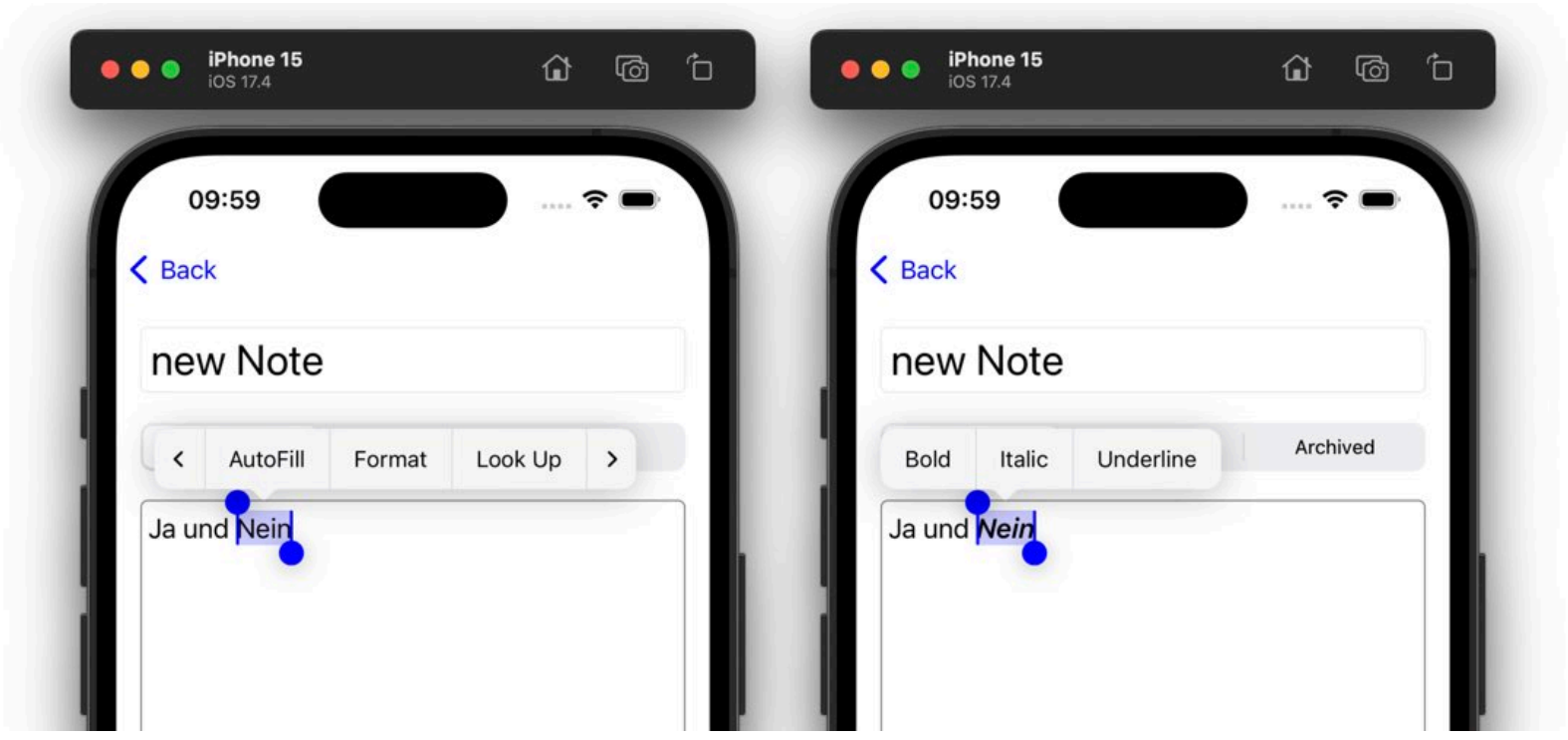
```
    import SwiftUI

    struct TextViewIOSWrapper: UIViewRepresentable {

        let note: Note

        func makeCoordinator() -> Coordinator {
            Coordinator(self, note: note)
        }

        func makeUIView(context: Context) -> UITextView {
            let view = UITextView()
            // Additional configuration goes here

            view.delegate = context.coordinator
            return view
        }

        func updateUIView(_ uiView: UITextView, context: Context) {
            uiView.textStorage.setAttributedString(note.formattedBodyText)
            context.coordinator.note = note
        }

        class Coordinator: NSObject, UITextViewDelegate {

            var note: Note
            let parent: TextViewIOSWrapper

            init(_ parent: TextViewIOSWrapper, note: Note) {
                self.parent = parent
                self.note = note
            }

            func textViewDidChange(_ textView: UITextView) {
                note.formattedBodyText = textView.attributedText
            }
        }
    }
```

In **makeUIView(context:)**, you create and configure the UITextView. The **updateUIView(_:context:)** method is where you update the view with new data. The Coordinator class acts as a delegate for the UITextView to handle text changes. When the text changes we update the “formattedBodyText” property of note and thus store the user input in CoreData.

UITextView has a lot of customisation available. You can change **updateUIView** to set the **font**, **view.allowsEditingTextAttributes**, **border** and **cornerRadius**:

```
    func makeUIView(context: Context) -> UITextView {
        let view = UITextView()

        view.allowsEditingTextAttributes = true
        view.isEditable = true
        view.isSelectable = true
        view.font = UIFont.systemFont(ofSize: 18)

        view.layer.borderWidth = 1
        view.layer.borderColor = UIColor.gray.cgColor
        view.layer.cornerRadius = 5

        view.textStorage.setAttributedString(note.formattedBodyText)
        view.delegate = context.coordinator

        return view
    }
```

Open “NoteDetailView.swift” file and add the new component. Build and run the iOS simulator. You can select the text, choose “Format” and change the text style to “Bold”, “Italic” or “Underline”:




```
struct NoteDetailView: View {

    @ObservedObject var note: Note

    var body: some View {
        VStack(spacing: 20) {
            …
```

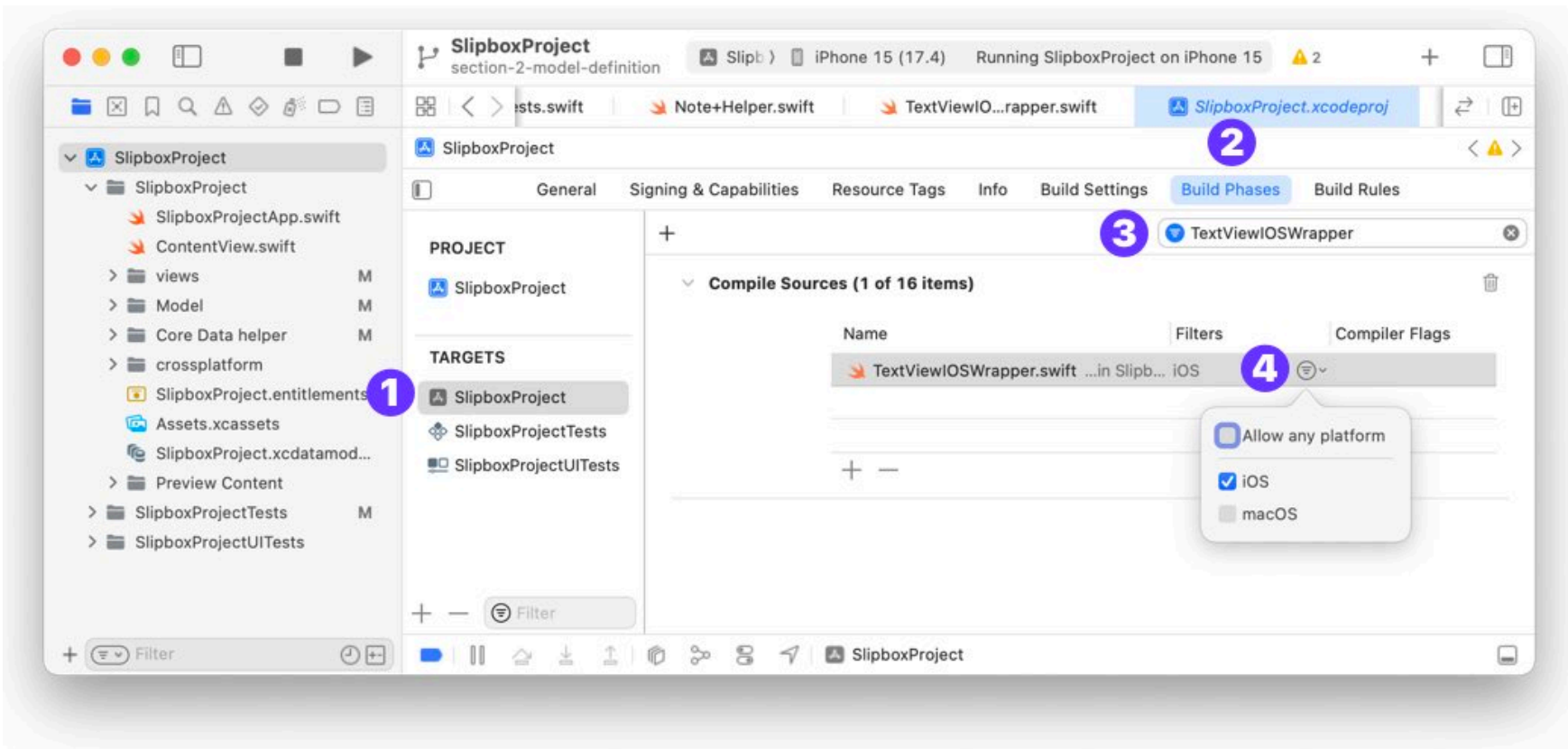
```
                Picker(selection: $note.status) {
                    …
                }
                .pickerStyle(.segmented)

                TextViewIOSWrapper(note: note)
            }
            .padding()
        }
    }
```

Cross-platform and Rich Text Editor for macOS

For macOS, you'll use NSViewRepresentable to wrap an NSTextView. The process is similar to UIViewRepresentable. Create a new SwiftUI file **"TextViewMacosWrapper".**

If you would now switch to build for macOS, Xcode would through an error for "TextViewIOSWrapper", which is using "UIViewRepresentable". Therefore we need to conditionally use these files for cross-platform.

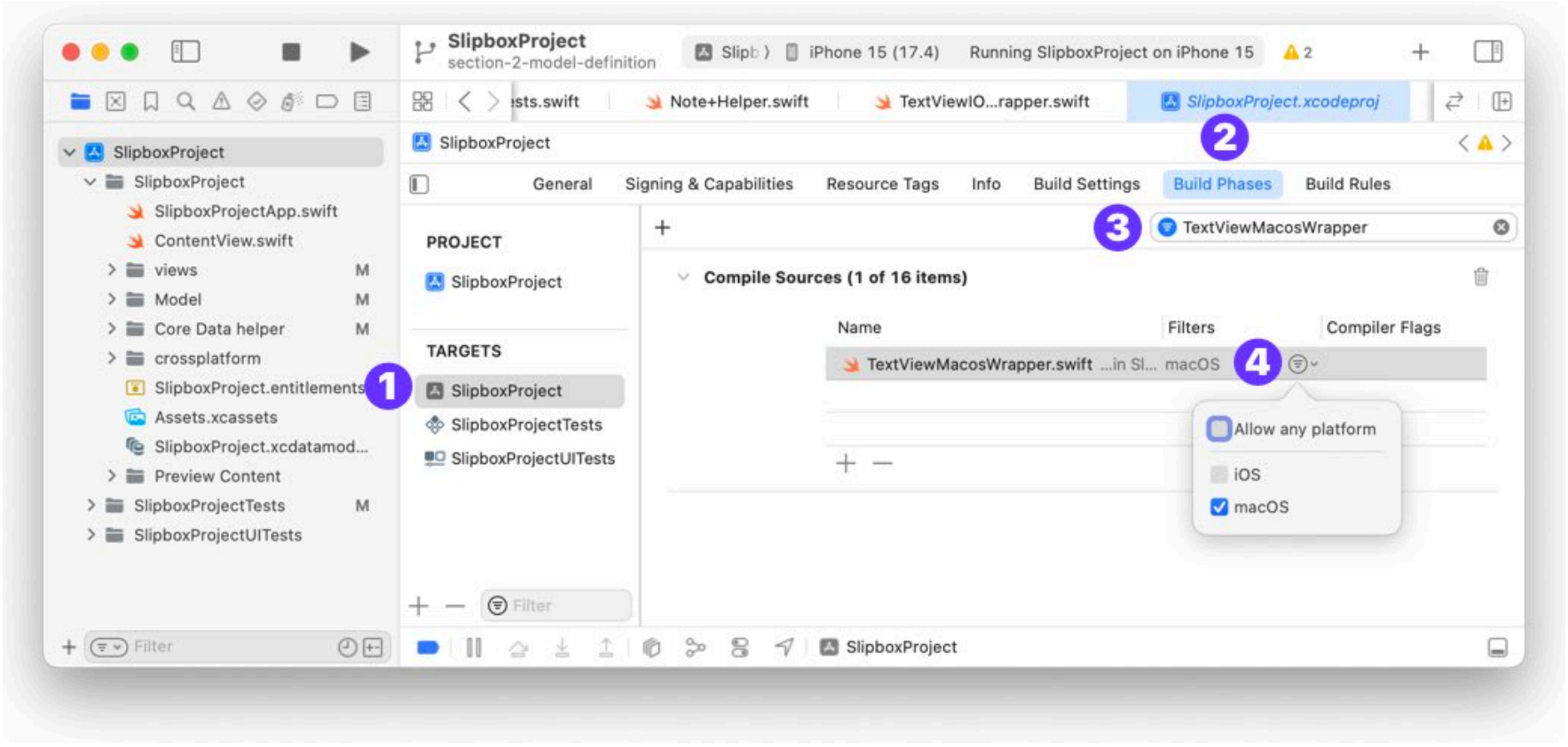
Go to your project's **target settings (1)**. Select the **"Build Phase" (2)** section. In the **search text field (3)** search for your file name "**TextViewIOSWrapper**":



You should see on result for "TextViewIOSWrapper". Change the **"Filters" (4)** from "all". Only enable to option for iOS.

"TextViewIOSWrapper.swift" file will now only be used for iOS.

Do the inverse for the macOS view. Search for “**TextViewMacosWrapper” (3):**



Change the **filter (4)** to only use this for macOS.

Now that you have succefully created two versions for the rich text editor. Let’s finish and add the macOS editor. Open “**TextViewMacosWrapper.swift**” and add the “NSViewRepresentable” protocol conformance:

```
struct TextViewMacosWrapper: NSViewRepresentable {

    let note: Note

    func makeCoordinator() -> Coordinator {
        Coordinator(note: note, parent: self)
    }

    func makeNSView(context: Context) -> NSTextView {
        let view = NSTextView()
        // Additional configuration goes here

        view.delegate = context.coordinator
        return view
    }

    func updateNSView(_ nsView: NSTextView, context: Context) {
        nsView.textStorage?.setAttributedString(note.formattedBodyText)
        context.coordinator.note = note
    }

    class Coordinator: NSObject, NSTextViewDelegate {

        var note: Note
        let parent: TextViewMacosWrapper

        init(note: Note, parent: TextViewMacosWrapper) {
            self.note = note
            self.parent = parent
        }

        func textDidChange(_ notification: Notification) {
```

```
                    if let textview = notification.object as? NSTextView {
                        note.formattedBodyText = textview.attributedString()
                    }
                }
            }
        }
```

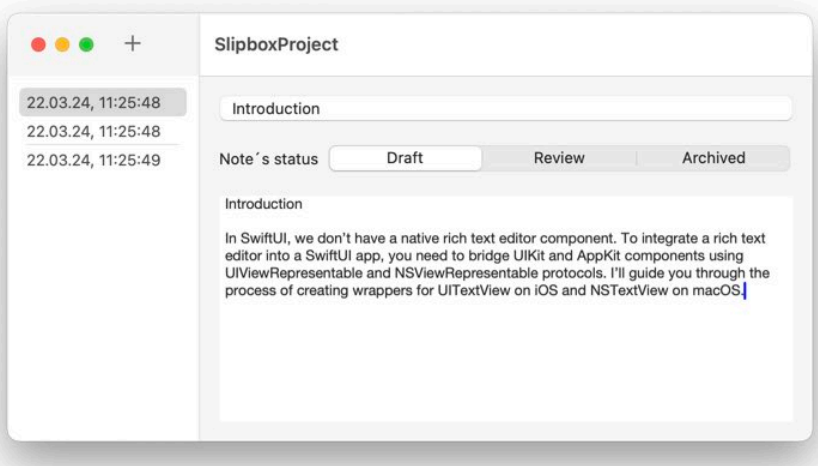
Other than using NSTextView instead of UITextView, this code is very similar to the iOS version. The main difference is how you can customize NSTextView. Here are some additional options:

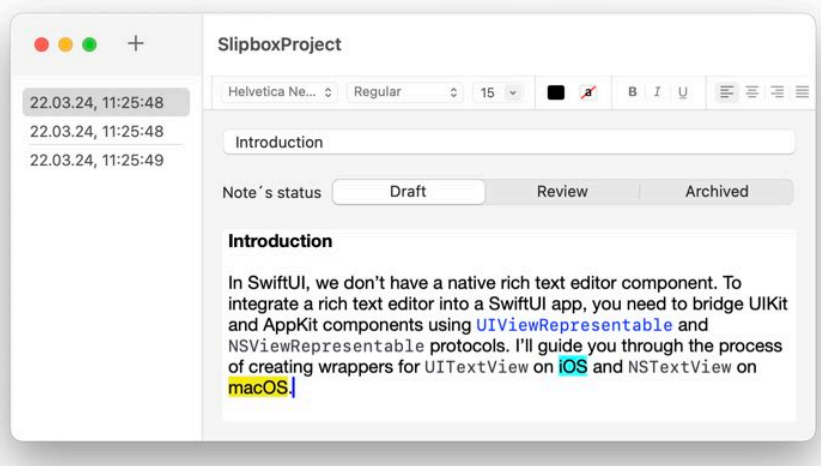
```
        func makeNSView(context: Context) -> NSTextView {
            let view = NSTextView()

            view.isRichText = true
            view.isEditable = true
            view.isSelectable = true
            view.allowsUndo = true

            view.usesInspectorBar = true

            view.usesFindPanel = true
            view.usesFindBar = true
            view.isGrammarCheckingEnabled = true

            view.isRulerVisible = true

            view.delegate = context.coordinator
            return view
        }
```

Back in NoteDetailView, add the platform check and show "**TextViewMacosWrapper**" for macOS:

```
        #if os(iOS)
            TextViewIOSWrapper(note: note)
        #else
            TextViewMacosWrapper(note: note)
        #endif
```

Build and run for macOS. For the basic NSTExtView (left) you cannot change the text to e.g. bold or make it larger. Adding the custom options to NSTExtView allows a rich text editor (right). You also get a menu in the toolbar to e.g. change the font, foreground color, and alignment:



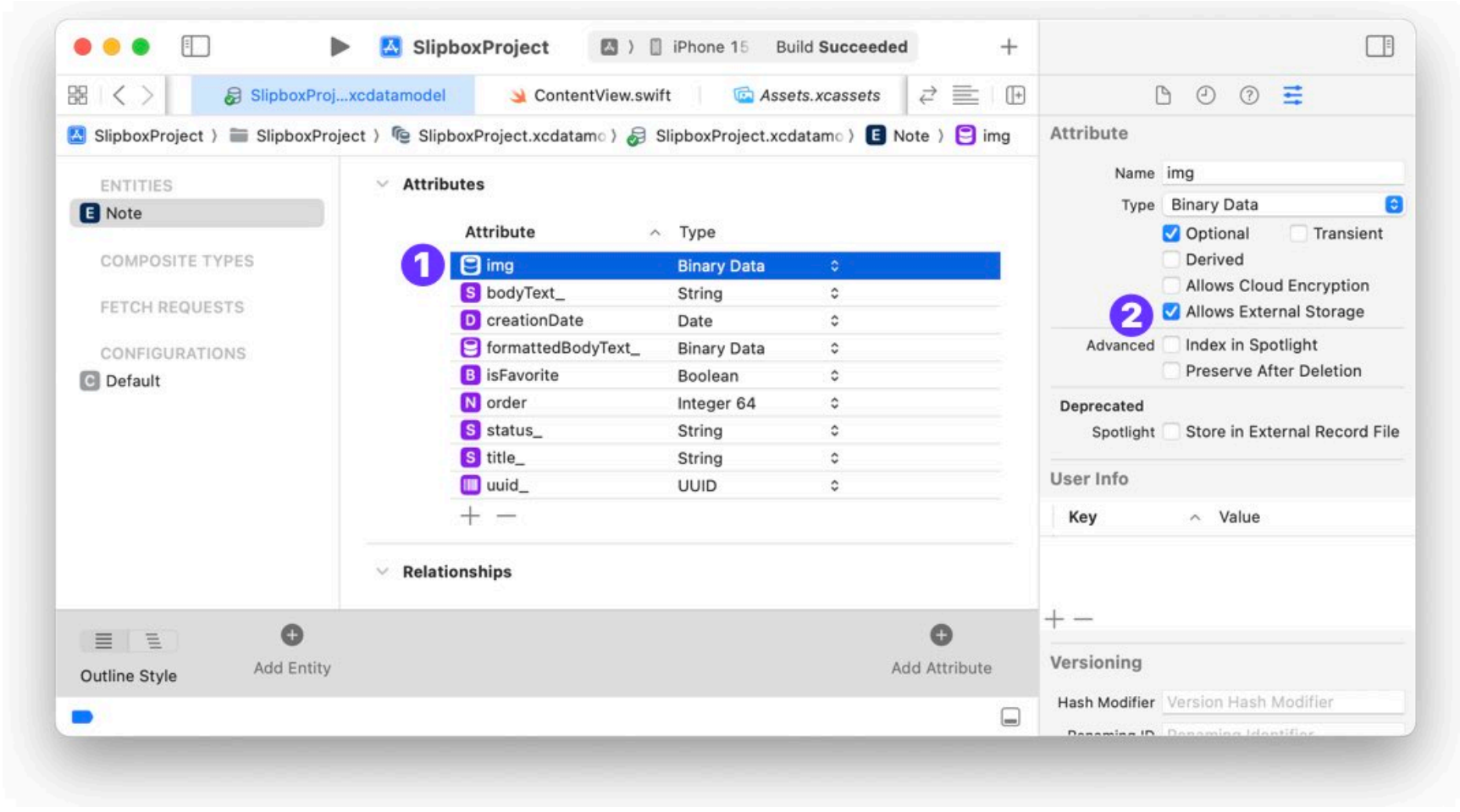


3.4 PHOTOPICKER AND SAVING IMAGES IN CORE DATA

In this section, I'll guide you through the process of handling images in your app and storing them in Core Data. We'll start by implementing the data model. Then you will use photo picker which allows users to select images from their photo library.

Storing Image Data

Open "xcdatamodeld" and add a new attribute "img". Set the type to "Binary Data" which is the most generic type. UIKit and AppKit have both functions included that allow you to convert from UIImage/ NSImage to Data. You will use this when we retrieve the image from ImagePicker.



Storing large images directly in your Core Data database can lead to performance issues. To avoid this, you should store images externally and only keep a reference in the database.

In the Core Data model editor, select the image attribute and check the **"Allows External Storage" option (2)**. This tells Core Data to store the image data as an external file rather than inside the database file. Core Data will handle the storage and keep reference to the external file URI. CoreData will store the data only externally if the data is very large ca. 1MB.

I will show you the database files later below. So you can see this behavior in action.

Displaying the Selected Image

To display the selected image, you need to convert the stored data back into an image. Create a new SwiftUI file named “OptionalImageView”. It takes in an optional Data type as input argument.

```
import SwiftUI

struct OptionalImageView: View {
    let data: Data?
    var body: some View {
        if let data = data, let uiImage = UIImage(data: data) {
            Image(uiImage: uiImage)
                .resizable()
                .scaledToFit()
        }
    }
}
```

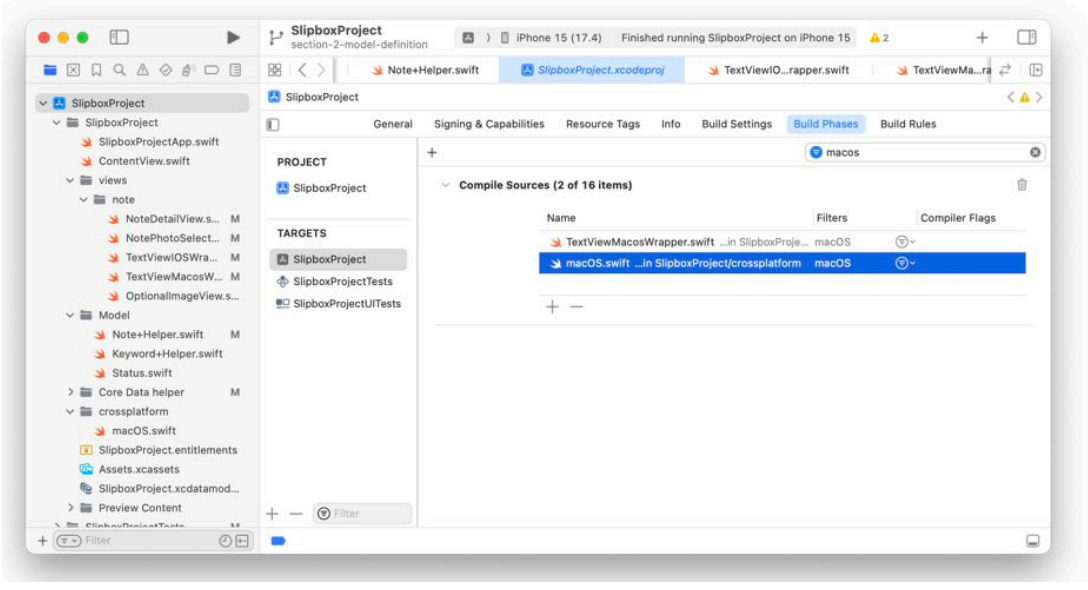
I use a conditional check to see if the data is not nil. If it is not nil I use UIImage(data:) to generate an UIImage. SwiftUI Image as initializers that take an UIImage as an argument.

This works for iOS, but will not build for macOS. There is a macOS version that uses NSImage. I want to handle this gracefully and tell the compiler to use NSImage for macOS and UIImage for iOS.

Create a new Swift file named “macOS.swift” in a new group “crossplatform” and add the following:

```
import SwiftUI

typealias UIImage = NSImage

extension Image {
    init(uiImage: UIImage) {
        self.init(nsImage: uiImage)
    }
}
```

The type alias tells Xcode to substitute the UIImage type as NSImage. I then create an initializer for Image that uses the macOS initializer Image(nsImage). This file should only run on macOS. Change the build settings in target editor:



Implementing the Photo Picker

First things first, you need to import the necessary framework to work with the photo picker. Add the following import statement at the top of your Swift file:

```
import PhotosUI
```

This API is available starting from iOS 15, so ensure your app targets iOS 15 or later. Next, you need to create a state property to hold the selected image:

```
@State private var selectedItem: PhotosPickerItem? = nil
```

Now, let's create the photo picker component. Here's how you do it:

```
PhotosPicker(selection: $selectedItem,
             matching: .images,
             photoLibrary: .shared()) {
    Text("import a photo")
}
```

This will open up the photo library and allow the user to select an image. The matching parameter specifies that we're only interested in images, not videos or other media types.

When a user selects an image, you need to handle the change and save the image data to Core Data. Here's how you can do that:

```
.onChange(of: selectedItem) { newValue in
    Task {
        if let data = try? await newValue?.loadTransferable(type: Data.self) {
            note.img = data
        }
    }
}
```

Here, I'm using an onChange modifier to listen for changes to selectedItem. When the selectedItem changes, a new Task is created to load the image data asynchronously. If successful, the image data is assigned to the image attribute of your Note entity.

This is the completed NotePhotoSelectorButton:

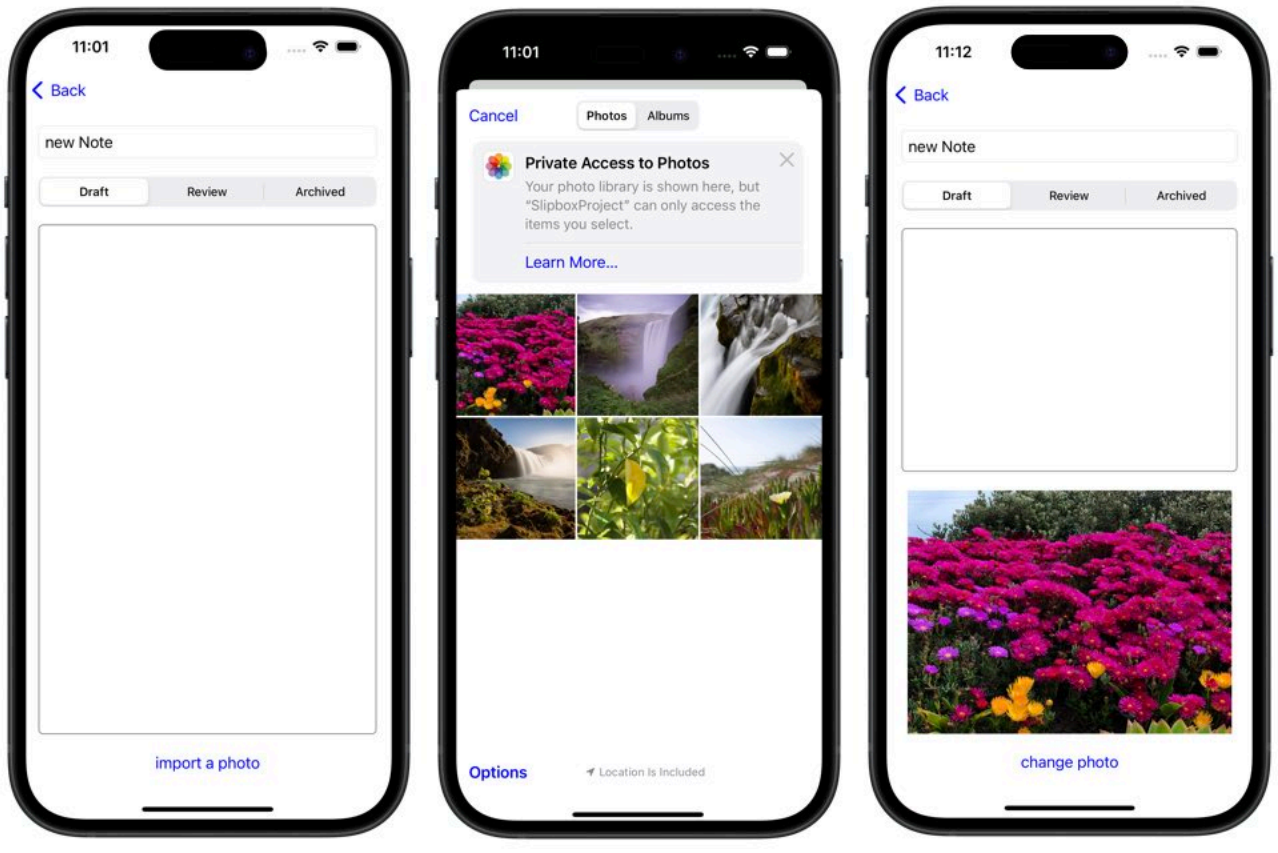
```
import SwiftUI
import PhotosUI

struct NotePhotoSelectorButton: View {

    @ObservedObject var note: Note
    @State private var selectedItem: PhotosPickerItem? = nil

    var body: some View {
        PhotosPicker(selection: $selectedItem,
                     matching: .images,
```

```
                        photoLibrary: .shared()) {
                if note.img == nil {
                    Text("import a photo")
                } else {
                    Text("change photo")
                }
            }
            .onChange(of: selectedItem) { newValue in
               Task {
                  if let data = try? await newValue?.loadTransferable(type: Data.self) {
                       note.img = data
                  }
               }
            }
        }
    }
```

Updating The UI to Load and Store Images

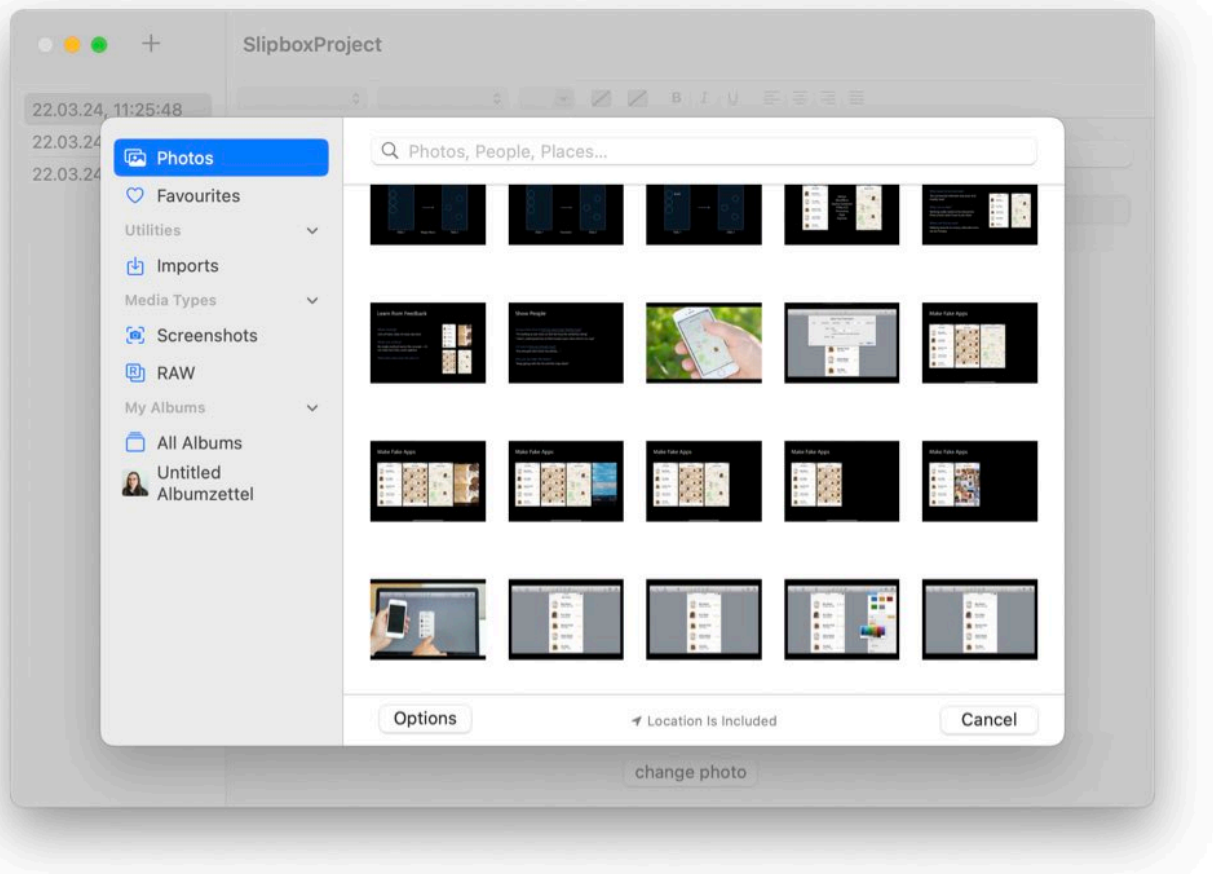
Now, that you have updated the Schema, created a View that can show the data of the Note image, and created a button to load images from the photos library, you can update NoteDetailView. At the end of the view add the new components “OptionalImageView” and “NotePhotoSelectorButton”:

```
    struct NoteDetailView: View {

        @ObservedObject var note: Note

        var body: some View {
            VStack(spacing: 20) {
                …
                OptionalImageView(data: note.img)
                NotePhotoSelectorButton(note: note)
            }
        }
    }
```



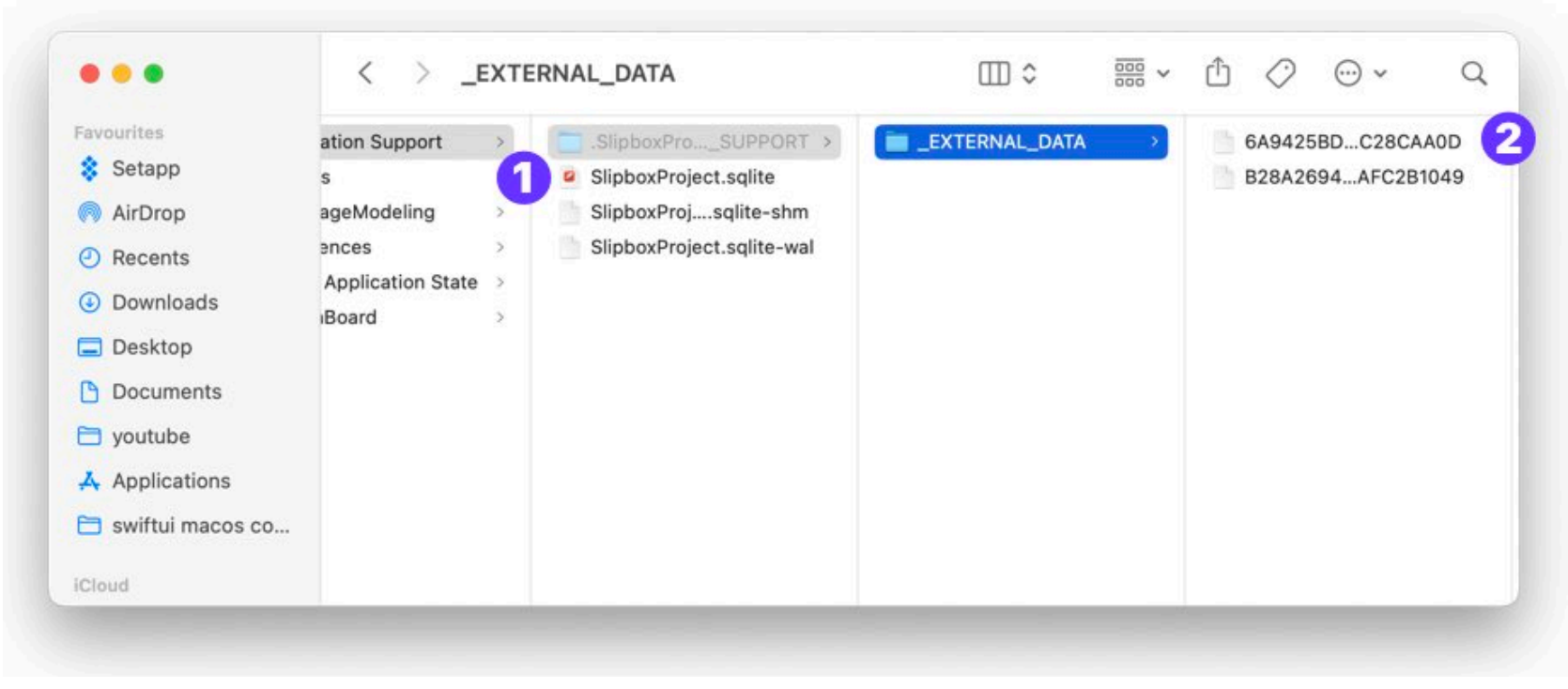
Build and run for iOS, open a note, and press the “import a photo” button. A sheet should open with the photo library. Select an image from the list. The sheet closes and the image should load with the “OptionalImageView”.

You can also test this on macOS. The photo picker is shown as follows:



Checking What data was Stored to File

After you added some data and closed the app (don’t press the stop button in Xcode or it will not be saved), we want to check what data was stored to file. Use the file url for the database and open the Finder app:

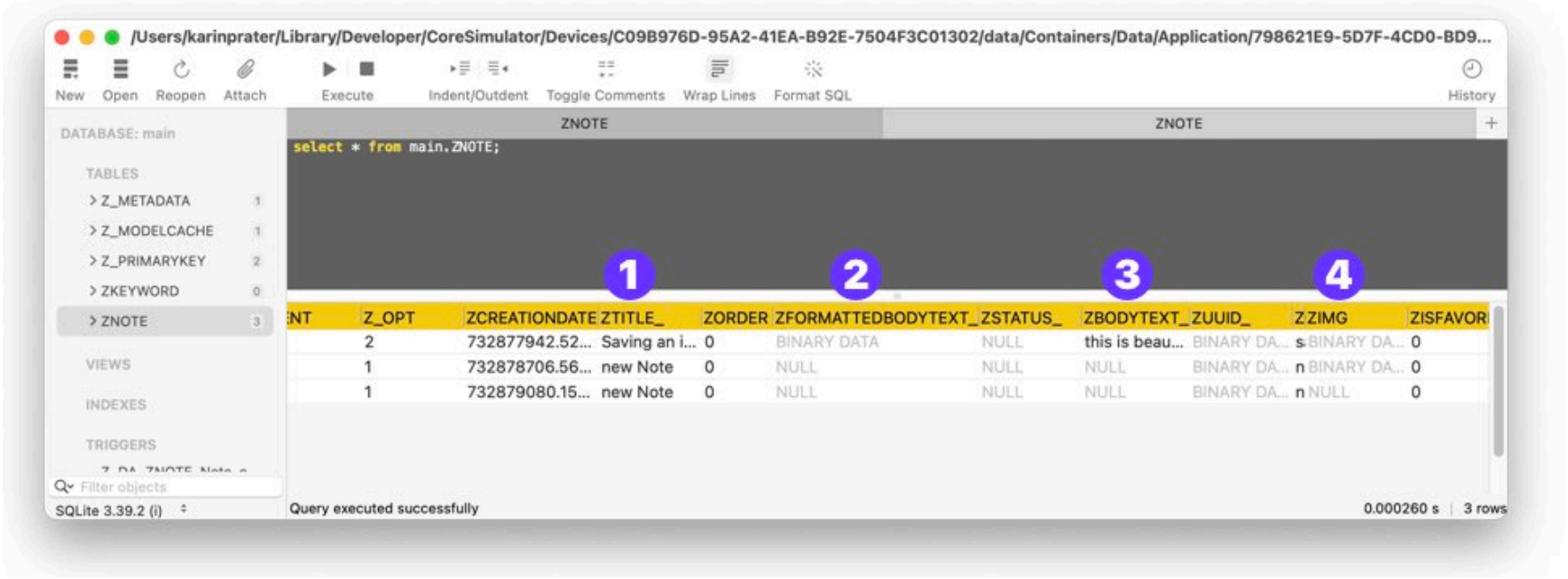


You will see the **SQLite file (1)** that stores the database including all notes that you create. Additionally, I folder is created that holds the **image data (2).** I saved 2 images with a large file size of 2.6MB and 4.1MB. This would be way to large to save in the database directly. The SQLite file in comparison is only 33KB. Thanks to the “allow External Storage” option we could optimize the database.

You can test this and run the simulator. Add more notes and images. You should not see file changes as long as you do not close the app (swipe up from the bottom edge of the screen). Close the app normally and see the files updating.

You can also delete notes in the app. After closing and saving, the corresponding files should disappear in Finder.

Let’s look at the SQLite file. Double click to open (1). The default Mac app should open. Below you see “Native SQLite Manager”:



Double tapping on “ZNOTE” in the left column, will open a table that shows the 3 notes that I created. The column names are mapped from the Core Data attributes with a “Z” prefix.

Have a look at the first row. You can see that I added a note body text because the “ZFORMATTEDTEXT_” column is not “NULL” but “BINARY DATA”. The corresponding “ZBODYTEXT_” column is of type string and you can see what I typed in the app “this is beautiful”.

This note also has an entry in the “ZIMG” column which refers to the stored image. The data that is stored here is the URI which is the file location.

You can find many apps in the app store that help you read the SQLite file. Have a look and choose your preference. This can be very helpful in understanding how your data is stored and managed.

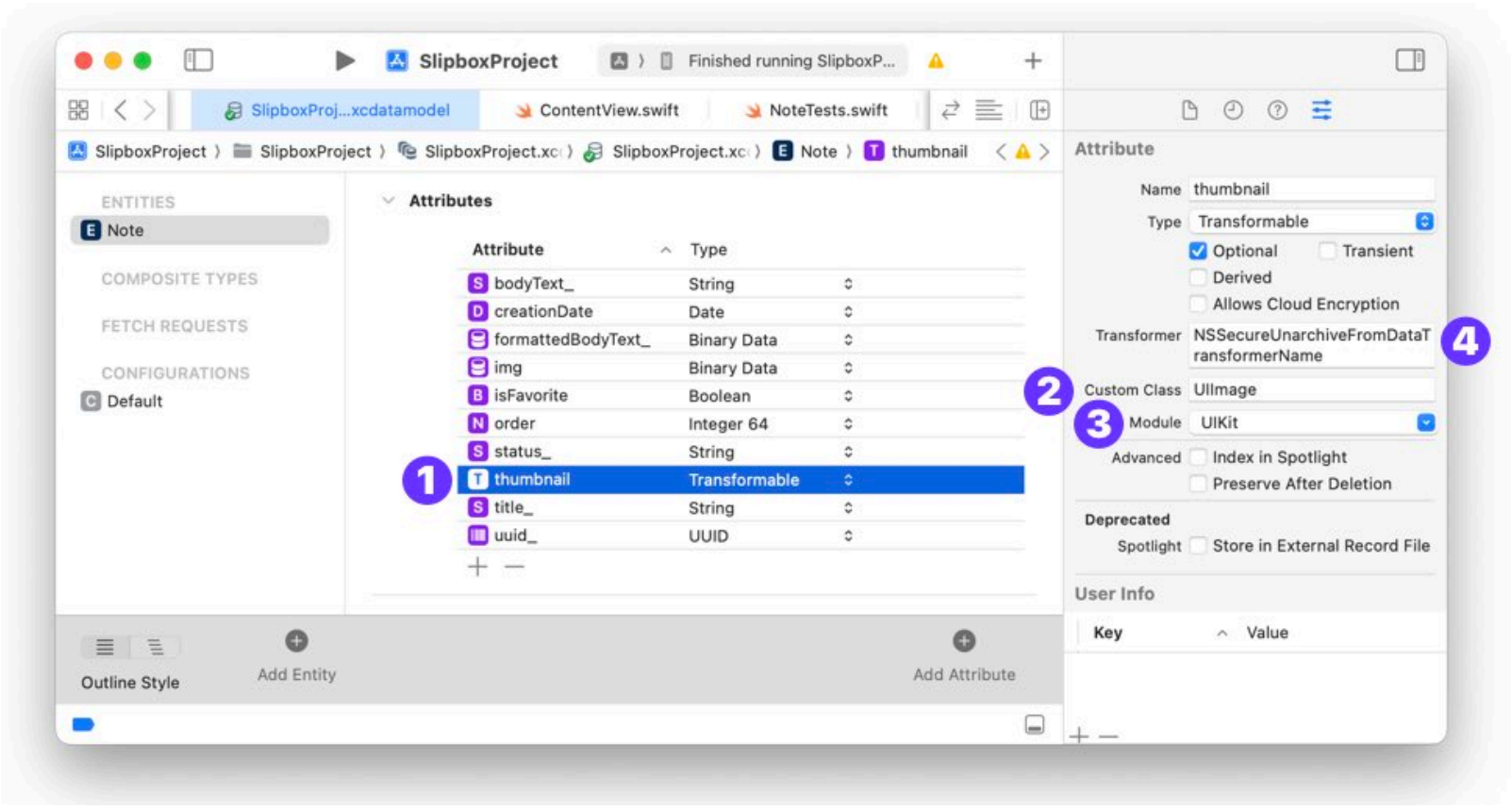
3.5 TRANSFORMABLE

An alternative to “Binary Data” is Transformable, which provides a more convenient way of converting your types to and from Data. Apple frameworks provide a lot of types that work directly with Transformable. As an example, I will use UIImage.

Create a new attribute “thumbnail” (1) in the Note schema and set the attribute type to Transformable. In the attribute inspector notice options for **transformer class (4)** set this to:

```
NSSecureUnarchiveFromDataTransformerName
```

Set the custom class to **“UIImage” (2)** and the Module to **“UIKit” (3).**



You probably see already one major drawback with this approach: you cannot conditionally chose UIKit/Appkit and UIImage/NSImage. Therefor this approach does not work for cross-platform applications.

If you work on an iOS only project, this works. Here is how you could update the NoteDetailView to use the “thumbnail” attribute. When accessing the “thumbnail” attribute in Swift, you will get a “UIImage?” type:

```
import SwiftUI

struct NoteDetailView: View {

    @ObservedObject var note: Note

    var body: some View {
        VStack {
            TextField("note", text: $note.title)
```

```
                NotePhotoSelectorButton(note: note)

                if let image = note.thumbnail {
                    Image(uiImage: image)
                        .resizable()
                        .scaledToFit()
                }
            }
            .padding()
        }
    }
```

When you save the image to Core Data, you can load the data, create an UIImage and set it to the “thumbnail” property:

```
    struct NotePhotoSelectorButton: View {

        @ObservedObject var note: Note
        @State private var selectedItem: PhotosPickerItem? = nil

        var body: some View {
           PhotosPicker {  …     }
             .onChange(of: selectedItem) { newValue in
                Task {
                  if let data = try? await newValue?.loadTransferable(type: Data.self) {
                      note.thumbnail = UIImage(data: data)
                }
             }
          }
    }
```

This works well, but personally I don’t see a big advantage over this approach compared to custom logic with “Binary Types”. You can compare transformable and “Binary Type” which you learned about in the previous section 3.4 PhotoPicker and saving images in Core Data.

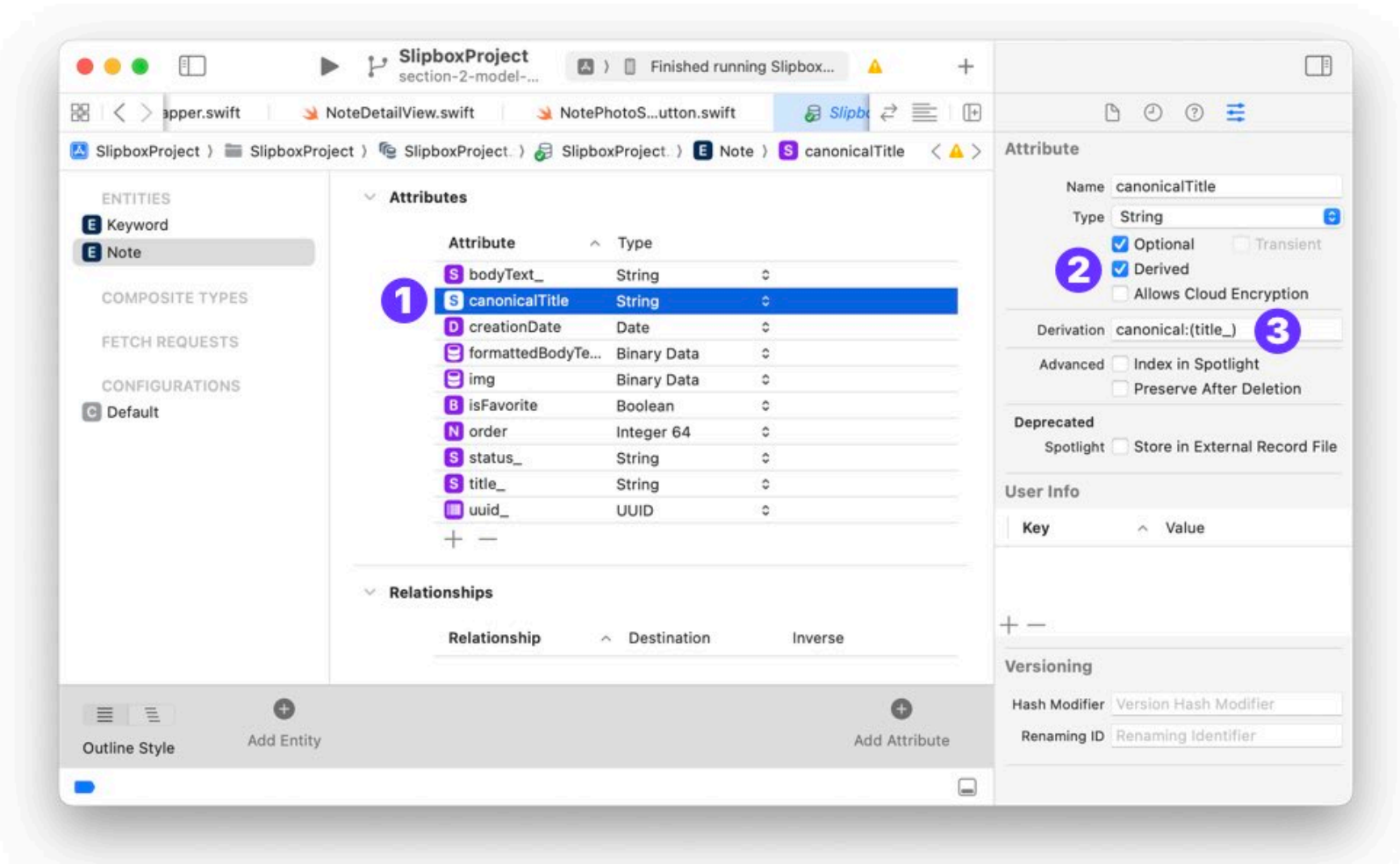
3.6 DERIVED DATA

In this section, we're going to delve into a more advanced feature of the Core Data schema: derived attributes. Derived attributes are incredibly useful, especially when you're looking to **optimize search performance** within your app. When marking an attribute as derived, the value of this attribute is derived from one or more other attributes.

Let's say you have a Note entity with a title property, which is a String. Imagine you're implementing a search feature and you want to find a note with "keyboard" in the title. If your search is case-sensitive, searching for "Keyboard" with a capital 'K' won't match a title stored as "keyboard" with a lowercase 'k'. This can be a problem.

You might consider **searching case-insensitive**, but beware, this can slow things down as it requires more comparisons. A better solution is to store a derived version of the title that is case-insensitive.

Here's an example of how you might set up such a derived attribute. Create a new attribute named **"canonicalTitle" (1) of type String**. In the attribute inspector enable the "**Derived" (2)** options:



Specify how the value is derived. Enter the following derivation (3):

```
canonical:(title_)
```

This will take the value of attribute "title__" and create a canonical version.

Another performance optimization is possible when you need to show the count of relationships. Later, we will add a Folder entity that points to Notes. A folder can have many notes. You could create a derived attribute “count” of type Int32 and use the following derivation:

```
notes.@count
```

You can also simply copy data. If you would want to show the name of the folder for each note, you could get the notes folder and get its name property. But every time you access an object, you will fetch the full objects with all attributes. Instead, you could add a derived attribute “folderName" to each note and use the following derivation:

```
folders.title
```

Remember, the goal here is to have a property that’s optimized for performance. You don’t necessarily want or need to display this derived data in your views.

To summarize, you have two ways to manage derived properties in Core Data:

1. **Core Data Handled**: Core Data automatically manages the derived attribute, but it only updates on save and doesn’t work with SwiftUI’s data observation directly.

2. **Manual**: You manually manage a property using getters and setters, giving you full control over when and how the property is updated. (See formattedBodyText_ in section 3.3)

By understanding and using derived attributes effectively, you can significantly improve the performance and functionality of your Core Data-backed SwiftUI applications.

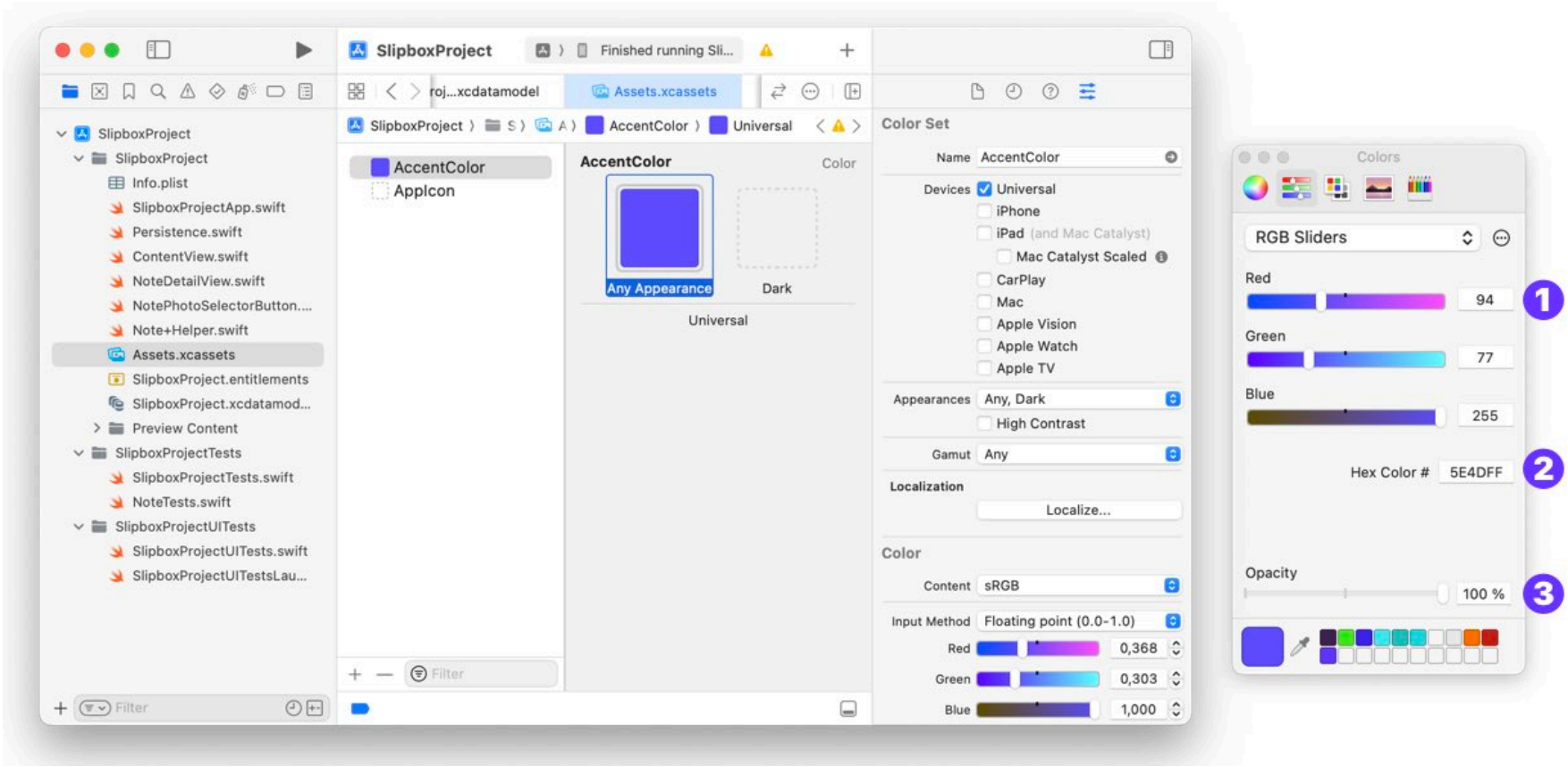
3.7 SAVING COLOR AS HEX VALUE

In this section, I'll guide you through the process of storing colors in Core Data using hex values. Unlike other data types, colors require a bit of finesse to handle correctly in a database. Let's dive into the details.

Understanding Color Data Types

First, I want to explore what a color data type really is. Colors can be represented in different ways, but the most common are RGB (Red, Green, Blue) and grayscale. RGB colors are composed of three integer values ranging from 0 to 255, as well as an opacity value. Conversely, grayscale colors are represented by a single white value.

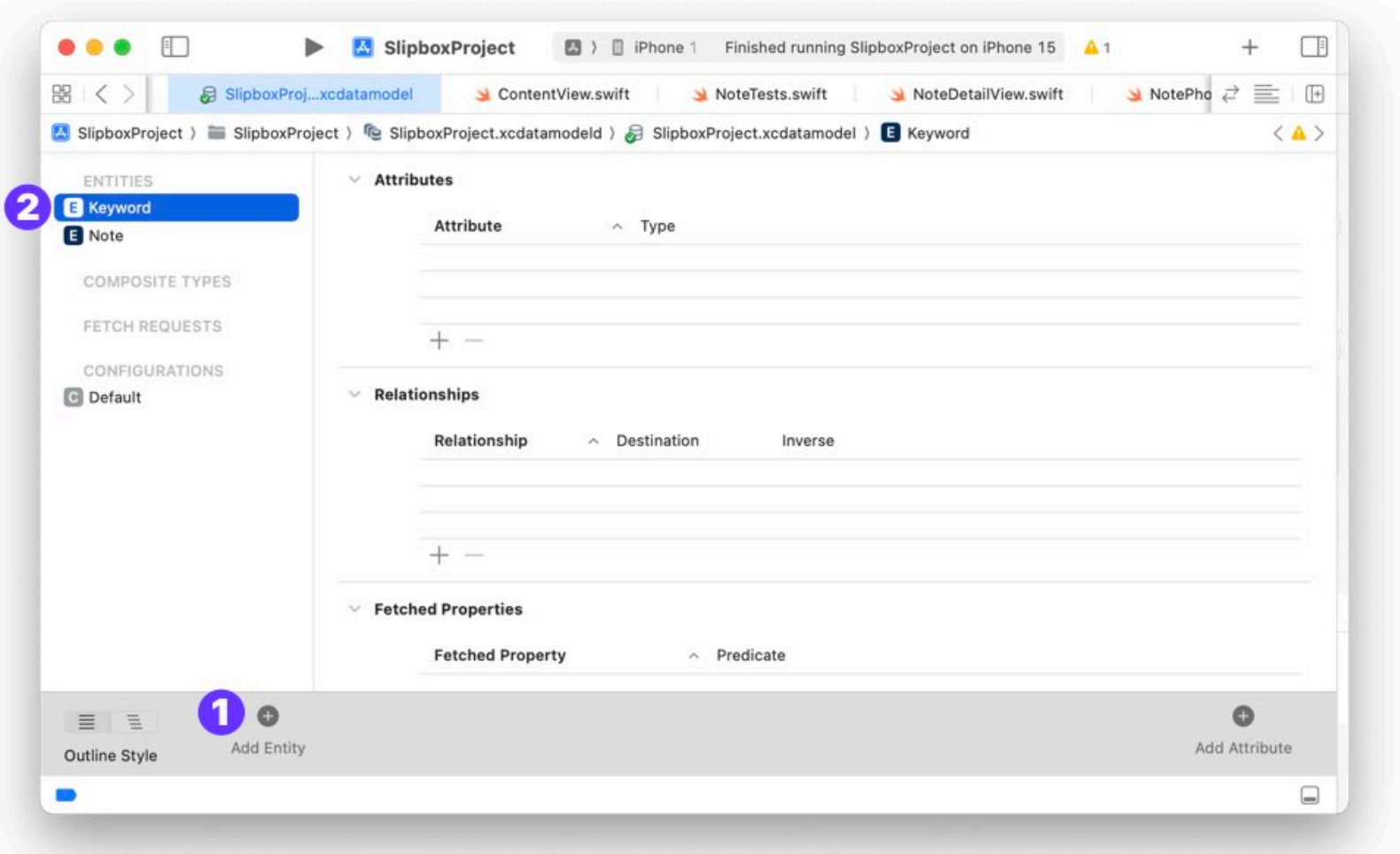
Another way to represent colors is with a **hex string (3)**—a single string value that encapsulates the RGB components and, optionally, the opacity. This string format is particularly database-friendly.



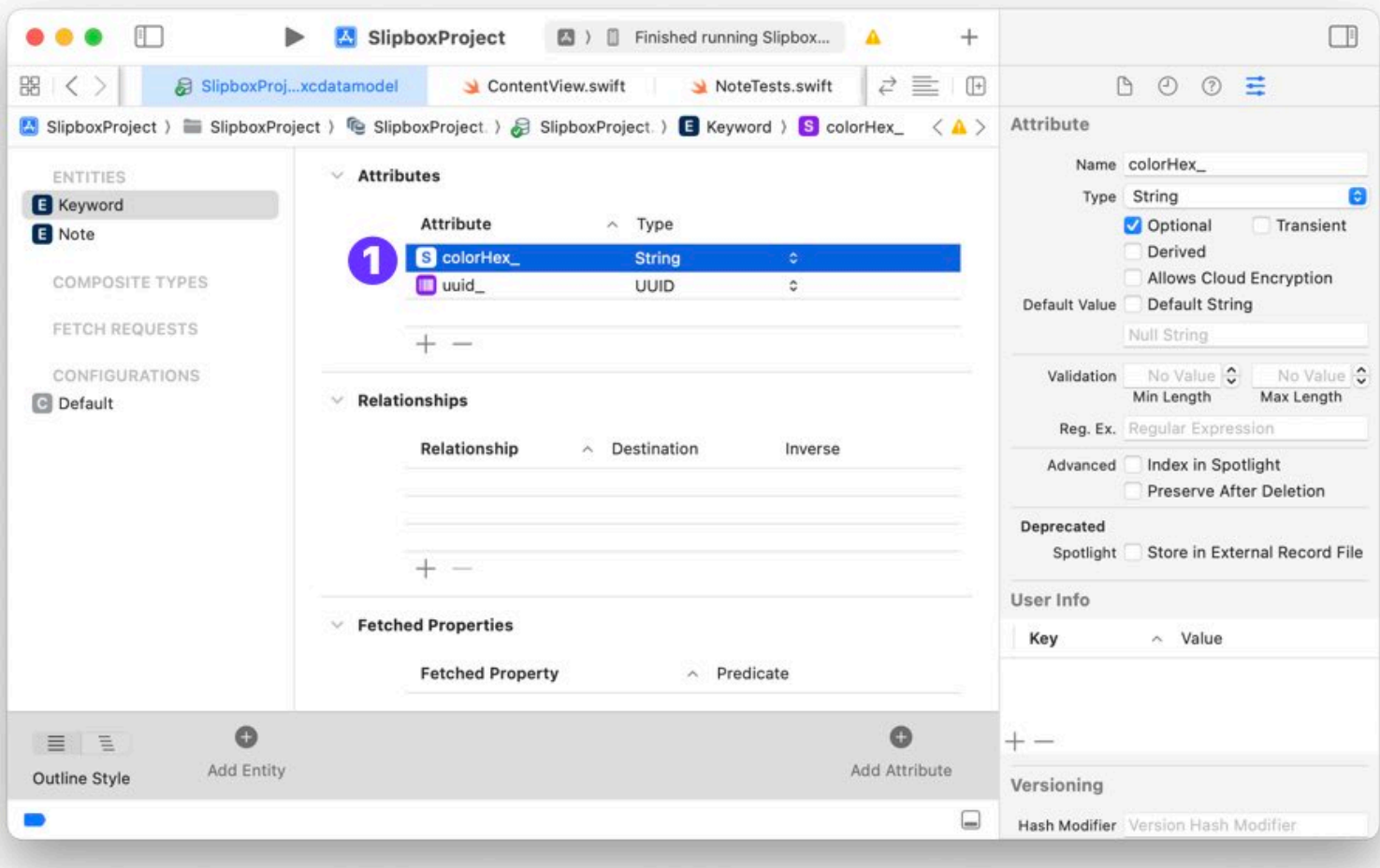
Setting Up Core Data

When configuring your Core Data model, you have the option to represent the color.

First, create a **new entity (1)** and name it **Keyword (2)**



Add an attribute named “colorHex_” and set the type to String:



Creating SwiftUI Color from HexColors

SwiftUI and UIKit do not provide functions to create color types from hex strings. Create a new Swift file and name it “Color+Helper.swift”. Past in the following code that adds conversion code in an extension to Color:

```
import SwiftUI

extension Color {

    init?(hex: String) {
        var hexNormalized = hex.trimmingCharacters(in: .whitespacesAndNewlines)
        hexNormalized = hexNormalized.replacingOccurrences(of: "#", with: "")

        var rgb: UInt64 = 0
        var r: Double = 0.0
        var g: Double = 0.0
        var b: Double = 0.0
        var a: Double = 1.0
        let length = hexNormalized.count

        Scanner(string: hexNormalized).scanHexInt64(&rgb)

        if length == 6 {
            r = CGFloat((rgb & 0xFF0000) >> 16) / 255.0
            g = CGFloat((rgb & 0x00FF00) >> 8) / 255.0
            b = CGFloat(rgb & 0x0000FF) / 255.0

        } else if length == 8 {
            r = CGFloat((rgb & 0xFF000000) >> 24) / 255.0
            g = CGFloat((rgb & 0x00FF0000) >> 16) / 255.0
            b = CGFloat((rgb & 0x0000FF00) >> 8) / 255.0
            a = CGFloat(rgb & 0x000000FF) / 255.0
        } else {
            return nil
        }

        self.init(red: r, green: g, blue: b, opacity: a)
    }


    func toHex() -> String? {
        guard let components = cgColor?.components, components.count > 2 else {
            return nil
        }

        let r = Float(components[0])
        let g = Float(components[1])
        let b = Float(components[2])
        var a: Float = 1

        if components.count == 4 {
            a = Float(components[3])
        }

        let hex = String(format: "%02lX%02lX%02lX%02lX",
                         lroundf(r * 255),
                         lroundf(g * 255),
                         lroundf(b * 255),
                         lroundf(a * 255))
        return hex
    }
}
```

Using Color Extensions in Core Data

To use the color conversions in Core Data, you can create an extension for your Core Data entity Keyword:

```
import SwiftUI

extension Keyword {

    var colorHex: Color {
        get {
            if let colorHexValue = colorHex_,
               let color = Color(hex: colorHexValue) {
               return color
            } else {
               return Color.black
            }
        }
        set {
           colorHex_ = newValue.toHex()
        }
    }
}
```

I am creating a computed property “colorHex” that gets the “colorHex_” value and converts it into a SwiftUI color. When I set the color e.g. with a SwiftUI color picker, the setter uses the “toHex” function and sets a new string value to “colorHex_”.

Testing Your Implementation

Testing is crucial, especially when you’re not dealing with UI elements yet. Here’s how you might write tests for your color conversions:

```
import XCTest
@testable import SlipboxProject
import SwiftUI

final class ColorTests: XCTestCase {


    func test_Hex_Color() {

        let colorBlue = Color(hex: "0000FF")
        let colorBlueAlpha = Color(hex: "0000FFFF")

        let referenceColorBlue = Color(red: 0, green: 0, blue: 1)

        XCTAssertTrue(referenceColorBlue == colorBlue)
        XCTAssertTrue(referenceColorBlue == colorBlueAlpha)
    }

    func test_Color_to_Hex() {

        let referenceColorBlue = Color(red: 0, green: 0, blue: 1)
```

```
            let hex = referenceColorBlue.toHex()

            XCTAssertTrue(hex == "0000FFFF")
        }
    }
```

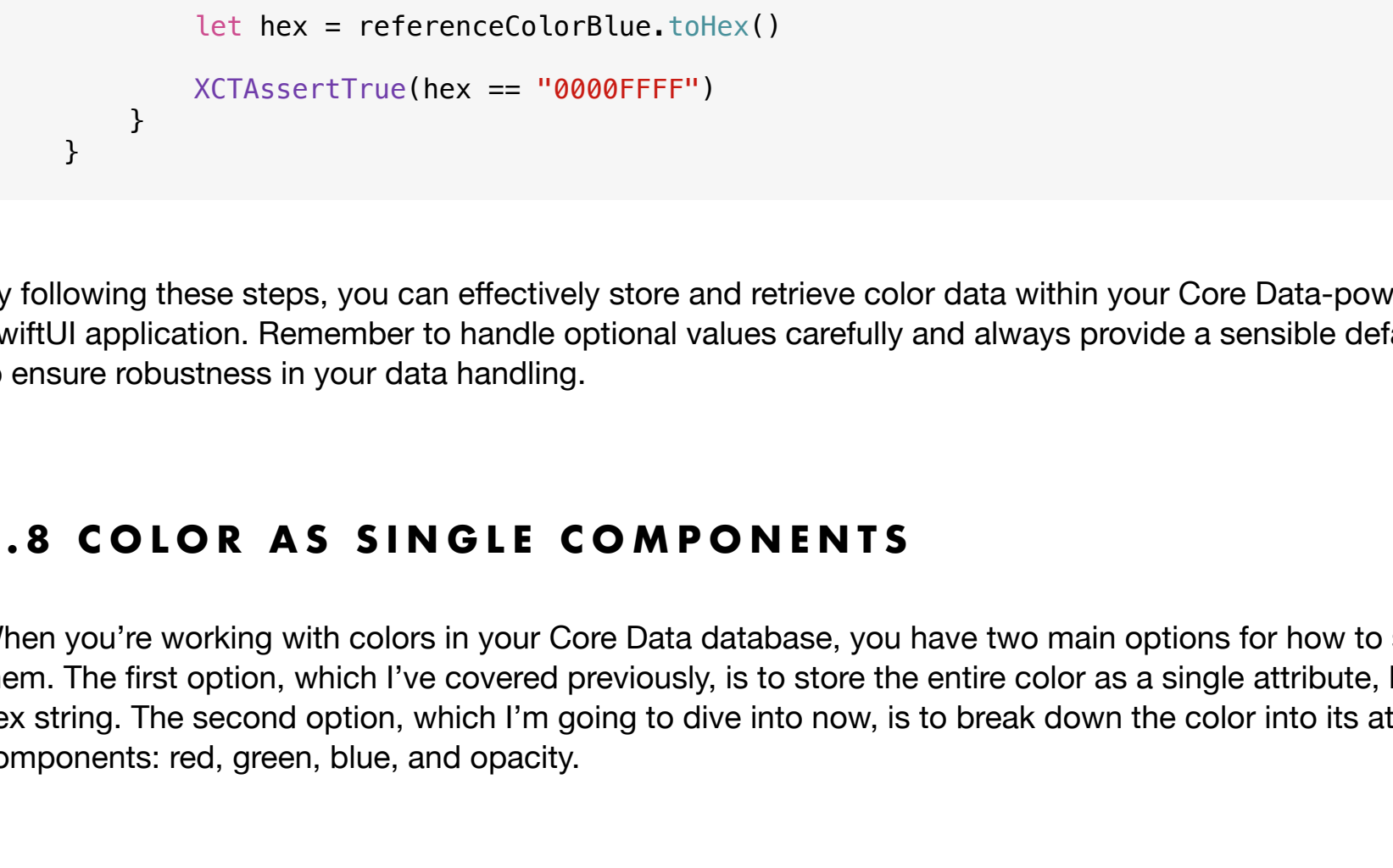
By following these steps, you can effectively store and retrieve color data within your Core Data-powered SwiftUI application. Remember to handle optional values carefully and always provide a sensible default to ensure robustness in your data handling.

3.8 COLOR AS SINGLE COMPONENTS

When you're working with colors in your Core Data database, you have two main options for how to store them. The first option, which I've covered previously, is to store the entire color as a single attribute, like a hex string. The second option, which I'm going to dive into now, is to break down the color into its atomic components: red, green, blue, and opacity.

Storing RGB and Opacity

Instead of storing a color as one attribute, I'm going to split the color into its basic building blocks: red, green, blue, and opacity. These are all numbers, so they fit nicely into the Core Data model. Here's how I define these attributes in my Core Data model. Create 4 attributes "blue_", "red_", "green_" and "opacity_":



Each color component is a Double. I choose Double over Float to avoid any unnecessary type conversions since the Color initializer in SwiftUI expects Double values.

I set the **default value (1)** to 0 for “red_”, “green_” and “blue_”. For “opacity_” I use 1 to make the color fully opaque. This corresponds to a black color. Change these color values here, for a different default value

Creating a Color Computed Property

To make it easy to work with these color components, I create a computed property on my Core Data entity extension that can combine these values into a single SwiftUI Color:

```
import SwiftUI

extension Keyword {

    var color: Color {
        get {
            Color(red: red_, green: green_, blue: blue_, opacity: opacity_)
        }
        set {
            guard let components = newValue.cgColor?.components,
                    components.count > 2 else { return }

            red_ = components[0]
            green_ = components[1]
            blue_ = components[2]

            if components.count == 4 {
                opacity_ = components[3]
            } else {
                opacity_ = 1
            }
        }
    }
}
```

In this computed property, the getter constructs a Color using the stored values. The setter takes a Color, breaks it down into its components, and stores those back into the entity.

Testing Color Values

I always recommend writing tests to verify that your colors are being stored and retrieved correctly. Here’s an example of how you might test the default color:

```
import XCTest
@testable import SlipboxProject
import SwiftUI
import CoreData

final class KeywordTests: XCTestCase {
```

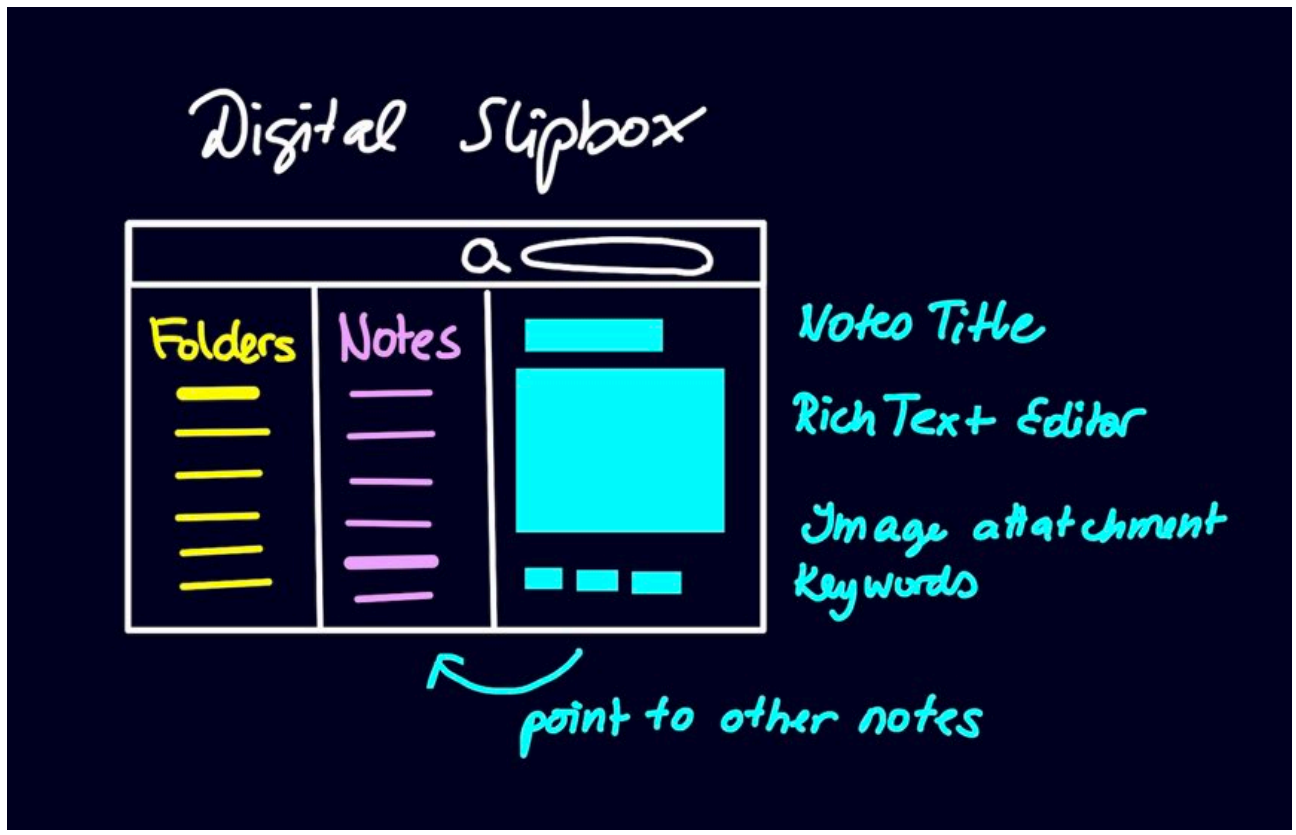
```
    var controller: PersistenceController!

    var context: NSManagedObjectContext {
        controller.container.viewContext
    }

    override func setUpWithError() throws {
        self.controller = PersistenceController.createEmpty()
    }

    override func tearDownWithError() throws {
        self.controller = nil
    }

    func test_Keyword_default_color() {

        let black = Color.black
        let keyword = Keyword(context: context)
        let defaultColor = keyword.color

        XCTAssertTrue(defaultColor == black)
    }

    func test_Keyword_component_Color() {

        let color = Color(red: 0, green: 0, blue: 1)
        let keyword = Keyword(context: context)
        keyword.color = color

        let retrievedColor = keyword.color

        XCTAssertTrue(retrievedColor == color)
    }
}
```

By breaking down a color into its atomic components, you gain a lot of flexibility. You can easily change individual color attributes without affecting others, and you avoid the complexities of dealing with hex values. This approach is straightforward and aligns with Core Data's capabilities, making it a practical solution for storing color information in your database.

4. RELATIONSHIPS

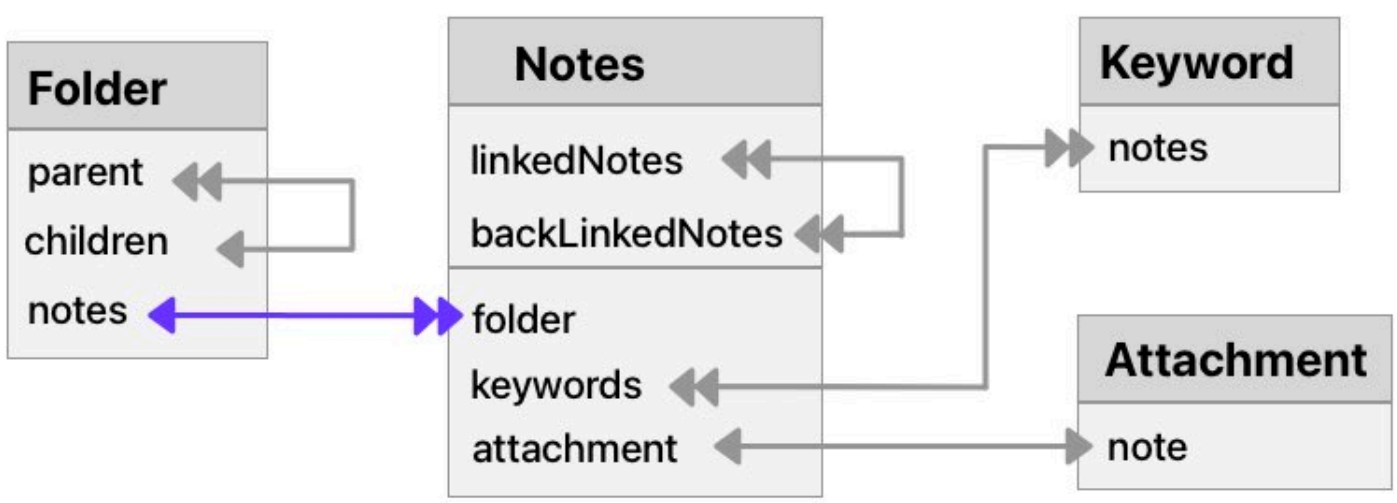
4.1 INTRODUCTION TO RELATIONSHIPS

So far, I've been focusing on the Node entity and exploring the various attributes we can define within it. Now, it's time to dive into another fundamental aspect of Core Data: relationships.
In our example app, we've dealt with a list of notes and a detail view for each note. It's pretty straightforward, but I'm going to introduce a new layer of complexity: folders.



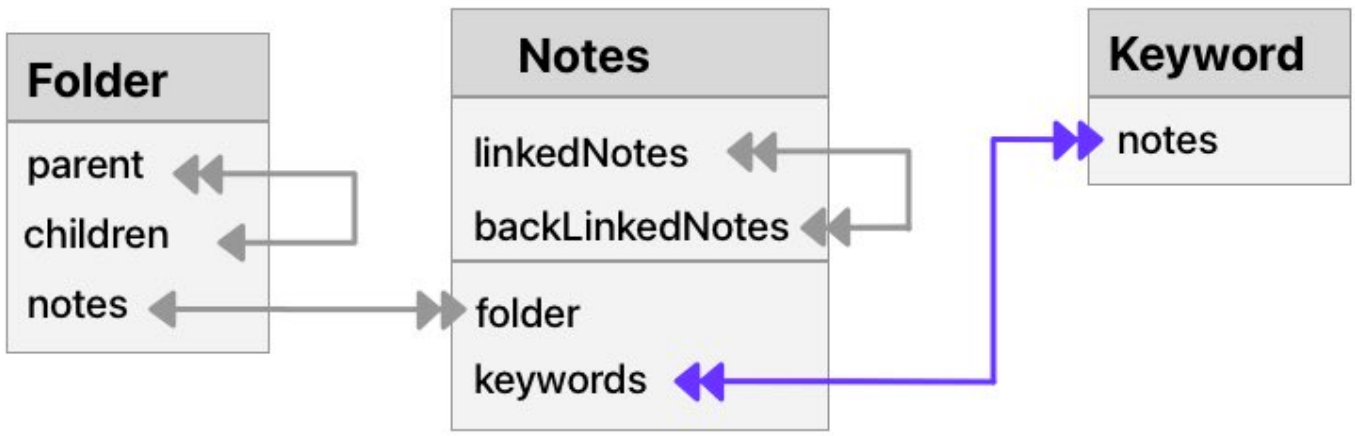
Concept of Folders and Notes

Imagine each folder in our app can contain many notes. Conversely, each note belongs to exactly one folder. This is a classic **one-to-many relationship**, with one folder having multiple notes. The graphical representation is to-one **—>** and to-many **—>>**



Keywords and Multiple Images

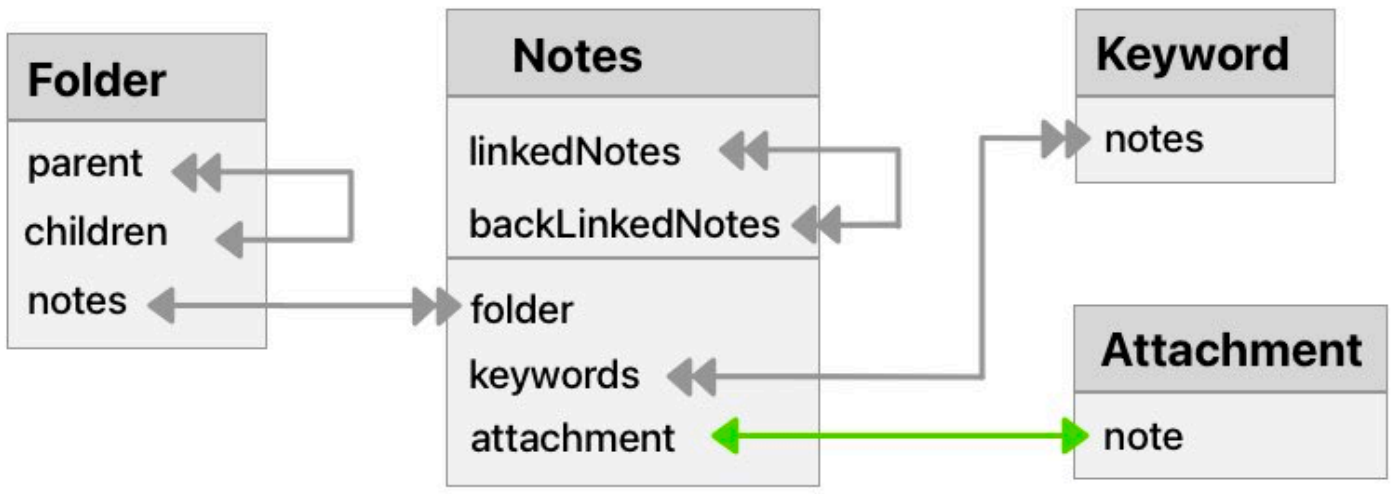
Let's consider keywords. A note can be associated with multiple keywords, and each keyword can be linked to many notes. This **many-to-many relationship** allows us to create a versatile tagging system.



Handling Image Attachments

When you access instances of an entity that have large data like images stored, you load all data into memory even if you don't use this data. This means you can run into performance issues. Instead, we will store the image data in a separate entity "Attachment" and connect the note and attachment with a one-to-one relationship.

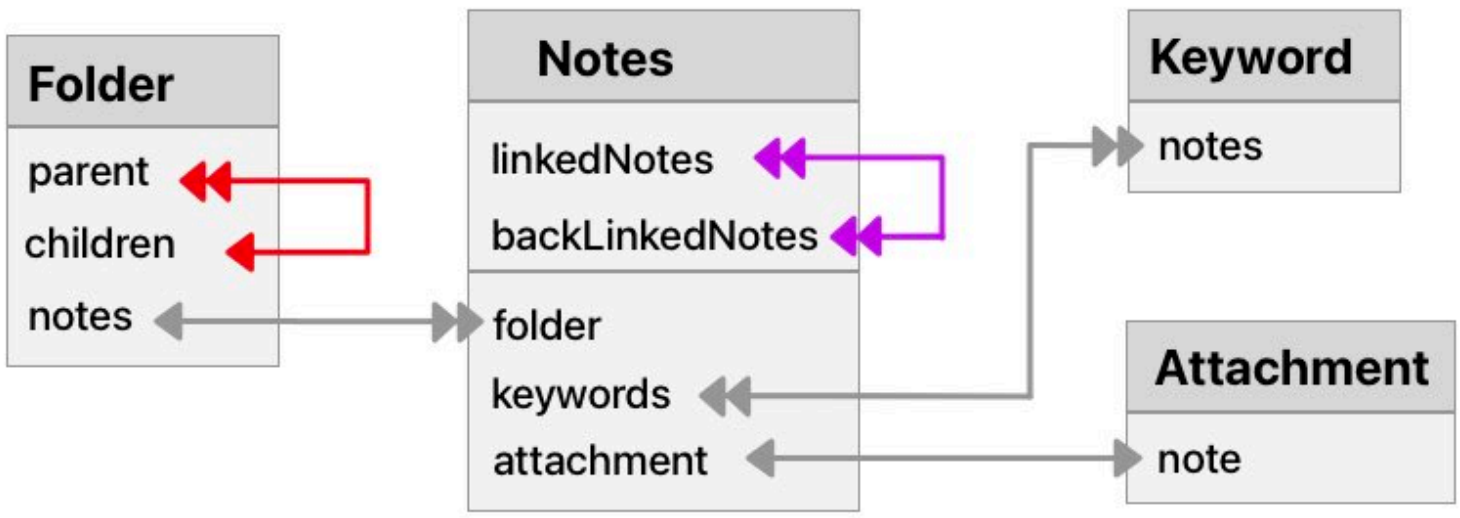
Additionally, you can store smaller versions of the images, like thumbnails. You can link a note to an image attachment entity for the thumbnail. This image attachment can then link to another image attachment entity that holds the full-sized image. You will set this up as a **one-to-one relationship**. This approach can **increase performance** and enhance reusability.



Currently, a note might have a single image attribute. This works well when there's only one image per note. However, if we want to attach multiple images to a note, we need to set up a one-to-many relationship between the note and its images. If you want to be able to share the same images for multiple notes, you can use a many-to-many relationship.

Hierarchical Folder Structures

Folders aren't just flat containers; they can have subfolders. Each folder can potentially be a parent folder to many other child folders, and each child folder has one parent folder. This hierarchy introduces a **one-to-many relationship** within the folders themselves.



Inter-Note Relationships

Notes can also reference other notes. This means we can have links from one note to another, creating a network of interconnected notes. Additionally, we can track backlinks, showing which notes are linked to a particular note.

Setting Up the Data Model

To get started, you'll first add the Folder entity to the data model. Then, you'll establish the relationships I just described.

I'll demonstrate how these relationships work using unit tests. This way, we can keep our tests focused and also examine the effects of different delete rules on relationships.

After setting up the relationships, we'll integrate them into the UI. For example, to display notes within a folder, I'll show you how to create fetch requests with specific filtering using NSPredicate.

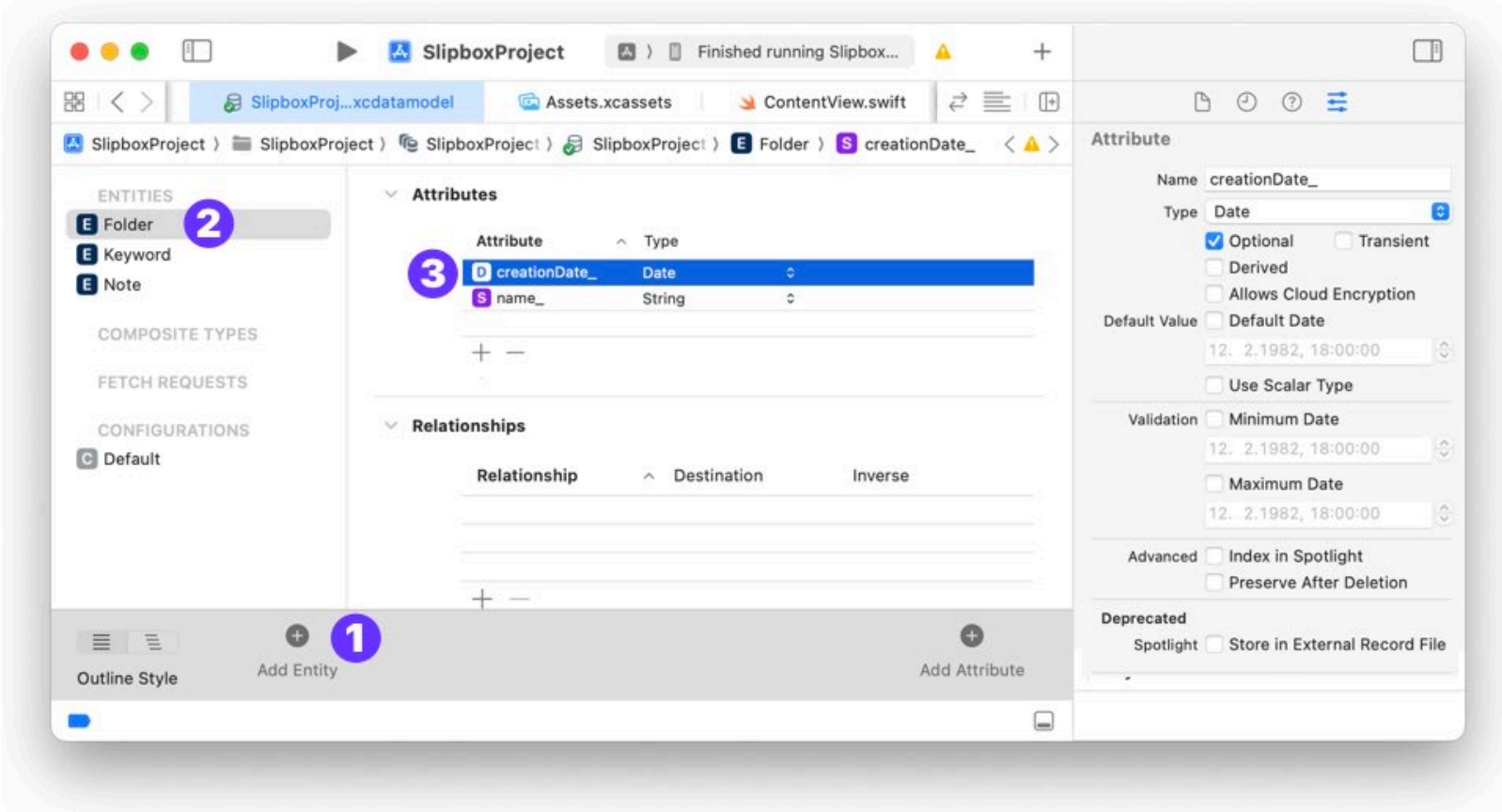
Working with relationships in a database like Core Data can initially seem daunting. But by the end of this section, you should have a solid grasp of how to implement and work with relationships in your apps. We'll explore a variety of relationship types and scenarios, ensuring you know how to apply these concepts to your projects.

4.2 FOLDER ENTITY

In this lesson, I'm going to guide you through the process of adding a Folder entity to our Core Data model. You'll learn how to create domain attributes and run some tests to ensure everything is working as expected.

First things first, let's create the entity and add attributes:

- Open your .xcdatamodeld file.
- Add a **new entity (1)** and name it "**Folder**" **(2)**.
- Add to 2 new **attributes (3).** Attribute "name_" of type String and "creationDate_" of type Date



Next, I want to create a Swift extension to make working with the Folder entity more convenient. So, in the same directory as my Note extension, I'll create a new Swift file named **"Folder+Helper.swift"**. This will be an extension to the Folder entity.

In the extension, I'll add computed properties to handle the optional attributes and a convenience initializer:

```
import Foundation
import CoreData

extension Folder {

    var name: String {
        get { name_ ?? "" }
        set { name_ = newValue }
    }
```

```
        var creationDate: Date {
            get { creationDate_ ?? Date() }
        }

        convenience init(name: String, context: NSManagedObjectContext) {
            self.init(context: context)
            self.name = name
        }

        public override func awakeFromInsert() {
            self.creationDate_ = Date()
        }
    }
```

Now, let's tackle fetching Folder instances. I'll write a static function to fetch Folder entities with a given predicate. This will be useful later when we need to filter or search for specific folders:

```
    extension Folder {

         …

        static func fetch(_ predicate: NSPredicate) -> NSFetchRequest<Folder> {
            let request = Folder.fetchRequest()
            request.sortDescriptors = [NSSortDescriptor(keyPath: \Folder.creationDate_,
                                                        ascending: true)]
            request.predicate = predicate
            return request
        }
    }
```

I set the sort order to use the creation date. Later, in the UI I want to show all folders ordered from oldest to newest.

Unit Tests for Folder

I'm also going to write some tests to ensure our fetching functionality works as intended. I'll create a new file for unit tests named **"FolderTests.swift"** and write the following test cases:

```
    import XCTest
    @testable import SlipboxProject
    import CoreData

    final class FolderTests: XCTestCase {

        var controller: PersistenceController!

        var context: NSManagedObjectContext {
            controller.container.viewContext
        }

        override func setUpWithError() throws {
            self.controller = PersistenceController.createEmpty()
        }
```

```
        override func tearDownWithError() throws {
            self.controller = nil
        }

        func test_Folder_creationDate() {
            let folder = Folder(name: "new", context: context)
            XCTAssertNotNil(folder.creationDate_,
                            "folder should have creation date")
        }

        func test_fetch_all_folders() {

            let folder1 = Folder(name: "first", context: context)
            let folder2 = Folder(name: "second", context: context)

            let request = Folder.fetch(.all)
            let retrievedFolders = try? context.fetch(request)

            XCTAssertNotNil(retrievedFolders)
            XCTAssertTrue(retrievedFolders?.count == 2,
                          "should retrieve 2 folders that where added to database")
            XCTAssertTrue(retrievedFolders!.contains(folder1))
            XCTAssertTrue(retrievedFolders!.contains(folder2))

            XCTAssertTrue(retrievedFolders?.first == folder1)
            XCTAssertTrue(retrievedFolders?.last == folder2)

        }
    }
```

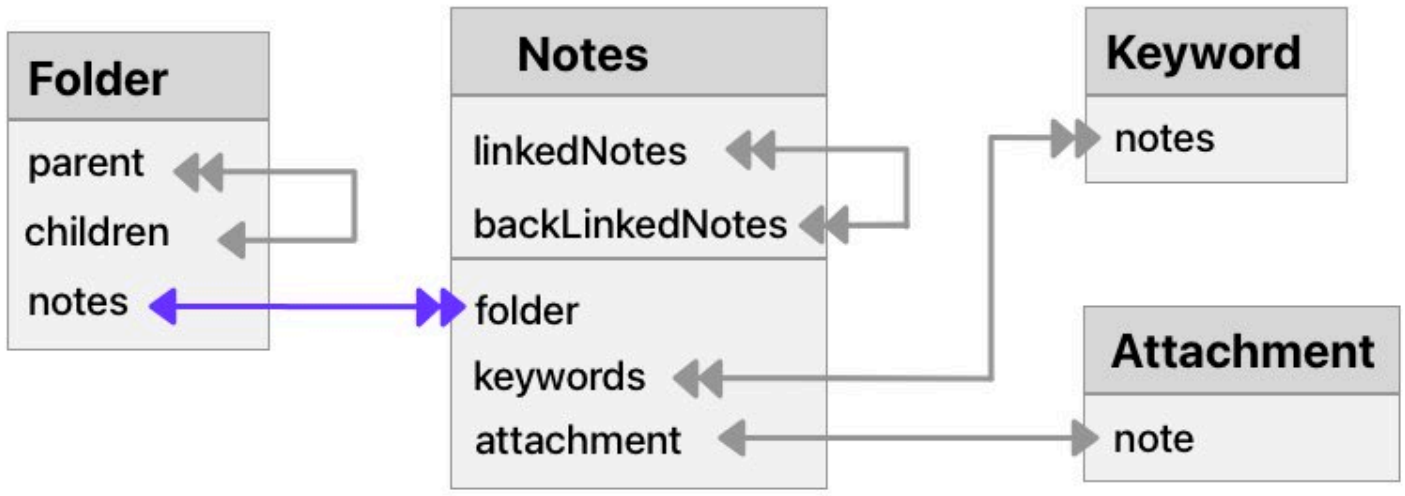
In the testFetchFolders function, I'm testing the order of the retrieved folders. If you're ever confused about the ascending parameter, you can run these tests to see if the sorting is working as expected. I am checking what the first and last folders are that I get back from my fetch request:

```
        XCTAssertTrue(retrievedFolders?.first == folder1)
              XCTAssertTrue(retrievedFolders?.last == folder2)
```

That's it for setting up the Folder entity and its attributes. In the next lesson, we'll dive into defining and managing relationships between our Folder and Note entities.

4.3 FOLDER NOTES RELATIONSHIP: ADDING LINKS AND DELETE RULES

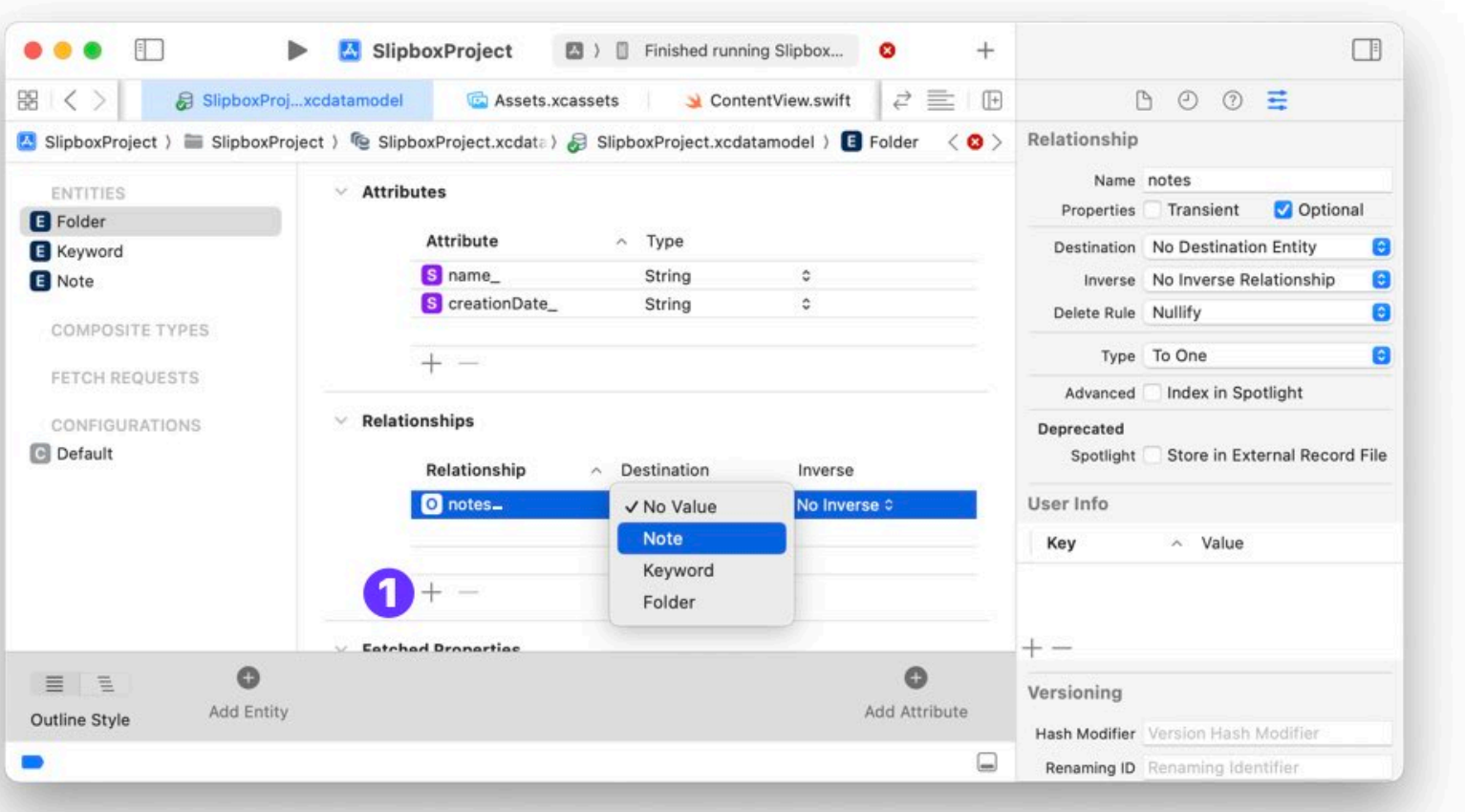
In this section, I'll guide you through setting up and managing the relationship between folders and notes in Core Data. We'll look at how to add and configure relationships in the Core Data model, and understand the implications of different delete rules.



Defining the Relationship

First, let's set up the relationship in our Core Data model. Assuming you have already defined the Note and Folder entities, you'll go to your .xcdatamodeld file and select the Folder entity.

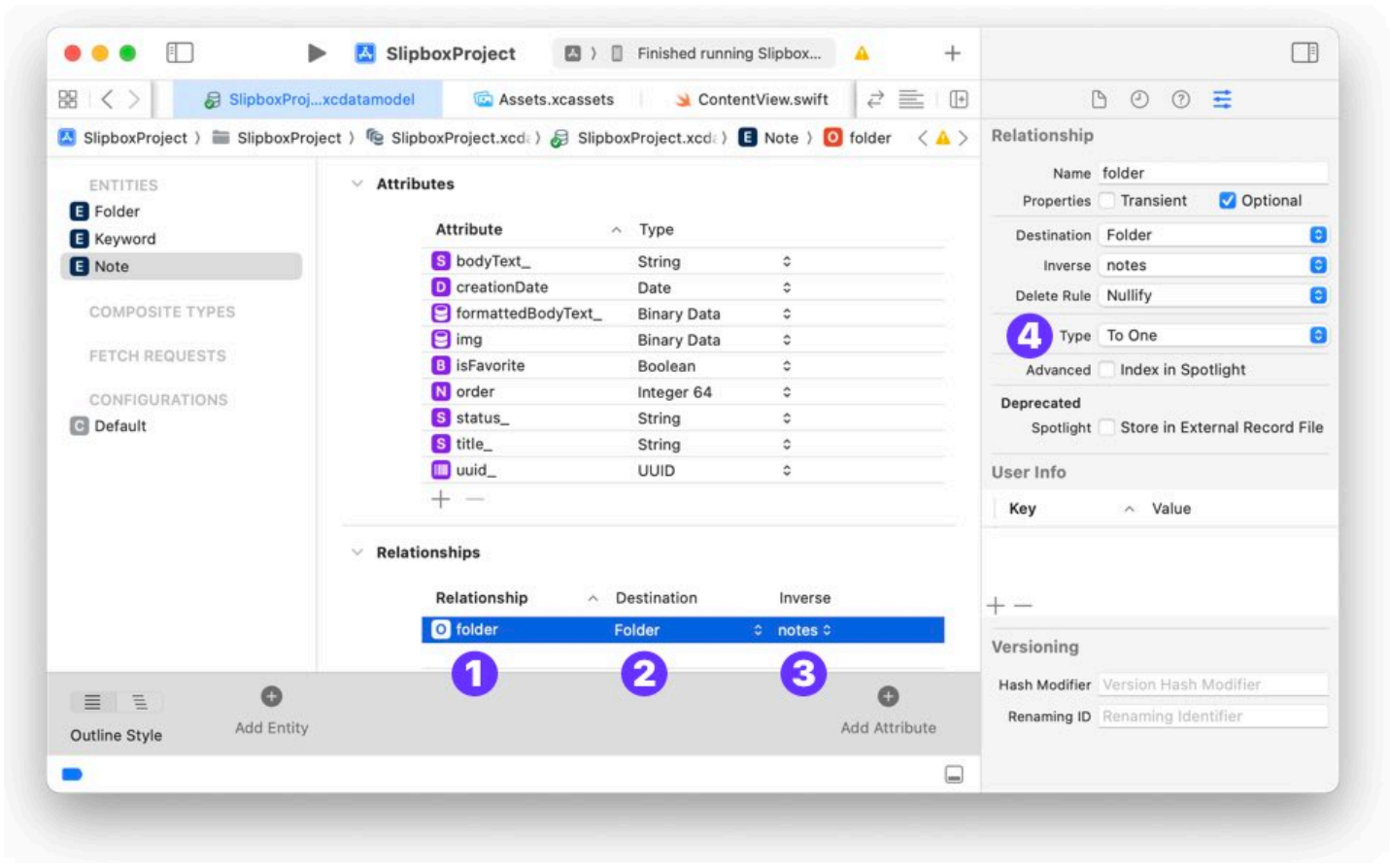
Previously, you might have only worked with attributes, but now we'll focus on the relationships section. Add a relationship by clicking the **plus button (1)**. Name this relationship **"notes_"**, which represents the collection of notes within a folder. Set the destination to "Note", because this relationship should point to the Note entity:



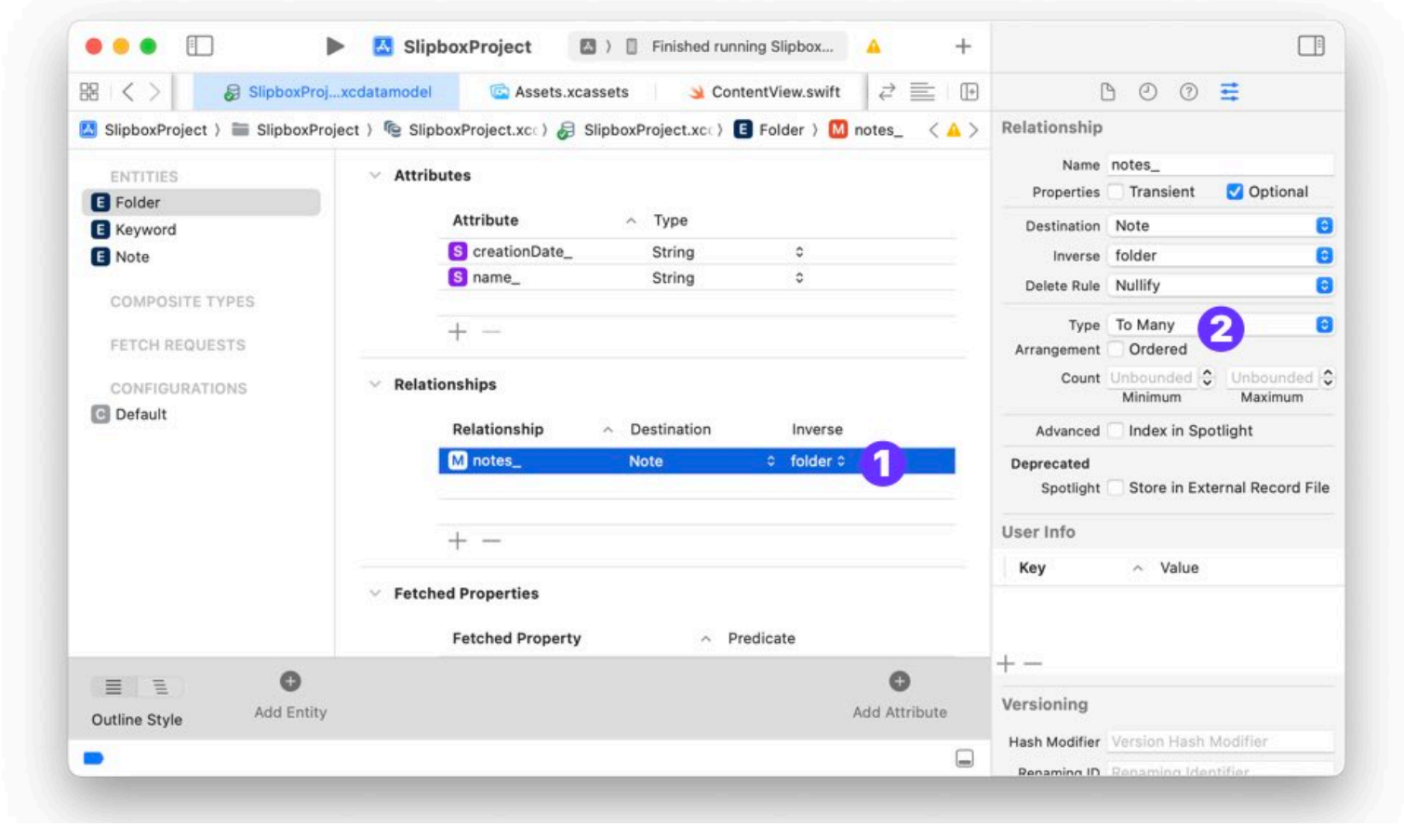
For the Note entity, add a **relationship named "folder" (1)** to signify which folder a note belongs to. Set the **destination to Folder (2).**

Core Data conveniently allows you to specify an **inverse relationship (3)**, so setting folder on a Note automatically updates the notes relationship on the Folder.

In the relationship inspector notice a field for **"Type" (4).** Here you can specify if a note should point "To One" or "To Many" folders. In this example, a note should only belong to one folder, so keep the setting as is:



Go back to the Folder entity and select the **notes relationship (1)**. Change the relationship type to "To Many". A folder can have many notes. This relationship from folder to note should point "To Many" notes:



Let’s now look at how this one-to-many relationship is set up in code:

```
extension Note {
    …
    @NSManaged public var folder: Folder?
}
```

Note has a property called “folder” that has an type of optional folder. The optional type is helpful because a note might or might not have a folder.

From the perspective of Folder we see a “notes_” property of type otpional NSSet:

```
extension Folder {
    …
    @NSManaged public var notes_: NSSet?
}
```

NSSet is an old Objective-C type, which is not soo easy to work with in Swift/SwiftUI. Additionally, the optional value is not very convenient. If I don’t have any notes added to my folder, I would rather have an empty collection. Thus, I use more syntactic sugar:

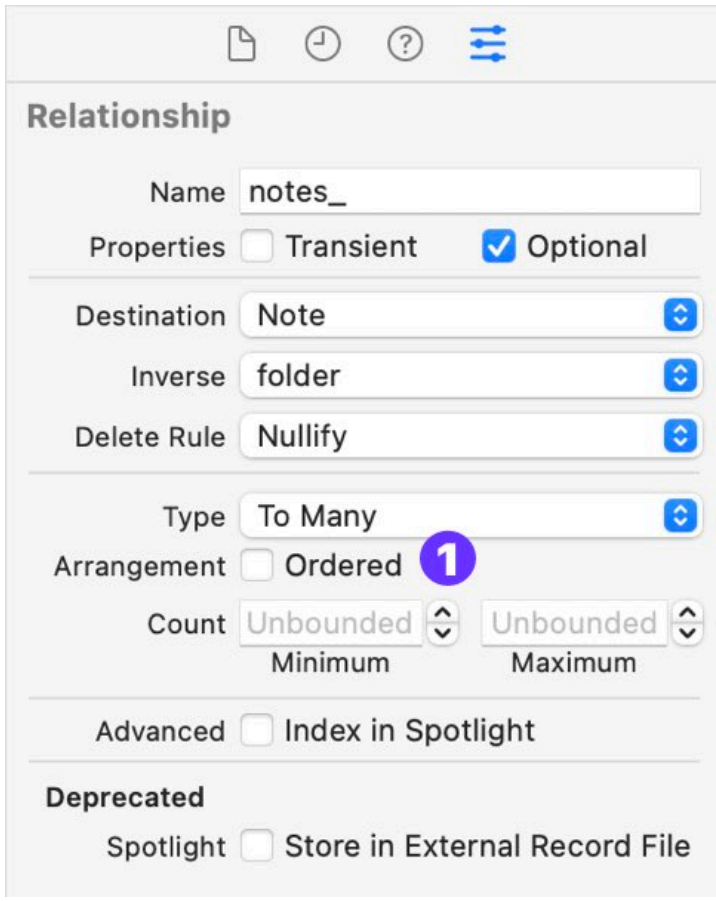
```
extension Folder {
    …

    var notes: Set<Note> {
        get { (notes_ as? Set<Note>) ?? [] }
        set { notes_ = newValue as NSSet }
    }
}
```

This is defining a computed property “notes” which is a set of Notes. If the underlying “notes_” attribute is nil, I return an empty collection.

Sorting To Many Relationships

The type of a-to-many relationship is Set, which is an unordered collection. It does not give any order of the notes in the folder. We previously added an order attribute to note that will be used for sorting later.

If you work with Core Data locally without iCloud sync you can also use another features. In the relationship inspector enable the **“Arrangment Ordered” setting (1)** :

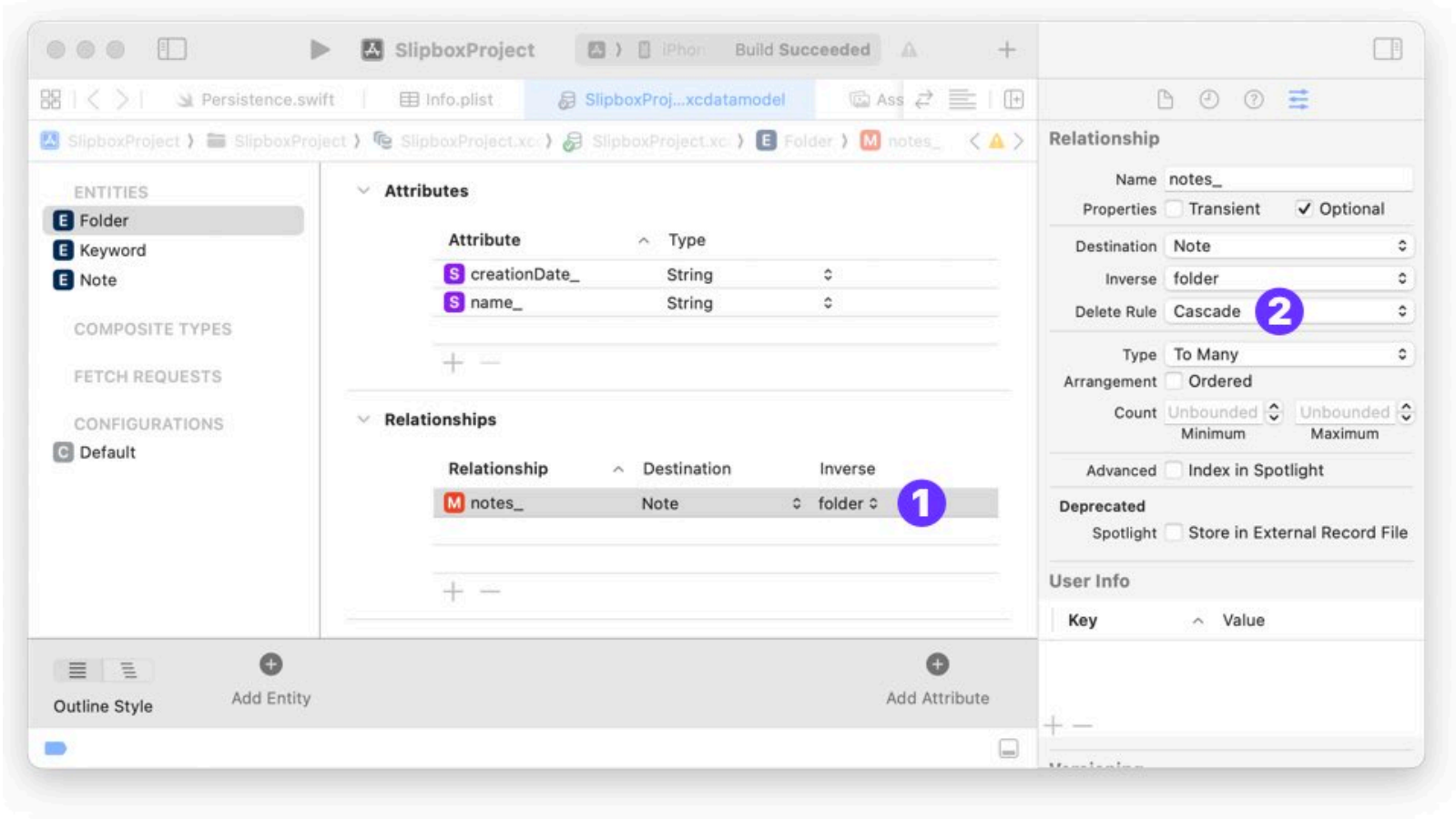


The type of “notes_” will become NSOrderedSet and manage the notes sorting for you. Since, most apps probably want to use iCloud sync, I will not enable this setting here and do the sorting manually.

Understanding Delete Rules

On the right side of the data model inspector, you'll notice properties like destination and inverse. These are more straightforward, but the **delete rule (1)** deserves special attention.

For the Folder to Notes relationship, if you delete a folder, you typically want all the notes within that folder to be deleted too. This is where the **Cascading Delete Rule (2)** comes into play. It ensures that deleting a folder cascades and deletes all associated notes:



Here is a list of delete rules available:

- **No Action:** Nothing happens. The notes in the deleted folder are unchanged and still have a relationship set. If you try to access the notes folder the app will crash. You would need to manually update the relationship.

- **Nullify:** The relationship is removed. The notes in the deleted folder are still present and their relationship to the folder is removed. If you try to access the notes folder, it will return nil. (Default Setting)

- **Cascading**: The objects that are referenced with this relationship are removed. If you delete a folder, all notes of the deleted folder are deleted as well. Use it if you have a strong connection between entities.

- **Deny:** Prevents the deletion of objects. The object will only be deleted if it has no relationship to other entities. If you try to delete a folder, it will only be removed, if no notes are connected (empty folder). This would be more useful for the Notes to Keywords relationship. You could only delete a keyword if it is not used e.g. not added to a note.

How to Add Notes to Folders

Now, let's see how this works in practice. I will create a new Unit Test class "NotesFolderRelationshipTests.swift". This is the basic code for the test class:

```
import XCTest
@testable import SlipboxProject
import CoreData

final class NotesFolderRelationshipTests: XCTestCase {

    var controller: PersistenceController!

    var context: NSManagedObjectContext {
        controller.container.viewContext
    }

    override func setUpWithError() throws {
        self.controller = PersistenceController.createEmpty()
    }

    override func tearDownWithError() throws {
        self.controller = nil
    }
}
```

To add a note to a folder, you can either set the folder property on the Note or add the note to the notes set on the Folder. Both actions are equivalent due to the inverse relationship.

```
folder.notes.insert(note)

note.folder = folder
```

You can test this with a unit test case:

```
    func test_add_note_to_folder() {

        let note = Note(title: "new note", context: context)
        let folder = Folder(name: "new folder", context: context)

        note.folder = folder // from the side of the note
        //note.folder = folder. // from the side of the folder

        XCTAssertTrue(folder.notes.contains(note))
    }
```

Removing a Note from a Folder

Similarly to adding you can remove a relationship. You can set the notes folder to nil:

```
note.folder = nil
```

Or remove the note from the folder list of notes:

```
folder.notes.remove(note)
```

Deleting Folders with a Notes Relationship

Now let's see what happens with the relationship when I delete a folder. First I will add a convenience function to my Folder extension that handles the delete:

```
extension Folder {
     …

    static func delete(_ folder: Folder) {
        guard let context = folder.managedObjectContext else { return }
        context.delete(folder)
    }
}
```

When I set the relationship, I used cascading delete rules. Thus all notes of the deleted folder are deleted as well. Here is a test function that would test the delete rule:

```
func test_delete_folder_with_notes() {
    // delete rule: cascading

    let note = Note(title: "new note", context: context)
    let folder = Folder(name: "new folder", context: context)
    note.folder = folder

    Folder.delete(folder)

    let retrievedFolders = try! context.fetch(Folder.fetch(.all))
    let retrievedNotes = try! context.fetch(Note.fetch(.all))

    XCTAssertTrue(retrievedFolders.count == 0, "should have deleted the one folder")
    XCTAssertTrue(retrievedNotes.count == 0)

}
```

I create a note and folder. Then I add the note to the folder. After, I deleted the folder, I fetch all notes and folders. Both note and folder should be deleted and my retrieved objects should all be zero.

If you would set a nullify delete rule, the folder would be deleted only. The note will still be in the database. Change the delete rule to “nullify” and test the relationship:

```
    func test_delete_folder_with_notes() {
        // delete rule: cascading

        let note = Note(title: "new note", context: context)
        let folder = Folder(name: "new folder", context: context)
        note.folder = folder

        Folder.delete(folder)

        let retrievedFolders = try! context.fetch(Folder.fetch(.all))
        let retrievedNotes = try! context.fetch(Note.fetch(.all))

        XCTAssertTrue(retrievedFolders.count == 0, "should have deleted the one folder")
        XCTAssertTrue(retrievedNotes.count == 1, “should still have a note”)
    }
```

You can also investigate the opposite relationship: What happens when I delete a note?

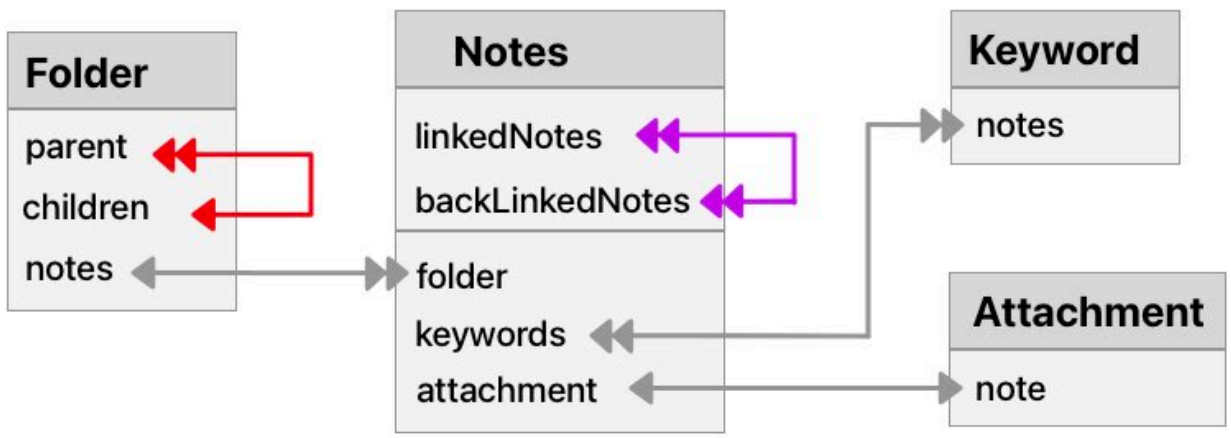
```
    func test_delete_note_in_folder() {
        // delete rule nullify

        let note = Note(title: "new note", context: context)
        let folder = Folder(name: "new folder", context: context)
        note.folder = folder

        Note.delete(note: note)

        let retrievedNotes = try! context.fetch(Note.fetch(.all))

        XCTAssertFalse(folder.notes.contains(note), “deleted note should not belong to
                                                    folder anymore”)

        XCTAssertTrue(folder.notes.count == 0, “folder should not have note,
                                               note was  deleted”)
        XCTAssertTrue(retrievedNotes.count == 0, “should have deleted the note”)
    }
```

For Note, the delete rule is “Nullify”. Thus when I delete a note, its folder is still in the database. The relationship is updated accordingly. If you try to access the folders notes, you will not find the deleted note anymore.

In this lesson, I’ve shown you how to create and manage one-to-many relationships between folders and notes in Core Data. You’ve learned about the importance of delete rules and how they help maintain the integrity of your data model. Always remember to consider the implications of each delete rule and test them thoroughly to ensure they behave as intended.

4.4 RELATIONSHIPS FOR SUBFOLDER, LINKED NOTES, AND KEYWORD TO NOTES

In this section, I'll guide you through setting up a hierarchical relationship in Core Data, specifically for folders and subfolders. This is a common scenario when you're dealing with file systems or categorization features in your apps.

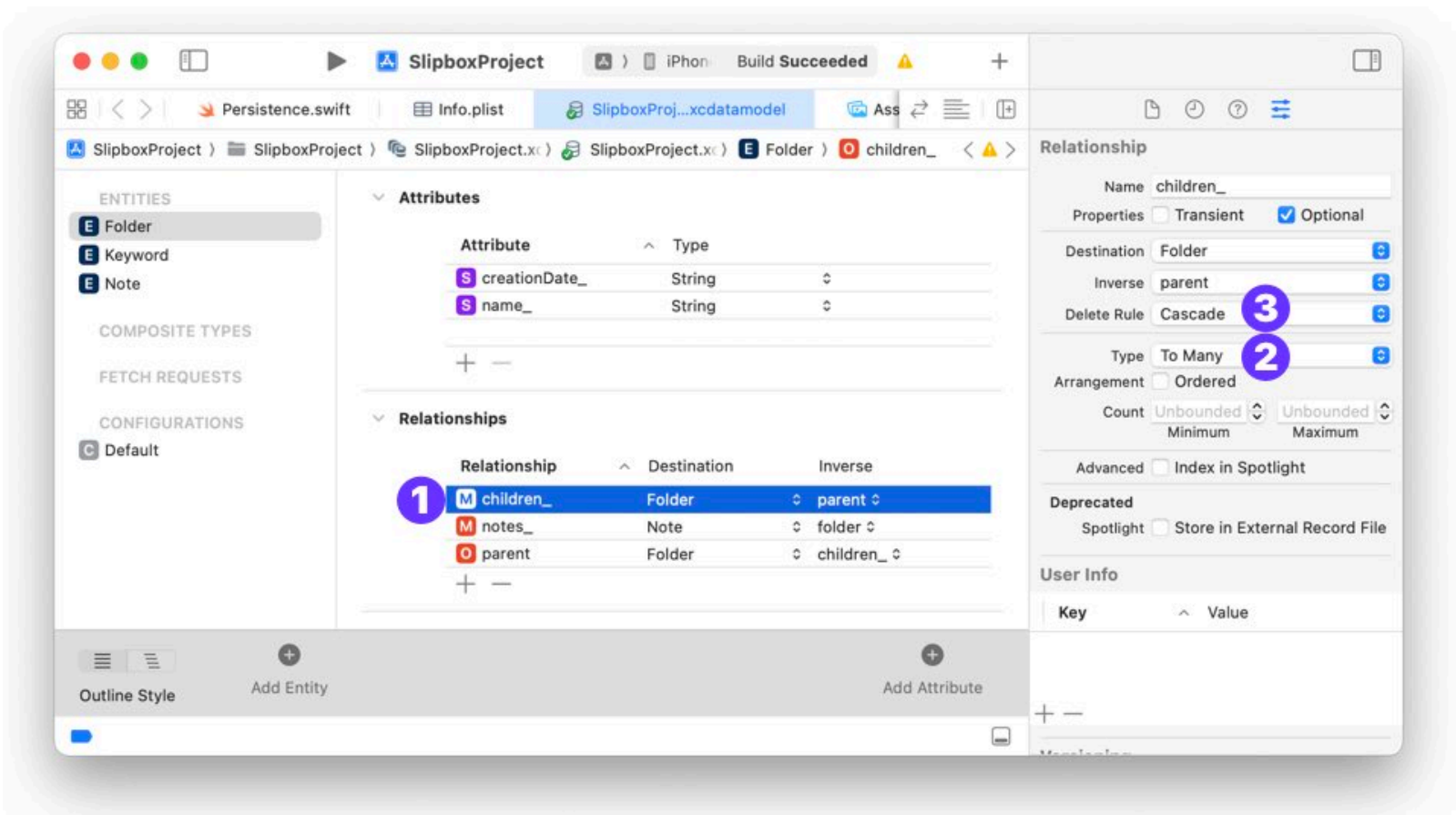


Folder Subfolder Relationship

Let's dive into the Folder entity. A folder may contain child folders, which are essentially subfolders. This sets up a relationship where a Folder entity points to other Folder entities.

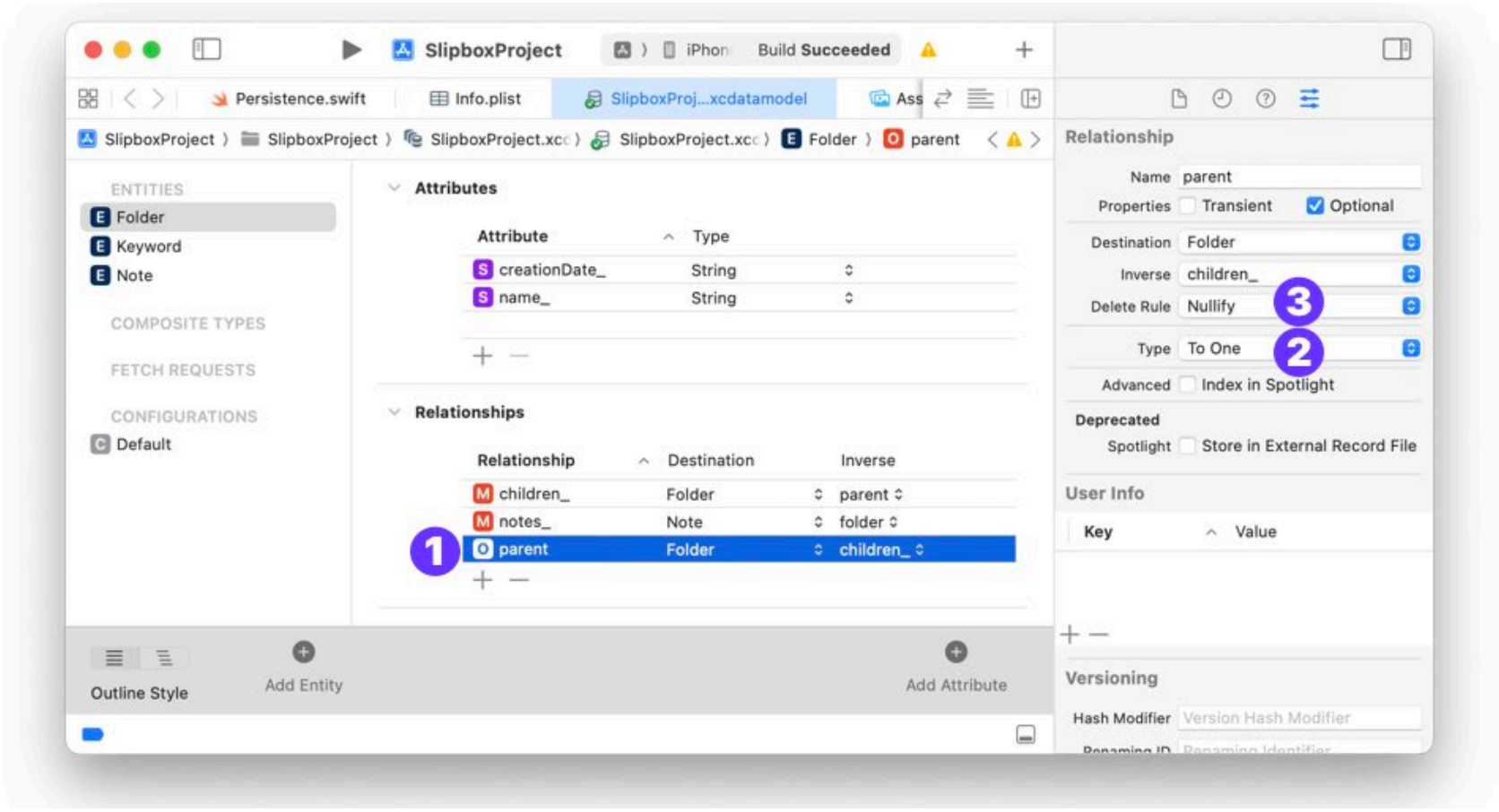
Here's how you define this relationship:

1. Name the relationship **"children_"**. The destination is Folder.
2. Since a folder can have multiple children, this is a **one-to-many** relationship.
3. For the delete rule, choose 'Cascade'. This means that when you delete a folder, all its child folders will also be deleted.



Now, let’s look at the reverse relationship: Each child folder can have one parent folder. This is a many-to-one relationship.

1. Add **a new relationship and name it “parent”** with the destination “Folder”. Set the reverse as “children"
2. Set the relationship type to “To One”. Each folder can only have one parent folder.
3. Set the delete rule to **“Nullify” (3).** Thus when a “subfolder” is deleted, its parent folder is persisted. Only the reference from f”older to subfolder” is removed



Again, I do not want to deal with NSSet, and thus created a computed property for the to-many relationship of children:

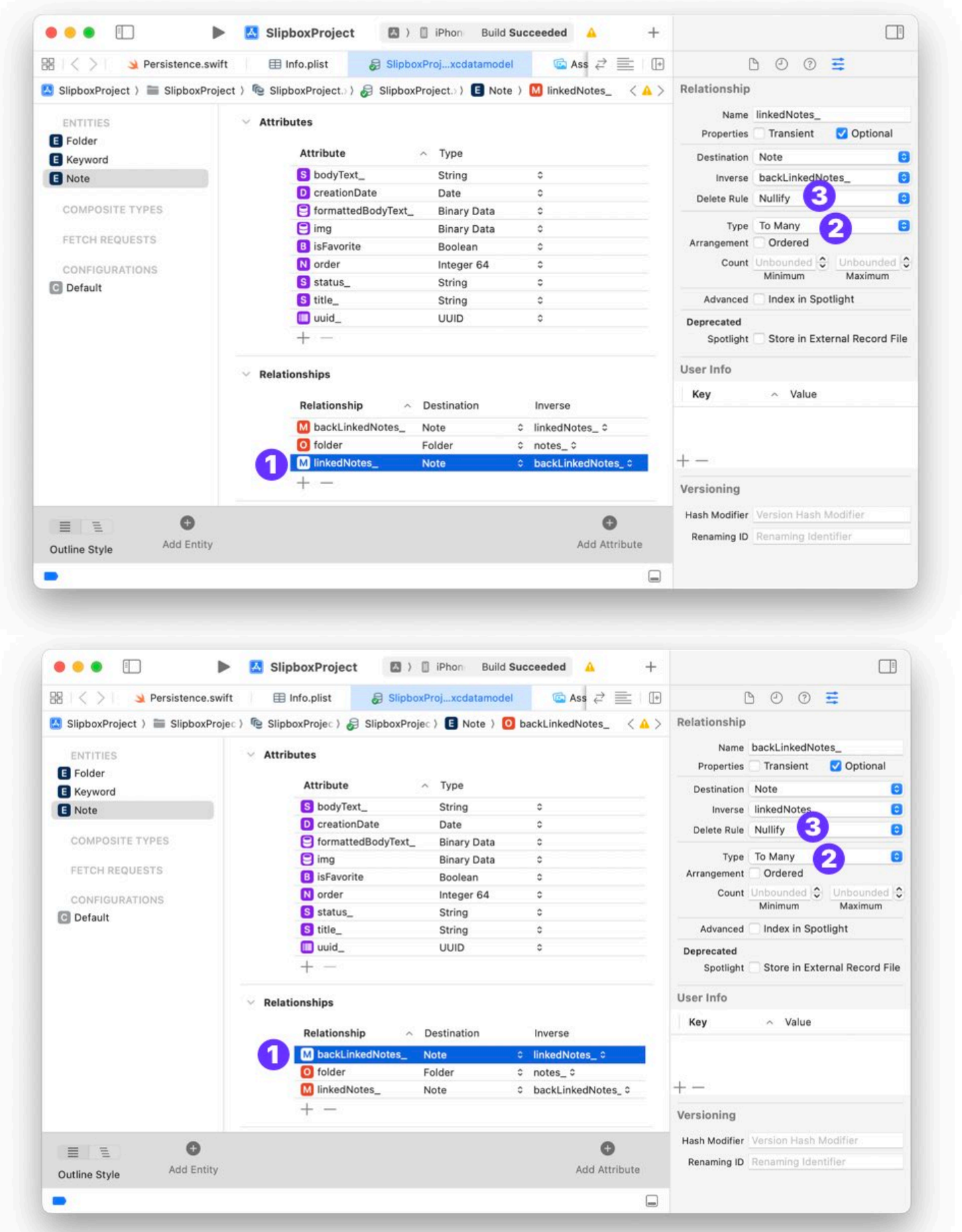
```
extension Folder {
    …

    var children: Set<Folder> {
        get { (children_ as? Set<Folder>) ?? [] }
        set { children_ = newValue as NSSet }
    }
}
```

Linking Notes with Many-To-Many Relationship

Next, let's consider another example. Suppose you want notes to point to other notes, creating a linked list of sorts. You'd set up a many-to-many relationship for this.

Here's how you do that:

1. In the Note entity, create a relationship called **"linkedNotes_"** with the destination set to Note.
2. Set the relationship type to '**To Many**'
3. Set the delete rule to **'Nullify'** because you don't want to delete other notes when one is deleted.



For the inverse direction, I want to know where this note was linked from. Here's how you do that:

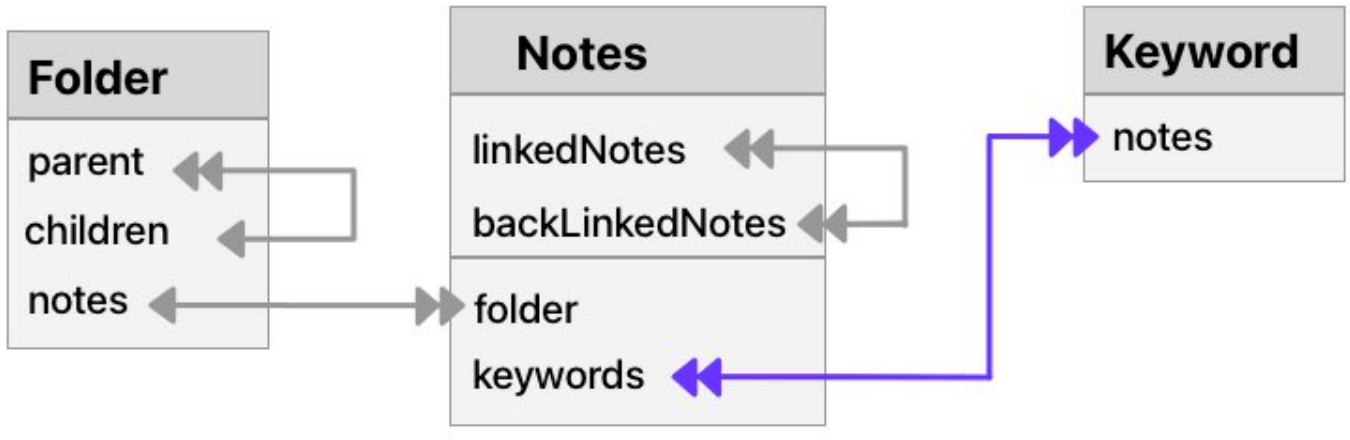
1. In the Note entity, create a relationship called **"backLinkedNotes_"** with the destination set to Note. Set the **Inverse to "linkedNotes_"**
2. Set the relationship type to '**To Many**'
3. Set the delete rule to **'Nullify'**.

Both relationships are set to 'To Many'. To work with a Swift type, I add computed properties that transform NSSet to Set:

```
extension Note {
    …

    var backLinkedNotes: Set<Note> {
        get { (backLinkedNotes_ as? Set<Note>) ?? [] }
        set { backLinkedNotes_ = newValue as NSSet }
    }

    var linkedNotes: Set<Note> {
        get { (linkedNotes_ as? Set<Note>) ?? [] }
        set { linkedNotes_ = newValue as NSSet }
    }
}
```

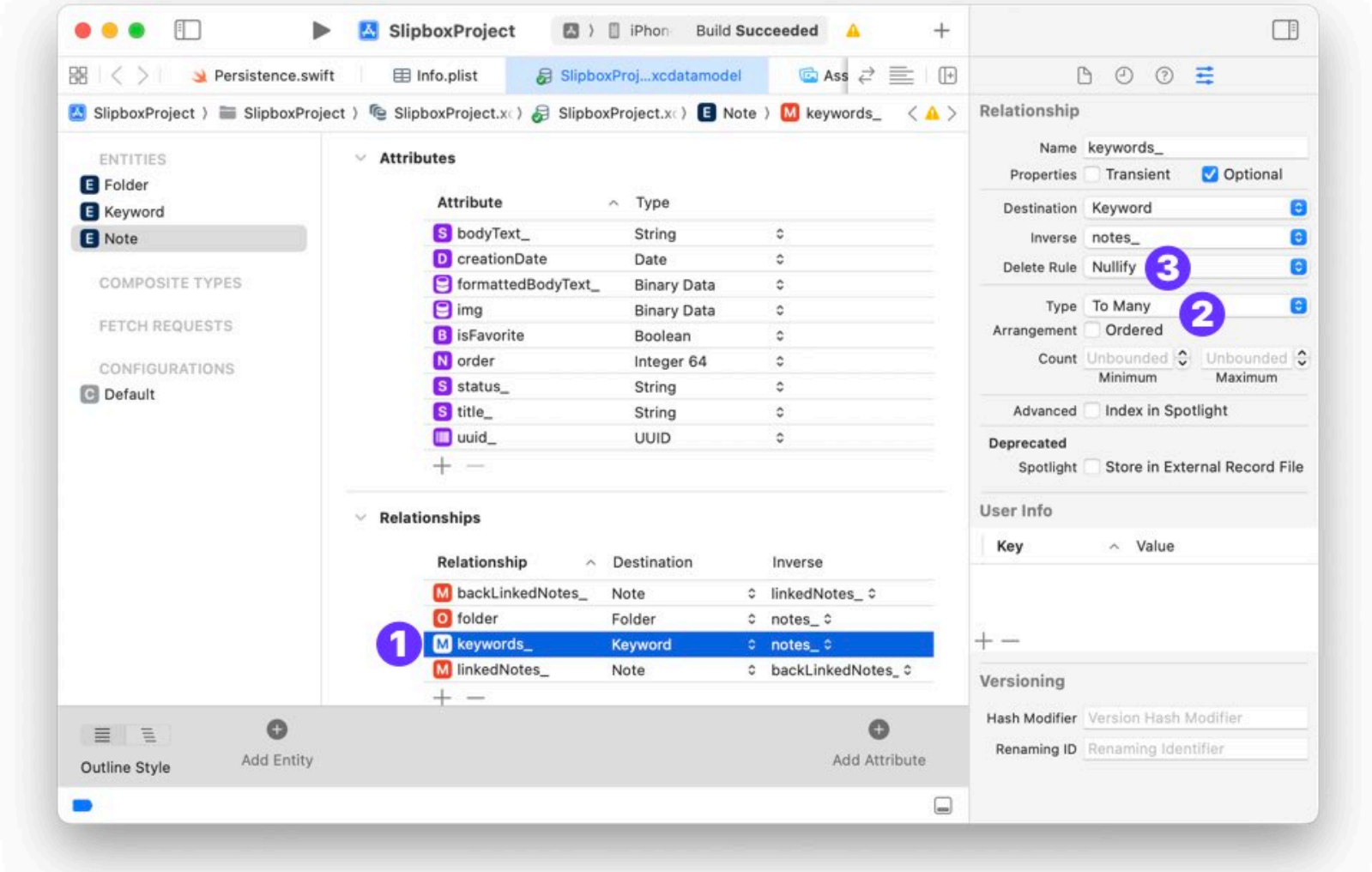
Adding Keywords to Notes for Better Organisation

Finally, let's address keywords. A Keyword entity can have many notes, and a Note can have many keywords. This is another many-to-many relationship.



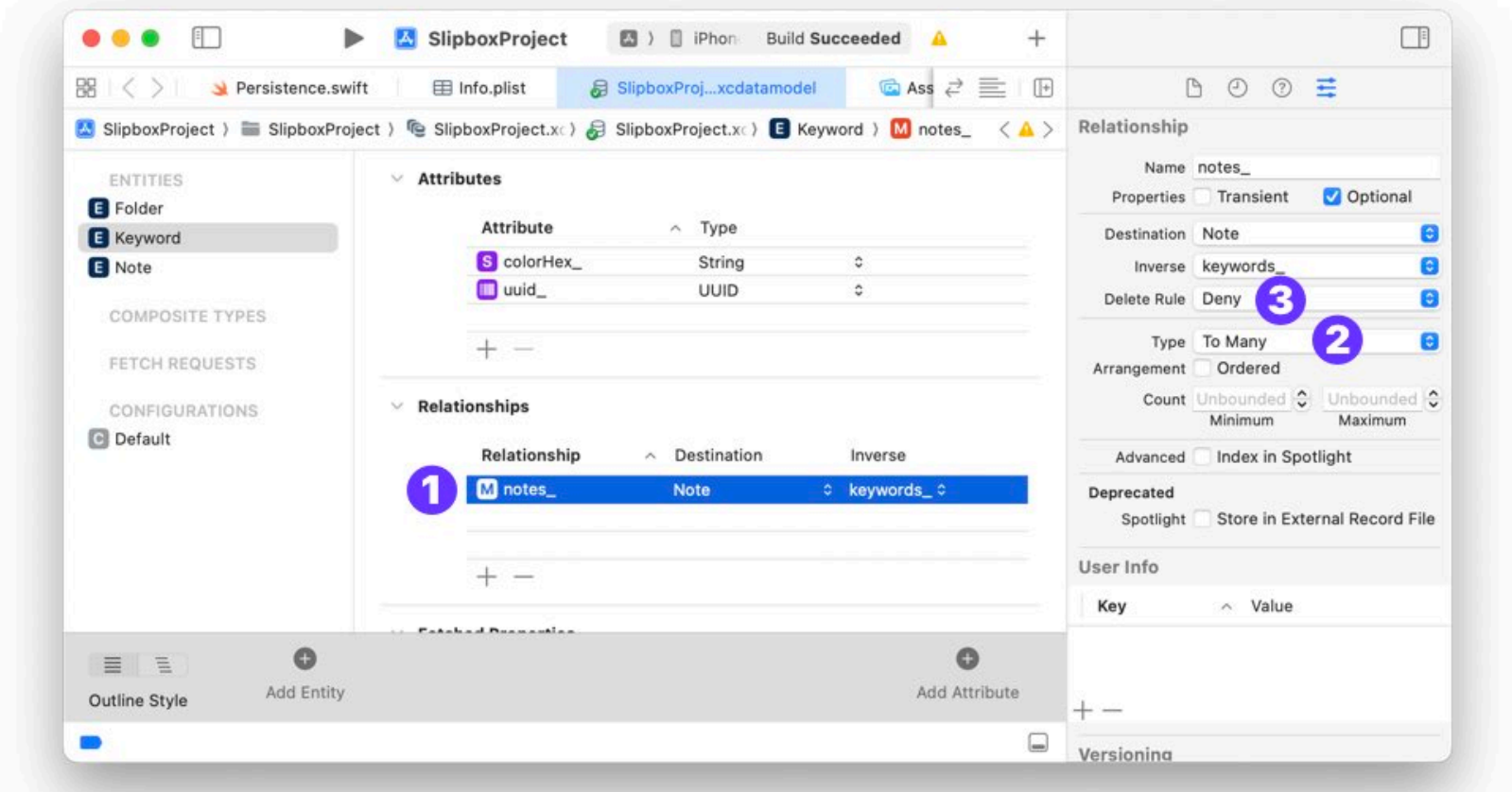
Here's how you do that:

1. In the Note entity, create a relationship called **"keywords_"** with the destination set to Keyword.
2. Set the relationship type to '**To Many**'
3. Set the delete rule to **'Nullify'** because you don't want to delete keywords when a note is deleted.



For the inverse direction, I want to see the notes that use these keywords. Here's how you do that:

1. In the Keyword entity, create a relationship called **"notes_"** with the destination set to Note. Set the **Inverse to "keywords_"**
2. Set the relationship type to '**To Many**'
3. Set the delete rule to **'Nullify'** or for more protection **'Deny'.** This means, if you try to delete a keyword that is used by one or more notes, Core Data will not delete the keyword.



I am again adding syntactic sugar to help deal with NSSet more Swifty:

```
extension Note {
    …

    var keywords: Set<Keyword> {
        get { (keywords_ as? Set<Keyword>) ?? [] }
        set { keywords_ = newValue as NSSet }
    }
}
```

```
extension Keyword {
    …

    var notes: Set<Note> {
        get { (notes_ as? Set<Note>) ?? [] }
        set { notes_ = newValue as NSSet }
    }
}
```

By following these patterns, you've created a robust way to handle relationships in Core Data. You've also made your life easier when dealing with these relationships in Swift by avoiding optional sets.

Remember, when you're working with one-to-one relationships or optional one-to-many relationships, it's okay to deal with optionals directly. But for non-optional many-to-many and one-to-many relationships, using computed properties to unwrap NSSets into Swift Sets is a clean and convenient approach.

You will see how will this approach works in the next sections when you use relationships in your SwiftUI views.

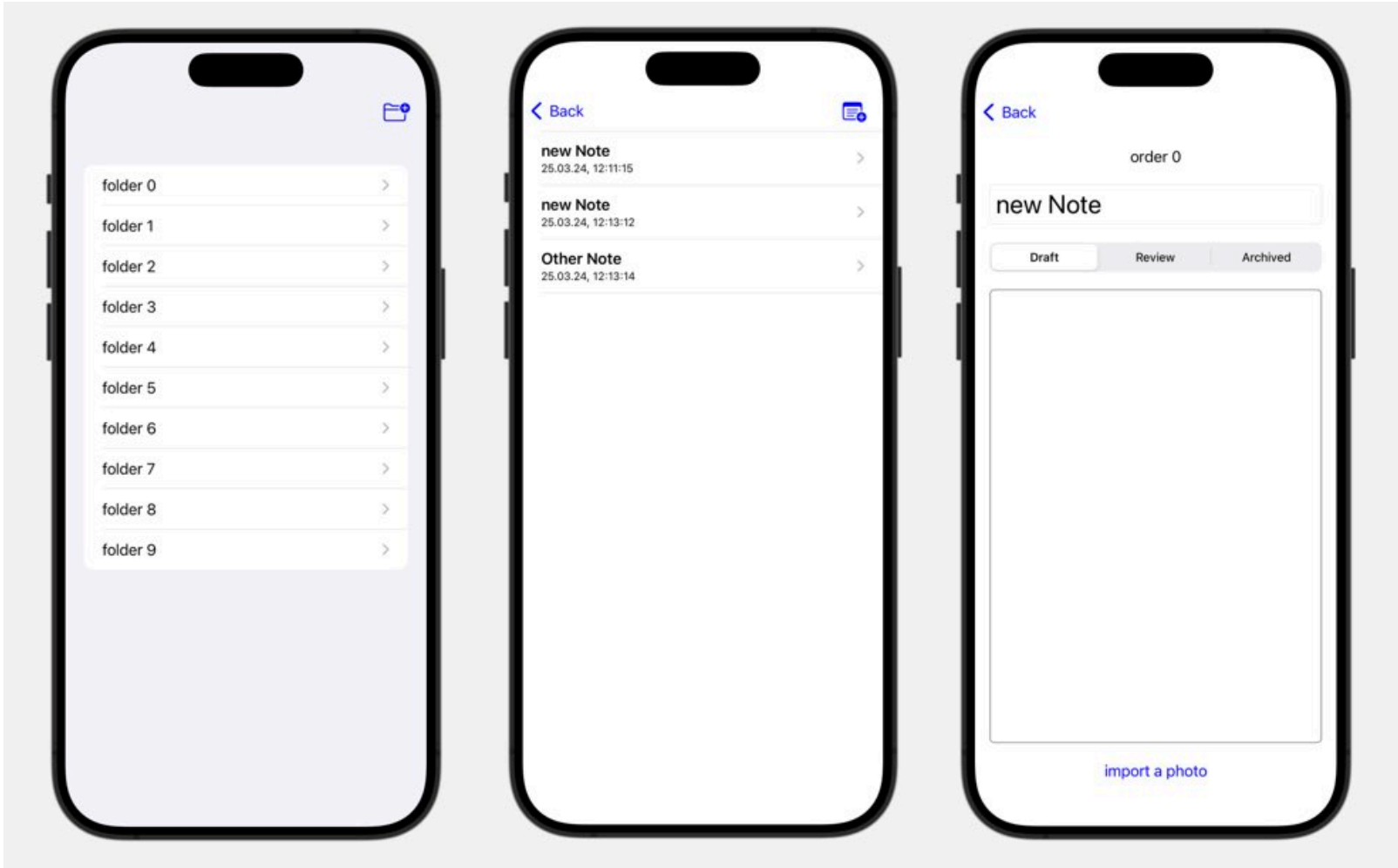
4.5 FOLDER LIST VIEW

In this section, you'll learn how to integrate the folder entities and relationships into the user interface (UI). Previously, the UI consisted of two columns: one displaying all notes and another showing selected notes. Now, I'm going to expand this to include three columns: one for folders, one for notes, and one for the details of the selected note.

To achieve the three-column layout, I'm going to use the NavigationSplitView API. This API is particularly handy because it adapts to different devices, providing a collapsible navigation view on the iPhone.

Here's how you can set up the NavigationSplitView:

```
NavigationSplitView {
   // FolderListView where you can select a folder
} content: {
   // NoteListView for the selected folder
} detail: {
    // NoteDetailView for selected note
}
```



To achieve this we need to implement the FolderListView and NoteListView. We also need to update ContentView and update the data flow from one view to the other so that e.g. the NoteListView shows the notes of the selected folder.

Showing a List of All Folders

I'm going to create a new group for folders within Xcode's project navigator. Within this group, I'll add a SwiftUI view called FolderListView. This view will display a list of all folders.

First I define a fetch request that gets all folders:

```
struct FolderListView: View {

    @Environment(\.managedObjectContext) var context
    @FetchRequest(fetchRequest: Folder.fetch(.all))
     private var folders: FetchedResults<Folder>

    @State var selectedFolder: Folder?

    var body: some View {
       …
    }
}
```

I am also defining a state property for the selected folder, that I can use for the selection in a list:

```
 List(selection: $selectedFolder) {
     ForEach(folders) { folder in
           NavigationLink(value: folder) {
               FolderRow(folder: folder)
           }
     }
}
```

For editing folder names, create a custom row view that includes a TextField for editing.

```
struct FolderRow: View {

    @ObservedObject var folder: Folder

    var body: some View {
          TextField("name", text: $folder.name)
               .textFieldStyle(.roundedBorder)
    }
}
```

Adding Folders

To enable users to add folders, I'll include a toolbar with a plus button. When tapped, this button will create a new folder entity.

```
List(selection: $selectedFolder) {
    …
}
```

```
    .toolbar {
        ToolbarItem(placement: .primaryAction) {
            Button {
                let newFolder = Folder(name: "new Folder", context: context)
                selectedFolder = newFolder // select the new folder
            } label: {
                Label("Create new folder",
                      systemImage: "folder.badge.plus")
            }
        }
    }
```

Deleting Folders

For deleting folders, you can implement swipe-to-delete functionality with **onDelete**:

```
struct FolderListView: View {
    …

    var body: some View {
        List(selection: $selectedFolder) {
            ForEach(folders) { folder in
                …
            }
            .onDelete(perform: deleteFolders(offsets:))

        }
        .toolbar {
            …
        }
    }

    private func deleteFolders(offsets: IndexSet) {
        offsets.map { folders[$0] }.forEach(Folder.delete(_:))
    }
}
```

Xcode Preview

Updating the preview to use the preview view context:

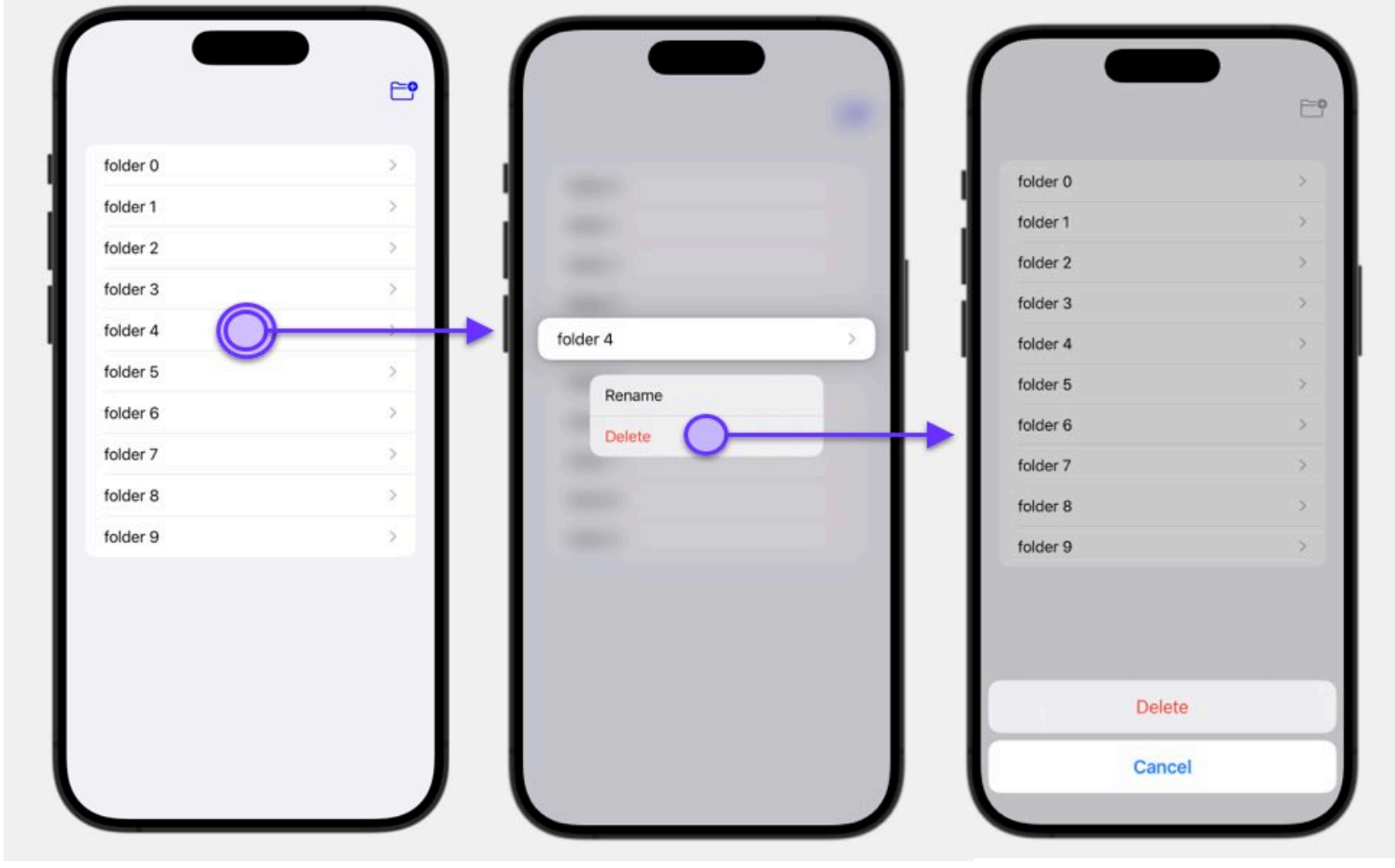
```
struct FolderListView_Previews: PreviewProvider {
    static var previews: some View {
        NavigationView {
            FolderListView(selectedFolder: .constant(nil))
            .environment(\.managedObjectContext,
                         PersistenceController.preview.container.viewContext)
        }
    }
}
```

I need to update the preview container to add example folders:

```
struct PersistenceController {

     …

    static var preview: PersistenceController = {
        let result = PersistenceController(inMemory: true)
        let viewContext = result.container.viewContext
        for index in 0..<10 {
            let newNote = Note(title: "note \(index)", context: viewContext)
            newNote.creationDate_ = Date() + TimeInterval(index)

            _ = Folder(name: "folder \(index)", context: viewContext)
        }

        return result
    }()
}
```

Context Menu to Delete Folders

You can add a context menu to the FolderRow to give more options. For example, you can show a delete button, when the user long press on the row. After tapping on “Delete”, a confirmation dialog is opened where the user can choose to either “Delete” or “Cancel”:

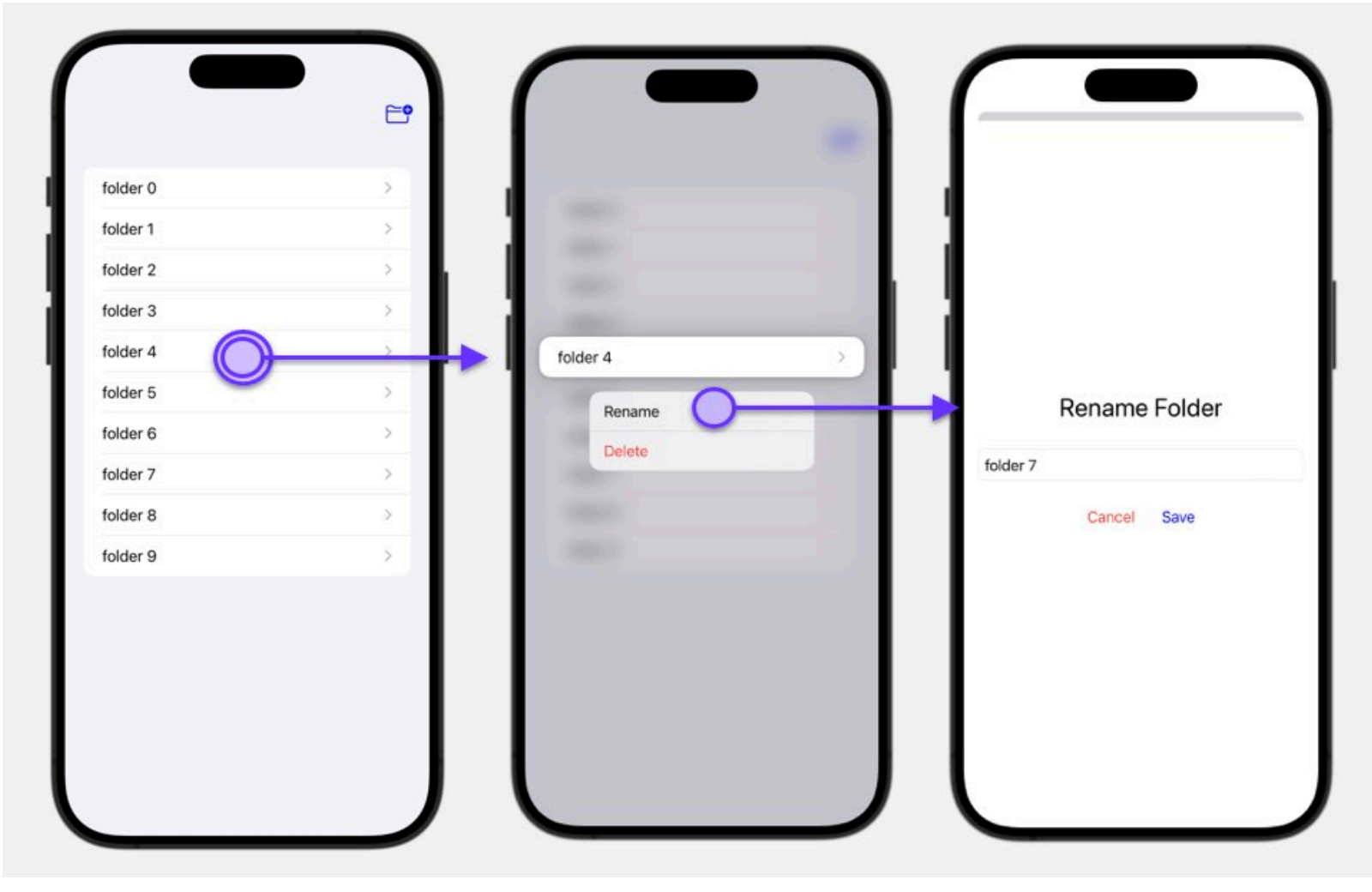


Here is the updated FolderRow that uses contextMenu and confirmationDialog:

```
struct FolderRow: View {

    @ObservedObject var folder: Folder

    @State private var showDeleteConfirmation: Bool = false

    var body: some View {
       TextField("name", text: $folder.name)
           .textFieldStyle(.roundedBorder)
           .contextMenu {
                Button(“Delete”, role: .destructive) {
                   showDeleteConfirmation = true
                }
            }
            .confirmationDialog("Delete", isPresented: $showDeleteConfirmation) {
                Button("Delete", role: .destructive) {
                    Folder.delete(folder)
                }
            }
    }
}
```

Folder Editor

If you try to tap on a folder row to enter the textfield, you will notice that it is not working with the NavigaitonLink on iOS. We will give the user an alternative to editing the folder name.



Create a FolderEditorView that gets the folder and shows a text field:

```
struct FolderEditorView: View {

    init(folder: Folder) {
        self.folder = folder
        self._name = State(initialValue: folder.name)
    }

    let folder: Folder
    @State private var name: String = ""

    @Environment(\.dismiss) var dismiss

    var body: some View {
        VStack(spacing: 30) {

            Text("Rename Folder")
                .font(.title)

            TextField("name", text: $name)
                .textFieldStyle(.roundedBorder)

            HStack(spacing: 30) {

                Button("Cancel", role: .destructive) {
                    dismiss()
                }

                Button("Save") {
                    folder.name = name
                    dismiss()
                }
            }
            .padding()
        }
```

I create a copy of the folder name in a state property “name”. Only when the user presses the “Save” button, will I set the updated name to the folder name and dismiss the editor.

Updating the FolderRow to use the rename flow:

```
struct FolderRow: View {

    @ObservedObject var folder: Folder

    @State private var showRenameEditor: Bool = false
    @State private var showDeleteConfirmation: Bool = false
    @FocusState private var textFieldIsSelected: Bool

    var body: some View {
      TextField("name", text: $folder.name)
           .contextMenu {
                Button("Rename") {
                    showRenameEditor = true
                }

                Button("Delete", role: .destructive) {
                   showDeleteConfirmation = true
                }
```

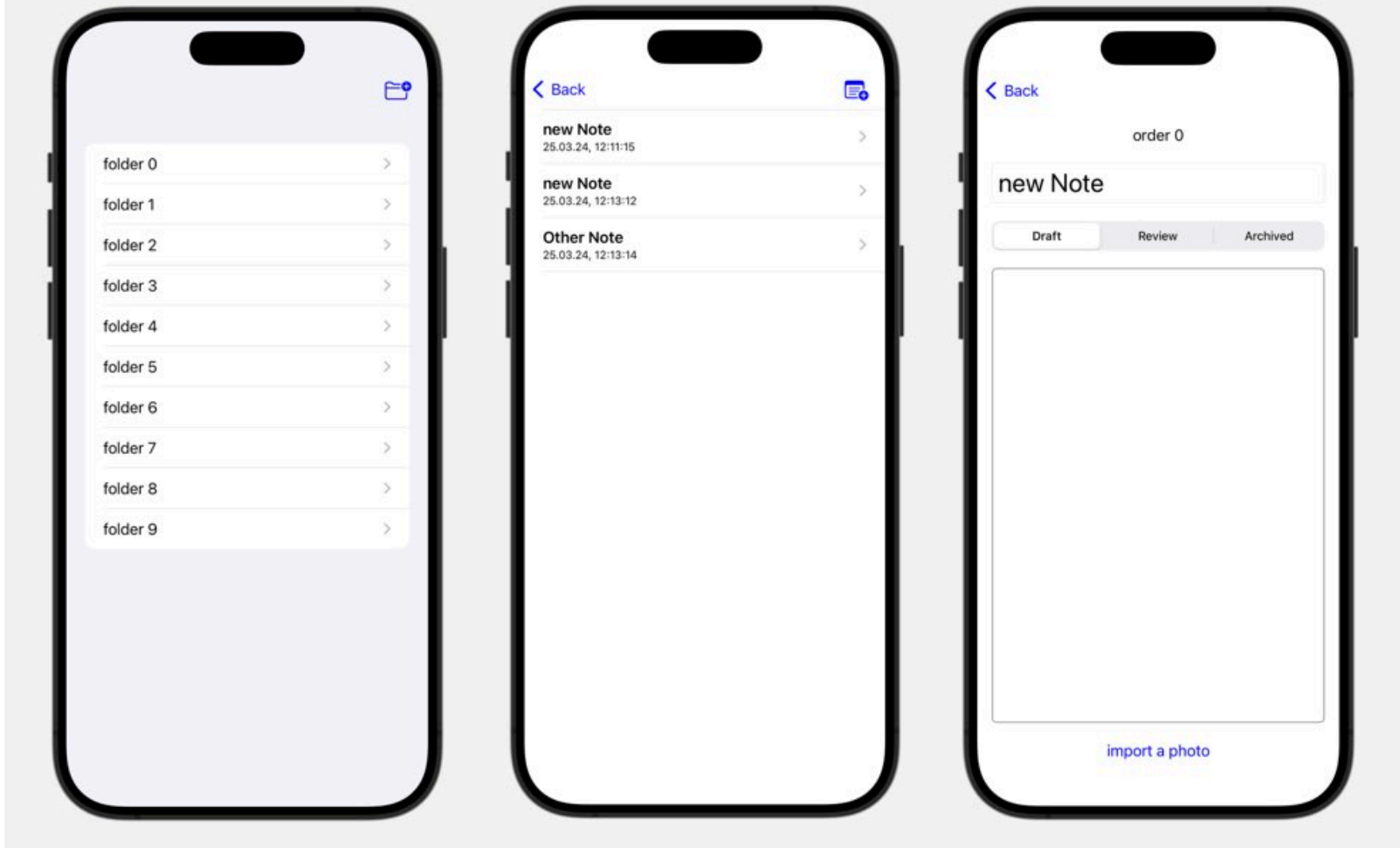
```
                }
                 …
                .sheet(isPresented: $showRenameEditor) {
                    FolderEditorView(folder: folder)
                }
        }
    }
```

4.6 NAVIGATIONSPLITVIEW

In this section, I'll be guiding you through the process of bringing together the components we've defined and organizing our project a bit more. The focus will be on the ContentView, which is the main view when we run our project.

Transition to NavigationSplitView

When you switch to an iPad and choose the landscape orientation, you'll notice that the NavigationView we've been using is no longer suitable for our needs. Instead, what you want is a three-column layout, which can be achieved by using NavigationSplitView.



Here's how you can set up a NavigationSplitView:

```
import SwiftUI
import CoreData

struct ContentView: View {

    @State private var selectedFolder: Folder? = nil
    @State private var selectedNote: Note? = nil

    var body: some View {
        NavigationSplitView {
            FolderListView(selectedFolder: $selectedFolder)
        } content: {
           // Note List View
        } detail: {
            //Note Detail View
        }
    }
}
```

I am using the FolderListView for the sidebar of NavigationSplitView. The selected property is defined in ContentView and a binding is passed to FolderListView where the user can select a folder:

```
struct FolderListView: View {
     …

    @Binding var selectedFolder: Folder?

    var body: some View {
       …
    }
}
```

For the detail of NavigationSplitView, I am showing the NoteDetailView, if the selectedNote property is not nil. Otherwise, I show a placeholder text "Select a note":

```
NavigationSplitView {
    FolderListView(selectedFolder: $selectedFolder)
} content: {
    // Note List View
} detail: {
    if let note = selectedNote {
        NoteDetailView(note: note)
    }
    else {
        Text("select a note")
            .foregroundColor(.secondary)
    }
}
```

Creating a NoteListView

I recommend creating a separate NoteListView in your Notes folder to keep things organized. This view will handle the display of notes. You'll want to pass the selected folder. I am showing the notes that belong to this folder:

```
struct NoteListView: View {

    @ObservedObject var selectedFolder: Folder
    @Binding var selectedNote: Note?

    var body: some View {
        List(selection: $selectedNote) {
            ForEach(selectedFolder.notes.sorted()) { note in
                NavigationLink(value: note) {
                    NoteRow(note: note)
                }
            }
        }
    }
}
```

The Folder notes property is a Set. ForEach requires an ordered collection. I am transforming the set of notes to:

```
selectedFolder.notes.sorted()
```

Which requires that Note conforms to 'Comparable'. I can define that the notes are sorted by creation date like so:

```
extension Note: Comparable {
    public static func < (lhs: Note, rhs: Note) -> Bool {
        lhs.creationDate < rhs.creationDate
    }
}
```

For adding and deleting notes, you'll need functions that handle these actions. In the ContentView, you can move these functions out of the NoteListView and into the main view. Make sure to pass the necessary environment objects and state properties to the NoteListView.

Handling Selection

You'll also need to manage the selection state for notes and folders. Create @State properties in your ContentView to keep track of the selected note and folder. Then, pass these as bindings to your NoteListView and FolderListView.

```
import SwiftUI
import CoreData

struct ContentView: View {

    @State private var selectedFolder: Folder? = nil
    @State private var selectedNote: Note? = nil

    var body: some View {
        NavigationSplitView(columnVisibility: $columnVisibility) {
            FolderListView(selectedFolder: $selectedFolder)
        } content: {
            if let folder = selectedFolder {
                NoteListView(selectedFolder: folder,
                             selectedNote: $selectedNote)
            } else {
                Text("select a folder")
                    .foregroundColor(.secondary)
            }
        } detail: {
            if let note = selectedNote {
                NoteDetailView(note: note)
            }
            else {
                Text("select a note")
                    .foregroundColor(.secondary)
            }
        }
    }
}
```

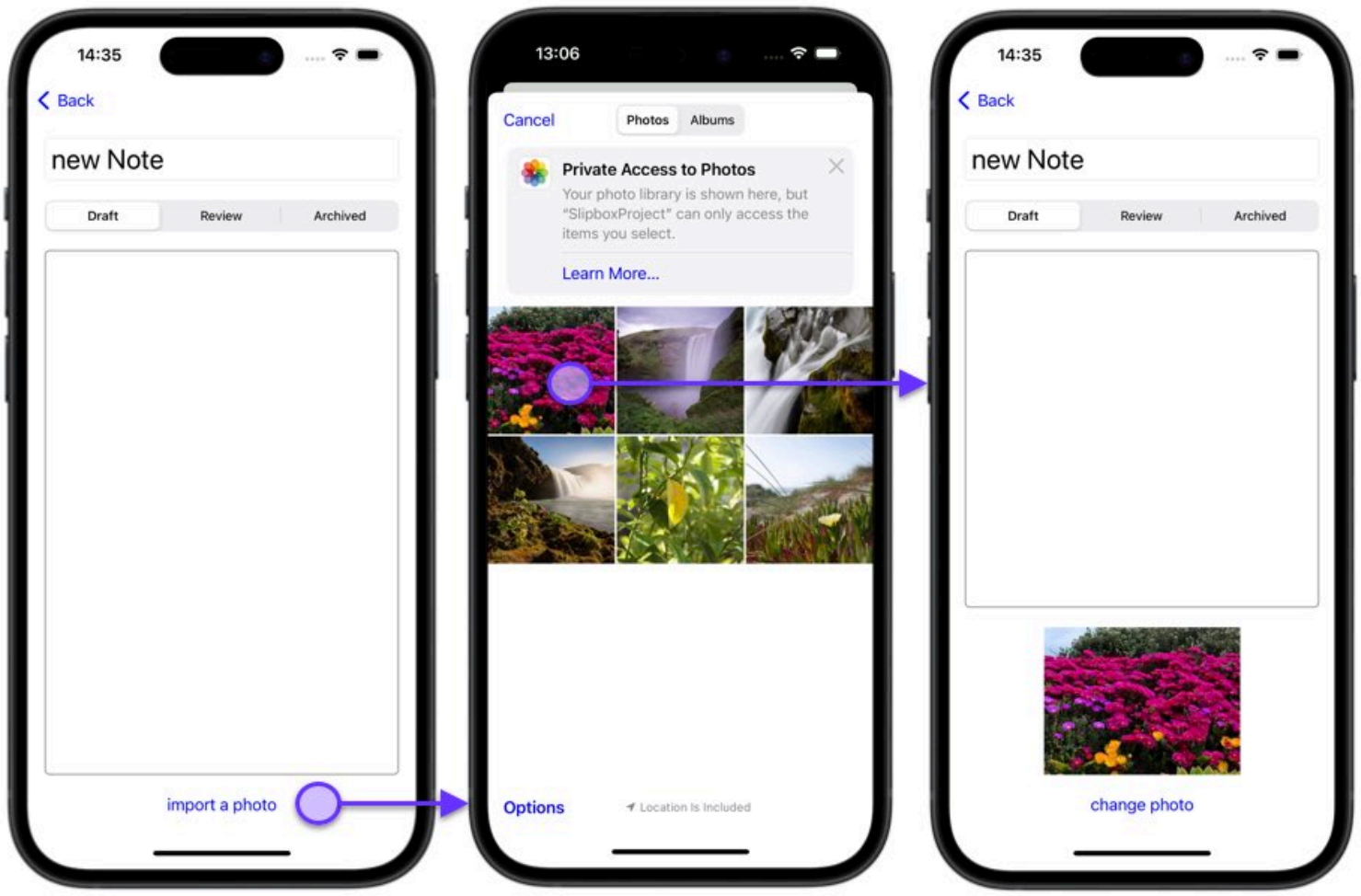
4.7 NOTE ATTACHMENT AND THUMBNAIL CREATION

In the previous lessons, I showed you how to attach images to notes in our Core Data database. However, there's room for improvement in how these images are handled. Let's dive into enhancing the user experience and performance when storing images in Core Data.

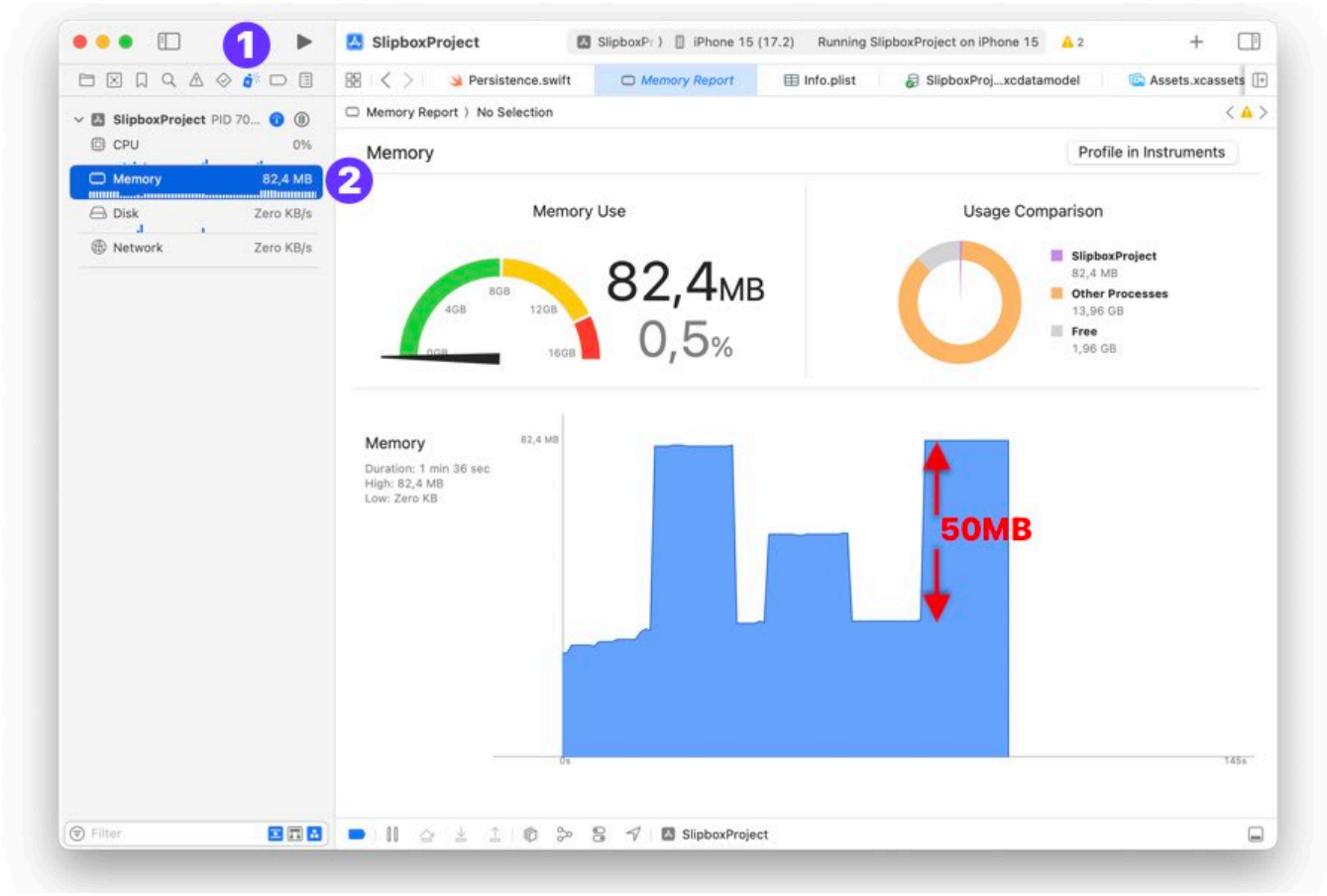
The Problem with Large Images

When you load a large image into your app, it can take a significant amount of time to process and the memory usage can increase significantly. If the image is particularly large, like the 18-megabyte image I tested with, the UI can become unresponsive. This is because the UI thread is blocked while the image is being loaded and processed. This is not acceptable behavior for a smooth user experience.

Try this out, build and run the app. Add notes and add images:



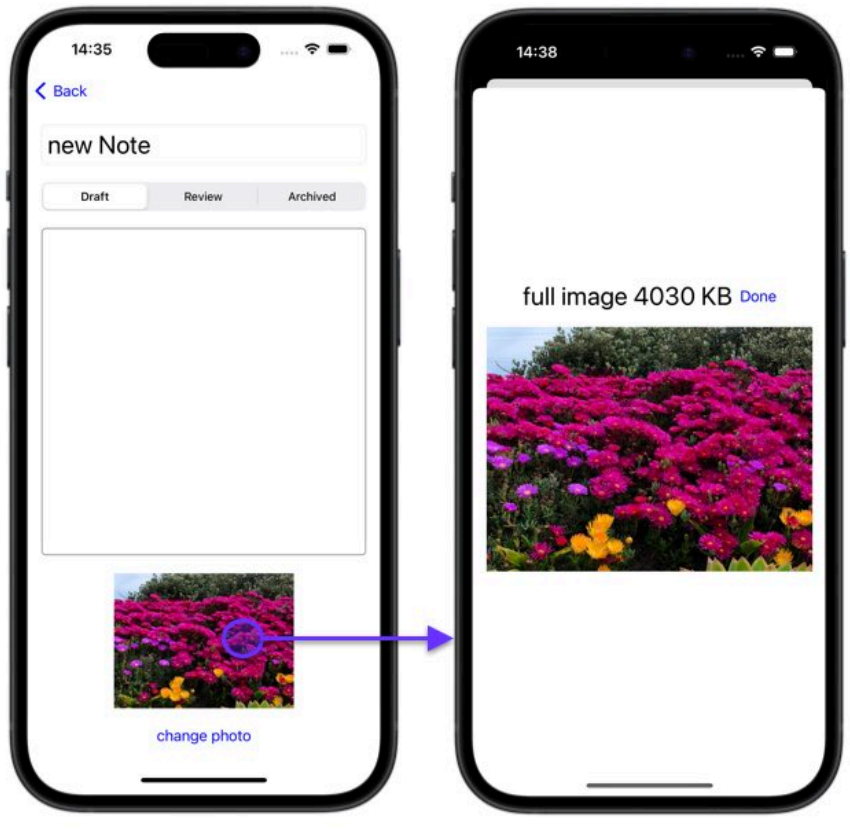
Look at Xcode in the **Debug Navigator area 1().** Have a look at the **memory graph (2):**



Initially, the memory usage is around 30MB. After I added an image to a note, it can jump by 50MB. This memory is also not directly released when you leave the NoteDetailView.

The images that I am importing have a size of 3 - 5MB. They are really large. The flower image from above is 4032x3024 pixels. Whereas the size the image takes up on screen is only ca. 1083x812. I am l**oading a 3 times larger image than I would need**.

To improve this, you wi**ll create a smaller "thumbnail" version of the images** that will be shown first. When the user double taps on the thumbnail, you will show a sheet with the full image:



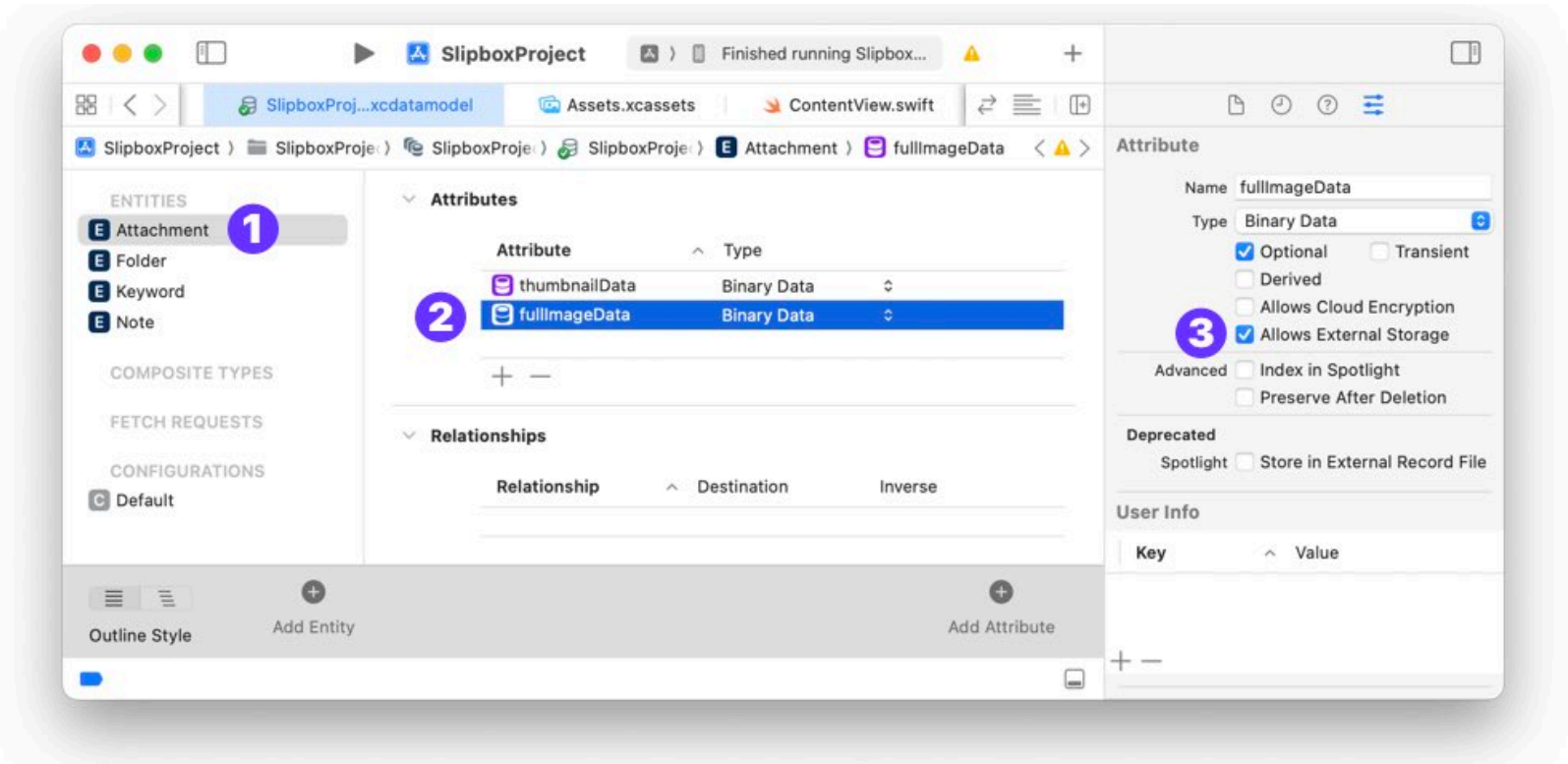
You are going to make another improvement, by **loading the images in the background,** so that the UI is not frozen for large images.

Additionally, we will store the image data not directly with the note. Every time you access the notes, like in the NoteListView, all note data is loaded into memory. This includes the large image data.

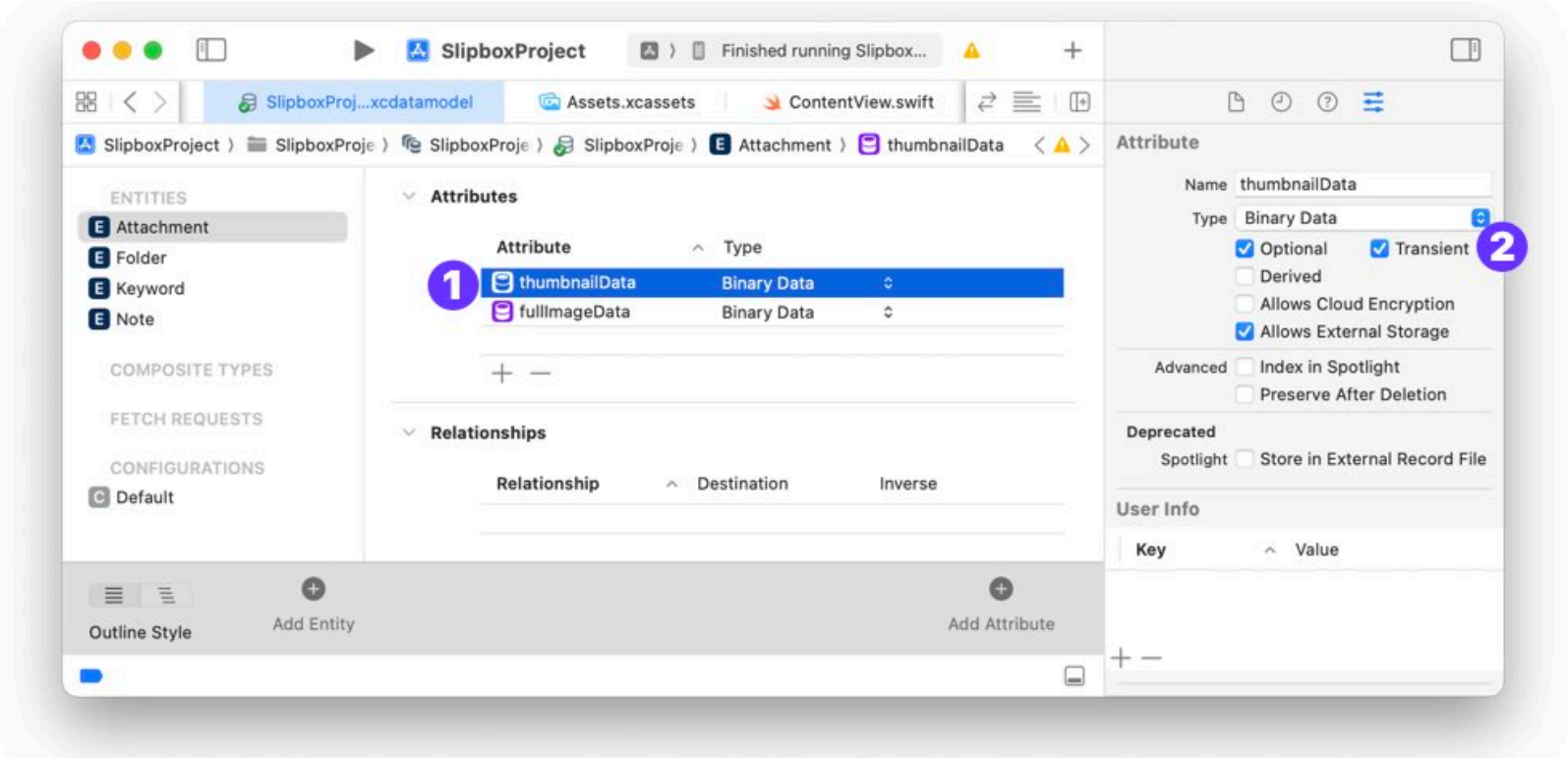
Instead, we are going to separate the **image into its separate entity "Attachment"**, which will be connected to the note via a one-to-one relationship.

Modifying the Data Model

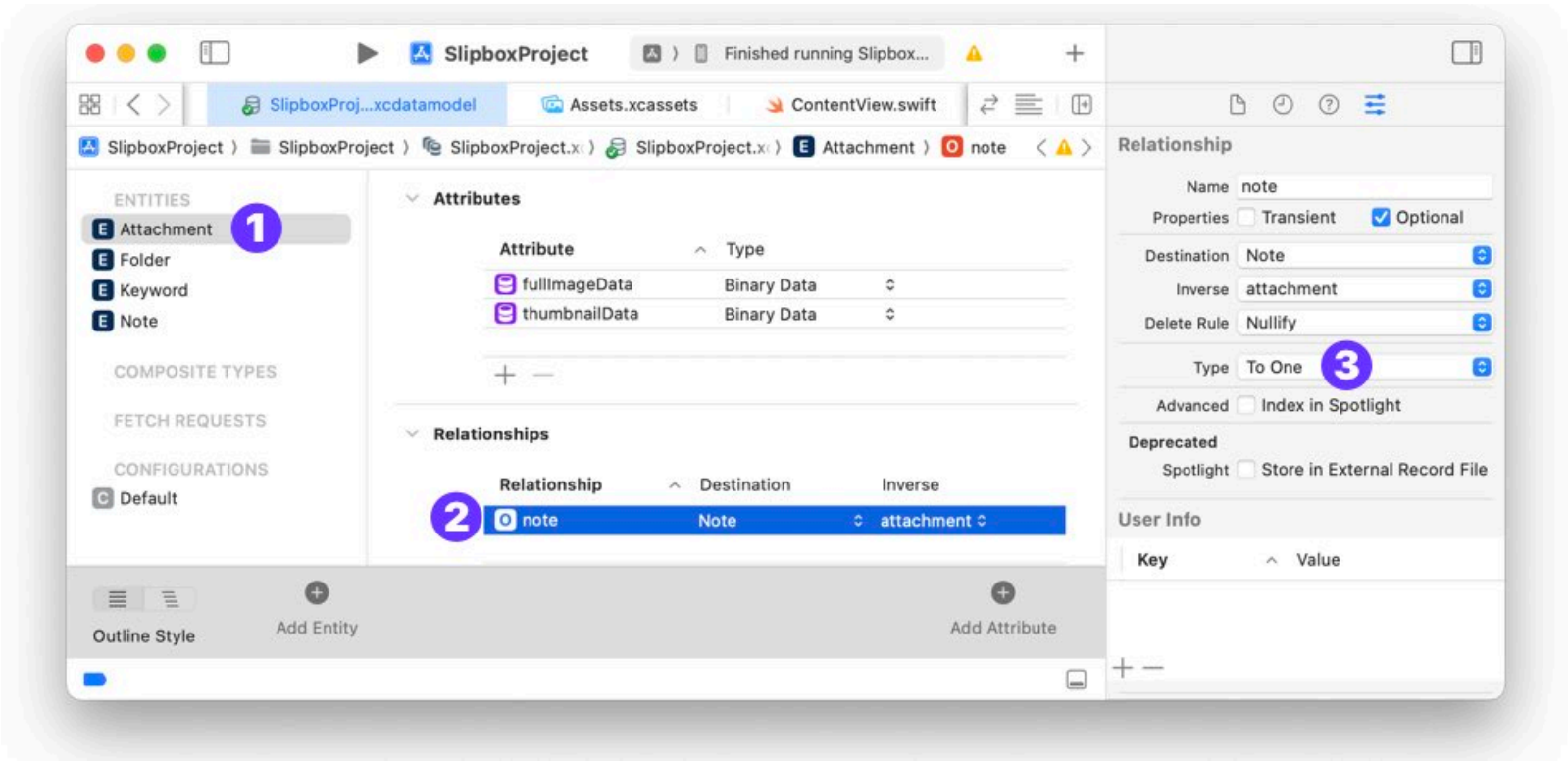
First, we'll create a new entity called **Attachment** to represent the image attachments in our notes. This entity will have two properties: **fullImageData** of type Binary Data



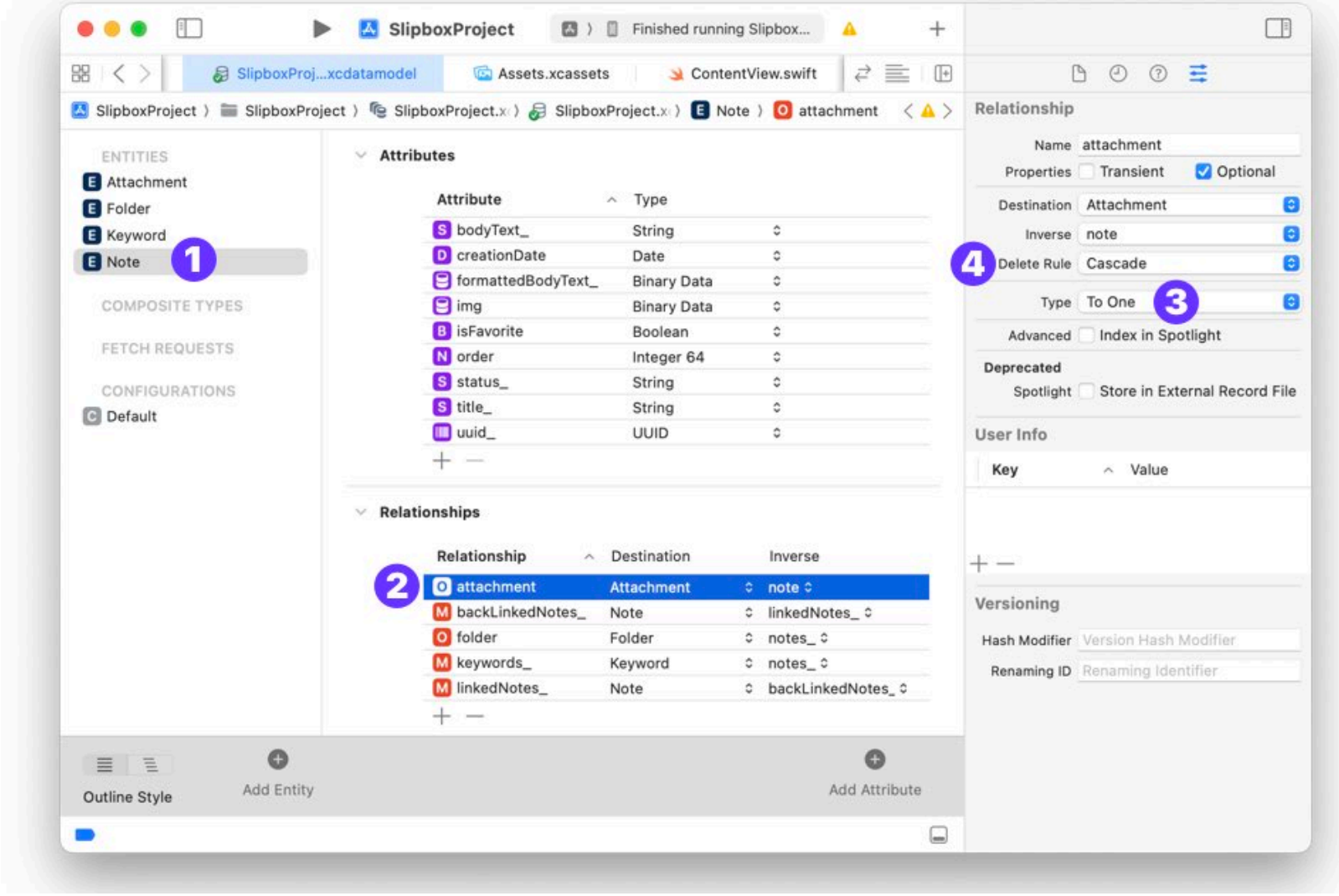
and **thumbnailData** of type Binary Data. We'll mark thumbnailData as **transient**, meaning it won't be stored permanently in the database. This allows us to regenerate the thumbnails when needed.



Next, we'll establish a one-to-one relationship between Note and Attachment. Each note can have one attachment, and each attachment belongs to one note. We'll use the "**Nullify**" delete rule, so when we delete an attachment, it doesn't delete the note but simply removes the link between them.



On the other hand, if we delete a note, we want to "**Cascade**" the deletion to the attachment, ensuring they are always deleted together:



Saving Image Data to Attachment Entity

First, I need to update the UI, when the user selects an image from the photo library. In NotePhotoSelectorButton, I create an update function:

```
struct NotePhotoSelectorButton: View {

    @Environment(\.managedObjectContext) var context
    @ObservedObject var note: Note
    @State private var selectedItem: PhotosPickerItem? = nil

    var body: some View {
        PhotosPicker(selection: $selectedItem,
                     matching: .images,
                     photoLibrary: .shared()) {
            if note.attachment == nil {
                Text("import a photo")
            } else {
                Text("change photo")
            }
        }

       .onChange(of: selectedItem) { newValue in
           Task {
               if let data = try? await newValue?.loadTransferable(type: Data.self) {
                   if let attachment = note.attachment {
                       attachment.fullImageData = data
                       attachment.thumbnailData = nil
                   } else {
                       note.attachment = Attachment(image: data, context: context)
                   }
               }
           }
       }
    }
```

I am checking if the note has already an attachment. If it does, I set the fullimageData property to the new image data and set the thumbnailData to nil.

If the note does not have an attachment, I create a new one and set it as the notes attachment.

I use a convenience initialiser to set the data to the fullImageData property:

```
extension Attachment {

    convenience init(image: Data?, context: NSManagedObjectContext) {
        self.init(context: context)
        self.fullImageData = image
    }
}
```

Creating Thumbnail Images

To create the thumbnail images, we'll add a helper function to the Attachment entity. This function, createThumbnail(fromImageData:imageSize:), takes the full image data and the desired thumbnail size as parameters. It returns an optional UIImage representing the generated thumbnail.

Here's the code for the createThumbnail function:

```
#if os(OSX)
import AppKit
#else
import UIKit
#endif
import CoreData

extension Attachment {
    static func createThumbnail(from imageData: Data,
                                 thumbnailPixelSize: Int) -> UIImage? {
        let options = [kCGImageSourceCreateThumbnailWithTransform: true,
                       kCGImageSourceCreateThumbnailFromImageAlways: true,
                       kCGImageSourceThumbnailMaxPixelSize: thumbnailPixelSize]
                      as CFDictionary

        guard let imageSource = CGImageSourceCreateWithData(imageData as CFData,
                                                            nil),
              let imageReference = CGImageSourceCreateThumbnailAtIndex(imageSource,
                                                               0, options) else {
            return nil
        }
            #if os(iOS)
            return UIImage(cgImage: imageReference)
            #else
            return UIImage(cgImage: imageReference, size: .zero)
            #
        }
}
```

This function uses the Core Graphics framework to create a thumbnail image from the given image data. We specify the maximum pixel size for the thumbnail using the kCGImageSourceThumbnailMaxPixelSize option. The resulting thumbnail image is then returned as a UIImage.

Showing the Thumbnail images

Now that we have our thumbnail creation function, we need to update our user interface to use the thumbnails instead of the full images.

Create a new view called NoteAttachmentView to display the thumbnail image. This view takes the Attachment object as a parameter and uses the getThumbnail() function to retrieve the thumbnail image.

```
struct NoteAttachmentView: View {

    @ObservedObject var attachment: Attachment
```

```
        var body: some View {
            Group {
                if let image = attachment.getThumbnail() {
                    Image(uiImage: image)
                        .resizable()
                        .scaledToFit()
                } else {
                    Color.gray
                }
            }
        }
    }
```

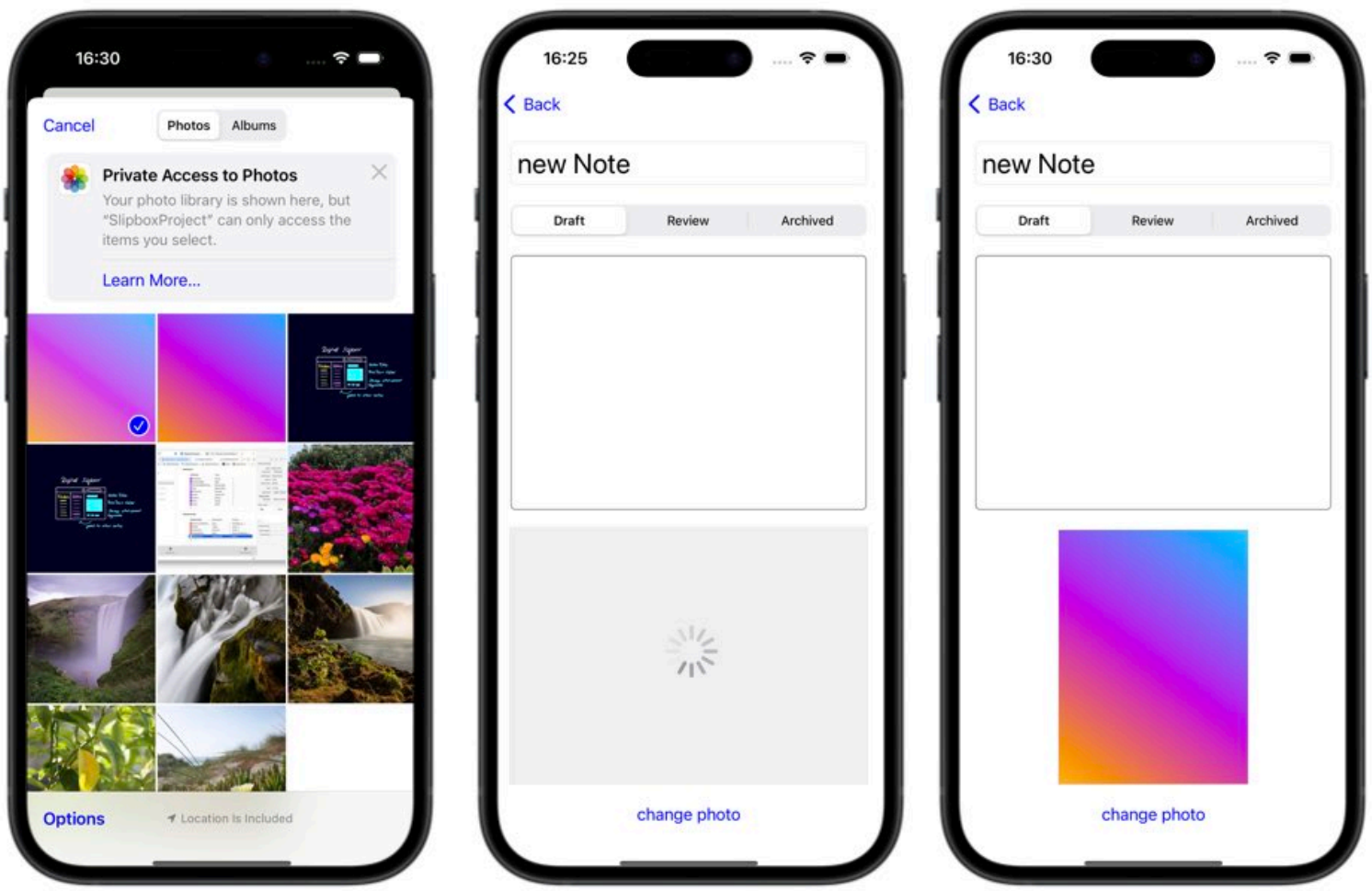
The code to retrieve the thumbnail belongs to the Attachment.:

```
    extension Attachment {
        …

        func getThumbnail() async -> UIImage? {

            //1. check if attachment has thumbnail data, this is transient and will
            //not be persisted
            guard thumbnailData == nil else {
                //2. We have data and use it to generate an image
                 //Stop here because we have used a recently generated thumbnail
                return UIImage(data: thumbnailData!)
            }

             //3. Don't have thumbnail data, start by checking the fullImageData
             guard let fullImageData = fullImageData else {
                 return nil
             }

             //4. generate the thumbnail data from the full image data
             let newThumbnail = Attachment.createThumbnail(from: fullImageData,
                                                           thumbnailPixelSize: 200)

            //5. We have the UIImage and convert it to data
            // which is stored in the transient thumbnailData property
            #if os(iOS)
             self.thumbnailData = newThumbnail?.pngData()
            #else
             self.thumbnailData = newThumbnail?.tiffRepresentation
            #endif

             //6. We have a new thumbnail and return it here
             return newThumbnail
        }

        // This is a convenient function that creates an UIImage from data in the
        // background with async await
        static func createImage(from imageData: Data) async -> UIImage? {
            let image = await Task(priority: .background) {
                UIImage(data: imageData)
            }.value
            return image
        }
    }
```

This code looks a bit long. But I am doing some optimization, because I only want to create a thumbnail if I don't already saved it before (1).

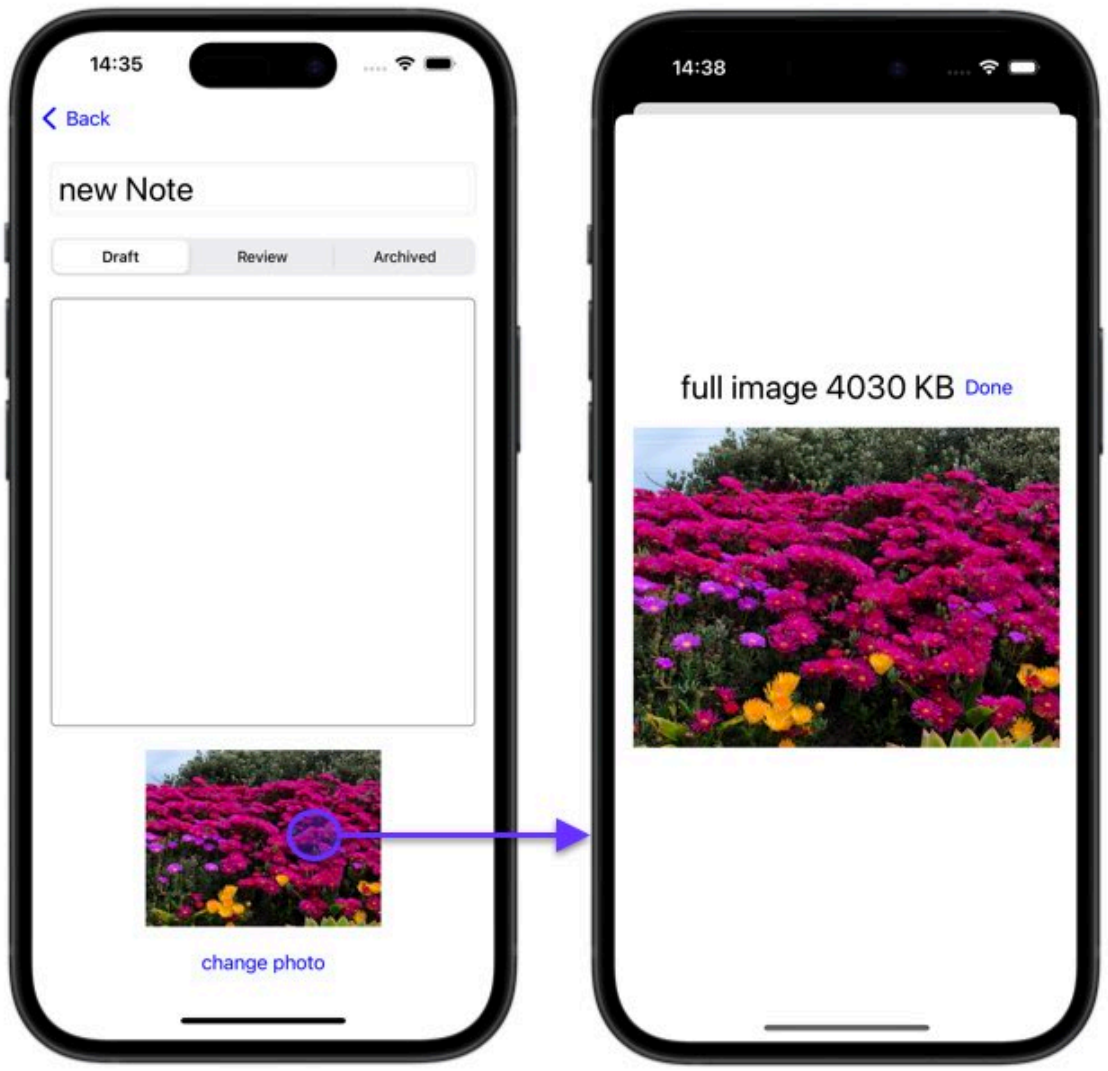
Now that we have our thumbnail creation function and the NoteAttachmentView, we need to update our user interface to use the thumbnails instead of the full images. Let's make the necessary changes in the NoteDetail view:

```
struct NoteDetailView: View {

    @ObservedObject var note: Note

    var body: some View {
        VStack(spacing: 20) {
            …
            //OptionalImageView(data: note.img)
            if let attachment = note.attachment {
                NoteAttachmentView(attachment: attachment)
            }
        }
    }
}
```



Showing The Full Image

We'll also add a double-tap gesture recognizer to show the full-scale image in a separate view.



First, create a new view "FullImageView" which shows the full image of the attachment:

```
struct FullImageView: View {

    let attachment: Attachment
    let title: String

    @State private var image: UIImage? = nil
    @Environment(\.dismiss) var dismiss

    var body: some View {
        VStack {
            HStack {
                Text(title)
                    .font(.title)
                Button("Done") {
                    dismiss()
                }
            }

            if let data = attachment.fullImageData,
               let image = UIImage(data: data) {
                Image(uiImage: image)
                    .resizable()
                    .scaledToFit()
            }
```

```
            }
            .padding()
        }
    }
```

In NoteAttachmentView, I add a double-tag gesture to the thumbnail that opens the sheet:

```
    struct NoteAttachmentView: View {

        @ObservedObject var attachment: Attachment

        @State private var showFullImage: Bool = false
        @State private var thumbnailImage: UIImage? = nil

        var body: some View {
            Group {
                if let image = thumbnailImage {
                    Image(uiImage: image)
                        .resizable()
                        .scaledToFit()
                        .gesture(TapGesture(count: 2).onEnded({ _ in
                            showFullImage.toggle()
                        }))

                        .sheet(isPresented: $showFullImage) {
                            FullImageView(attachment: attachment,
                                          title: "full image \(dataSize(data:
                                                 attachment.fullImageData)) KB")
                        }
                } else {
                    Color.gray
                }
            }
        }
    }
```

You can also calculate the size of the image in KB:

```
    func dataSize(data: Data?) -> Int {
        if let data = data {
            return data.count / 1024
        } else {
            return 0
        }
    }
```

And show this as the title of the sheet:

```
    FullImageView(attachment: attachment,
                  title: "full image \(dataSize(data: attachment.fullImageData)) KB”)
```

4.8 CORE DATA OBJECT IN BACKGROUND TASK

When working with Core Data in a SwiftUI application, managing data on background threads can lead to unexpected issues if not done correctly. I want to highlight a common problem you might encounter and how to solve it.

In the following code ,you're executing a task in the background where you're creating a new Attachment object with the context running on the main queue. You might have experienced problems where the object identifier isn't set properly, or the insert functions aren't called as expected:

```
struct NotePhotoSelectorButton: View {
    …
   @Environment(\.managedObjectContext) var context // main queue

    var body: some View {
       PhotosPicker(…)
       .onChange(of: selectedItem) { newValue in
           Task {
               if let data = try? await newValue?.loadTransferable(type: Data.self) {
                  // executed in background
                   if let attachment = note.attachment {
                       attachment.fullImageData = data
                       attachment.thumbnailData = nil
                   } else {
                       note.attachment = Attachment(image: data, context: context)
                   }
               }
           }
    }
```

Core Data objects are thread-safe and should only be used on the thread they are created on. The solution is to ensure that updates happen on the main queue, or more specifically, on the main actor. The main actor is a relatively new concept in Swift that ensures your UI updates and Core Data manipulations run on the main thread.

Extract a function that handles the update:

```
struct NotePhotoSelectorButton: View {
    …
   @Environment(\.managedObjectContext) var context // main queue

    var body: some View {
       PhotosPicker(…)
       .onChange(of: selectedItem) { newValue in
           Task {
               if let data = try? await newValue?.loadTransferable(type: Data.self) {
                  update(data: data)
               }
           }
    }

    func update(data: Data) {
        if let attachment = note.attachment {
            attachment.fullImageData = data
            attachment.thumbnailData = nil
```

```
            } else {
                note.attachment = Attachment(image: data, context: context)
            }
        }
    }
```

Now mark the update function to run on the main actor:

```
    @MainActor
        func update(data: Data) {
            …
    }
```

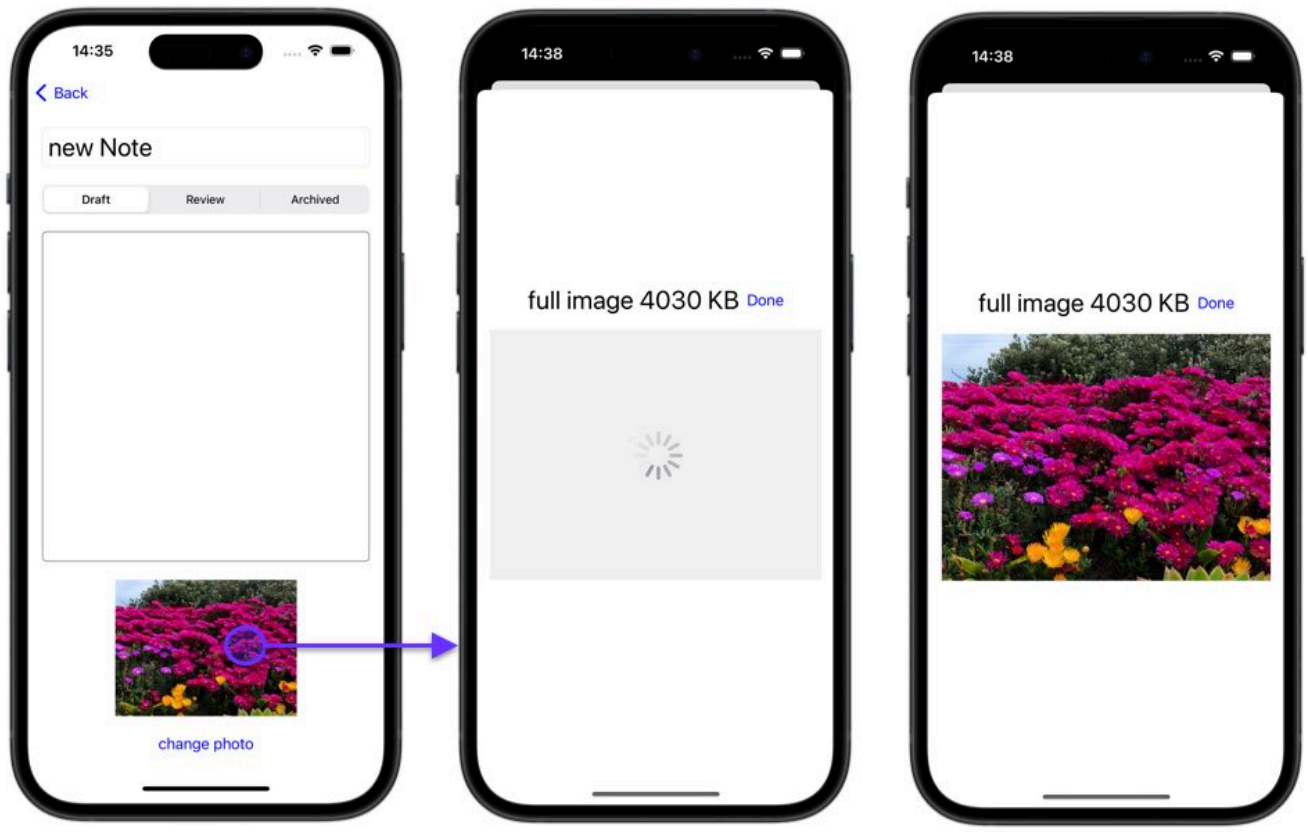
With this @MainActor annotation, you’re telling Swift that everything inside updateData should be executed on the main thread.

You can run Core Data objects in the background. You would need to create a new background context. Fetch the objects in this context and do the work in the background. However, I found it hard to get the changes propagated to the main thread and reflected in the UI.

4.9 BACKGROUND PROCESSING OF IMAGES WITH ASYNC AND TASK

When you're dealing with images in your app, especially large ones, it's crucial to ensure that the UI remains responsive. If you've ever tapped on an image thumbnail expecting a quick load, only to be met with a lagging screen, you know how frustrating it can be. To enhance your app's performance, you should load images in the background.

Imagine you have a thumbnail, and when you tap it, you expect a full-screen image to appear promptly. But what if it takes three seconds for the overlay to load? That's not ideal. Instead, you want the image to pop up instantly, with an activity indicator. If the user doesn't want to wait, they can hit 'Done' to close the sheet.



Let's dive into how to implement this improved functionality.

Loading the Full Image in the Background

In the NoteAttachmentView, which displays the thumbnail, you might open a sheet to show the full image when the thumbnail is double-tapped. The delay you're experiencing comes from the time it takes to create a UIImage from the image data, as large images take longer to load:

```
struct FullImageView: View {
    …
    var body: some View {
        VStack {
            …
            if let data = attachment.fullImageData,
               let image = UIImage(data: data) {
                Image(uiImage: image)
                    .resizable()
```

```
                        .scaledToFit()
                }
            }
        }
    }
```

To move this operation to the background, you can use the task view modifier from SwiftUI. This is perfect for running one function asynchronously without blocking the main queue. Plus, the task modifier is tied to the view's lifecycle, which means it's automatically canceled when the view disappears.

Here's how you can use it:

```
    private struct FullImageView: View {
        …
        @State private var image: UIImage? = nil

        var body: some View {
            VStack {
                …
                if let image = image {
                    Image(uiImage: image)
                        .resizable()
                        .scaledToFit()
                } else {
                    ProgressView("Loading Image ...")
                        .frame(minWidth: 300, minHeight: 300)
                }
            }
            .padding()
            .task {
                image = await attachment.createFullImage()
            }
        }
    }
```

In the createFullImage() function, you'll want to perform the heavy lifting of creating a UIImage from data in the background thread. Once I have the image, I set it to the state property "image" that is shown in the body.

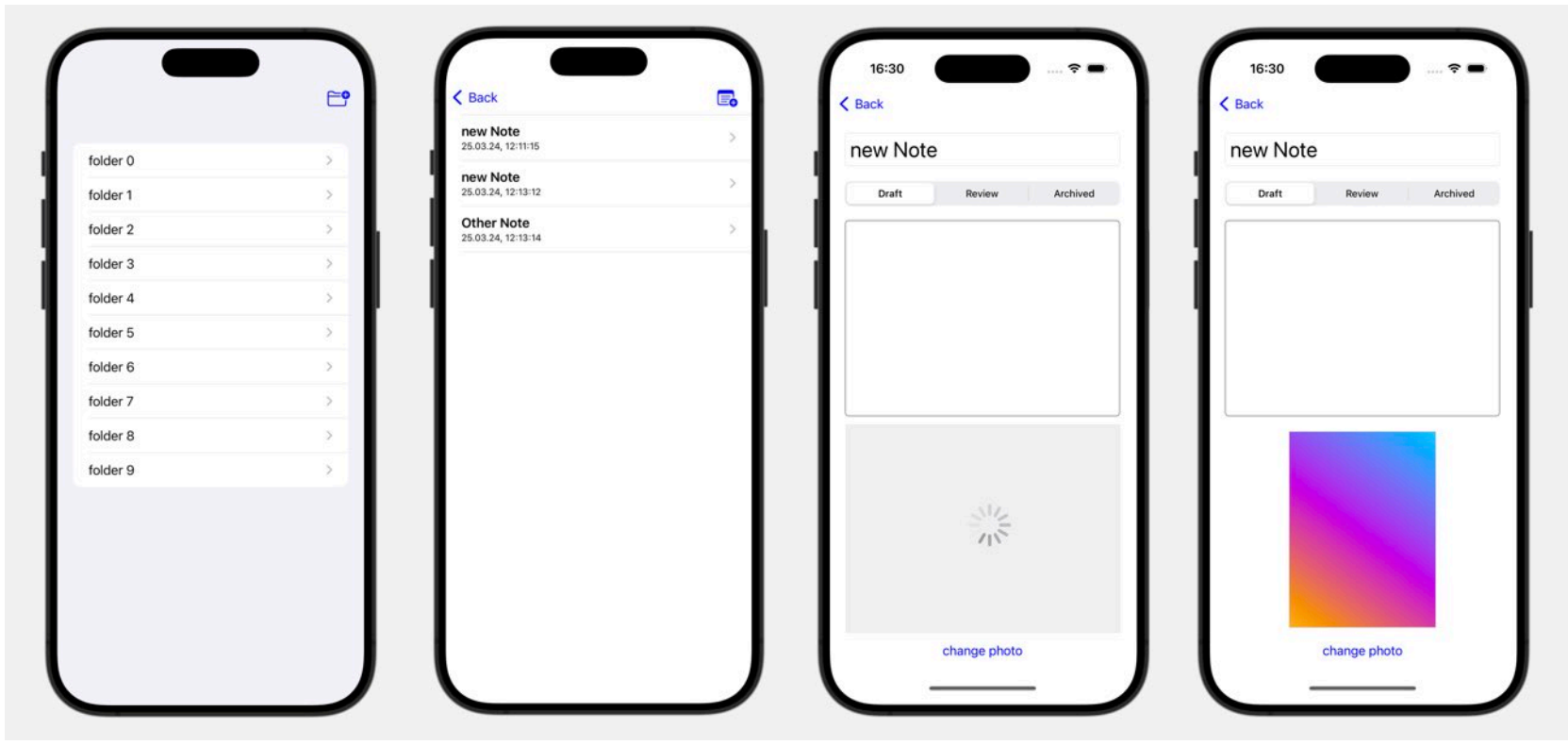
You can place this image processing logic in an extension of your attachment model. This is a good practice for keeping your code organized and reusable. Here's an example of how you might extend your attachment model to include an asynchronous image-loading function:

```
    extension Attachment {
        …

        func createFullImage() async -> UIImage? {
            guard let data = fullImageData else { return nil }
            let image = await Attachment.createImage(from: data)
            return image
        }

        static func createImage(from imageData: Data) async -> UIImage? {
            let image = await Task(priority: .background) {
                UIImage(data: imageData)
            }.value
```

```
            return image
        }
    }
```

Creating the Thumbnail in the Background

When you open a note, the view may not appear directly. It needs to get and if necessary create the thumbnail. This process is running on the main thread. I would like to change this to move the thumbnail generation to the background. Thus when you open the note detail view with an attachment, you would first see a loading indicator. Shortly after, the thumbnail should be visible:



First, let's look at the code that blocks the main thread. In NoteAttachmentView, the "getThumbnail" is blocking the main thread:

```
    struct NoteAttachmentView: View {

        @ObservedObject var attachment: Attachment

        var body: some View {
            Group {
                if let image = attachment.getThumbnail() {
                    Image(uiImage: image)
                        .resizable()
                        .scaledToFit()
                }
            }
        }
    }
```

I am updating the getThumbnail function to use async/await and run the code in the background:

```
extension Attachment {
    …

    func getThumbnail() async -> UIImage? {
        // If thumbnail data is already loaded, no need to run in the background
        guard thumbnailData == nil else {
            // This is fine to run on the main thread as it is expected to be quick
            let image = await Attachment.createImage(from: thumbnailData!)
            return image
        }

        // If there is no full image data, return nil
        guard let fullImageData = fullImageData else {
            return nil
        }

        // Create a thumbnail from the full image data in the background
        let newThumbnail = await Task(priority: .background) {
            // This should be run in the background as it can be time-consuming
            Attachment.createThumbnail(from: fullImageData,
                                       thumbnailPixelSize: 600)
        }.value

        // Convert the thumbnail to data in the background
        let thumbnailData = await Task(priority: .background) {
            // This should be run in the background as it can be time-consuming
            #if os(iOS)
            return newThumbnail?.pngData()
            #else
            return newThumbnail?.tiffRepresentation
            #endif
        }.value

        // Update the thumbnail data on the main thread
        await MainActor.run {
            // UI or state updates should be performed on the main thread
            self.thumbnailData = thumbnailData
        }

        return newThumbnail
    }
}
```

Here are some important points to consider:

1. The **thumbnail creation** function createThumbnail is placed in a background task using Task(priority: .background). This is because creating a thumbnail from the full image data can be time-consuming.

2. Similarly, **converting the image to data** with “newThumbnail?.pngData()” or “newThumbnail?.tiffRepresentation" is also placed in a background task. This operation can also be time-consuming because it involves data encoding.

3. The await MainActor.run block is used to **update self.thumbnailData on the main thread,** which is necessary because it potentially affects the UI state.

The time-consuming tasks are being dispatched to the background and UI updates being handled on the main thread.

Using the new async function in NoteAttachmentView:

```
struct NoteAttachmentView: View {

    @ObservedObject var attachment: Attachment
    @State private var thumbnailImage: UIImage? = nil

    var body: some View {
        Group {
            if let image = thumbnailImage {
                Image(uiImage: image)
                    .resizable()
                    .scaledToFit()

             } else {
                Color.gray
            }
        }
        .task(id: attachment.fullImageData) {
            thumbnailImage = nil
            thumbnailImage = await attachment.getThumbnail()
        }
    }
}
```

By using the task view modifier and asynchronous functions, we significantly improve the user experience. The image now loads in the background, allowing the user to interact with the app while the image is being processed. The use of the task modifier tied to the view's lifecycle ensures that the image is only updated when necessary.

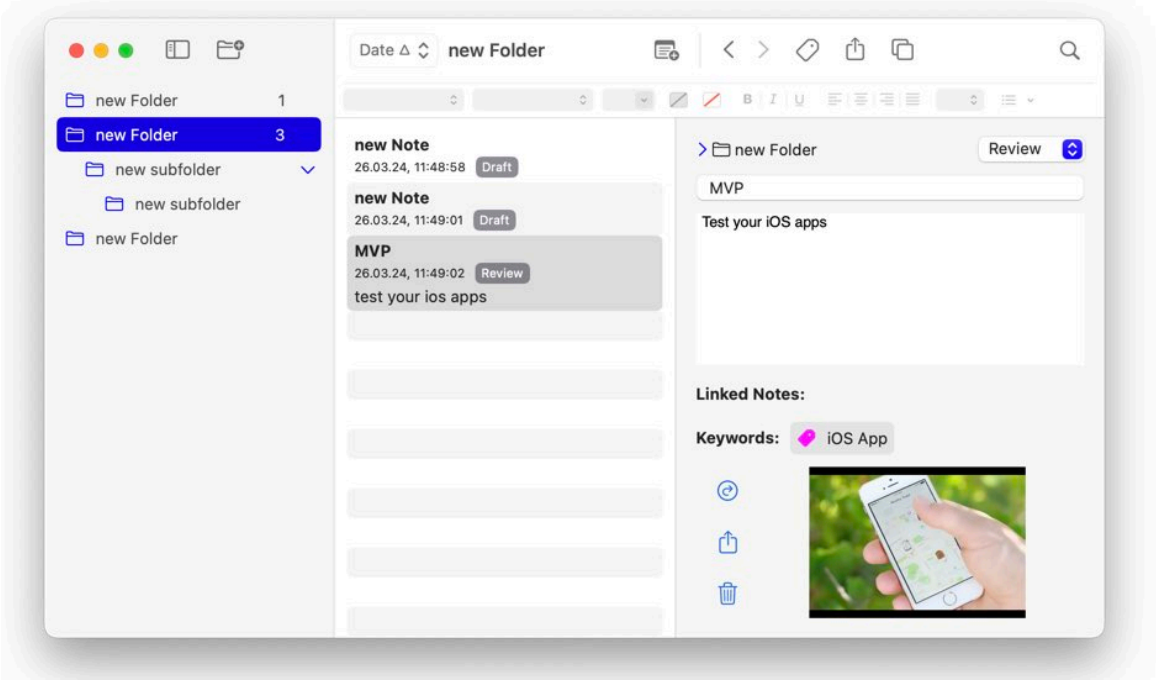
Remember, using background processing and asynchronous functions can greatly improve the responsiveness of your app's UI. It's a powerful technique to consider when you want to balance performance and user experience.

5. FETCH REQUEST WITH PREDICATES

5.1 INTRODUCTION TO SEARCH AND SORT

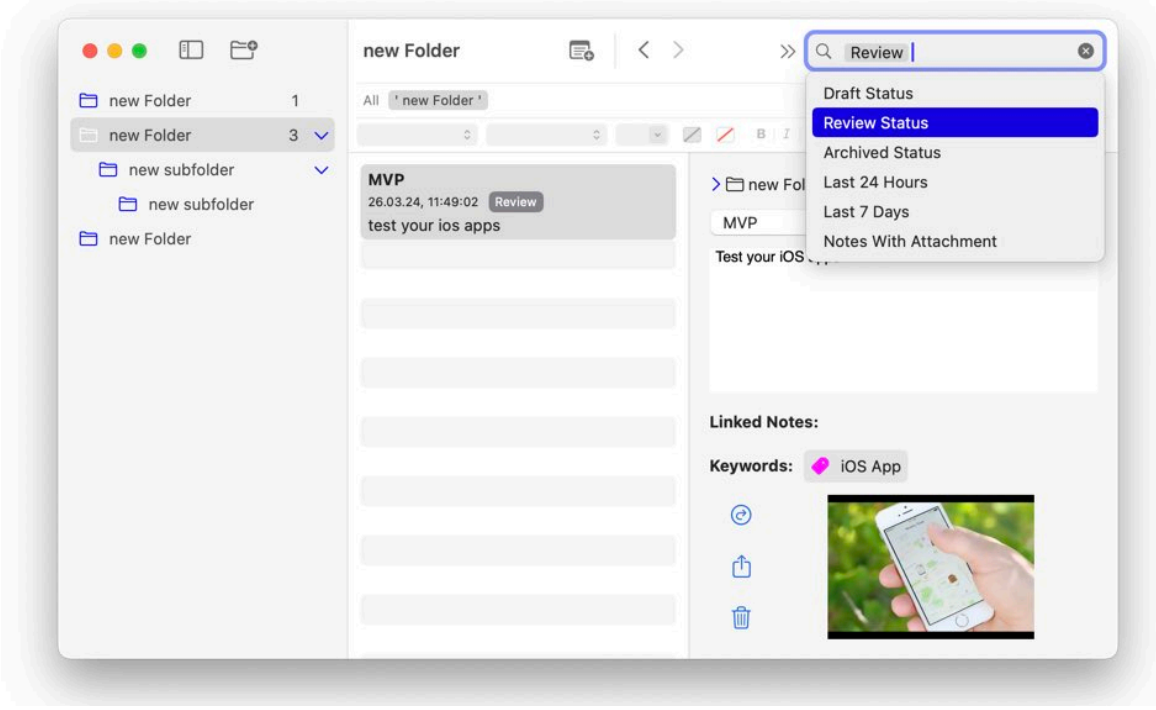
In any relational database, the ability to perform advanced sorting and filtering is a key advantage. By mastering these techniques, you can introduce powerful features to your iOS and macOS applications.

Up until now, I've established numerous relationships in our data model, but I haven't yet leveraged filtering to its full potential. For instance, we haven't implemented functionality to display top folders or nested folders based on specific criteria.



In this section, I'm going to demonstrate how to apply filtering within unit tests for folders. Then, we'll explore how to integrate keywords, which I've yet to add to the app. We'll need to include a button to reveal the search fields, allowing us to either link to an existing keyboard or create a new one. Finally, I'll cover how to perform searches and sorting on notes.

Imagine adding a search text field to our app, enabling users to search across various attributes, such as note titles or types. For notes, the search could encompass the note's title or body text. Furthermore, you can introduce as many filters as you like. For example, you might want to find notes that are not archived, contain an image, and were created within the last week.

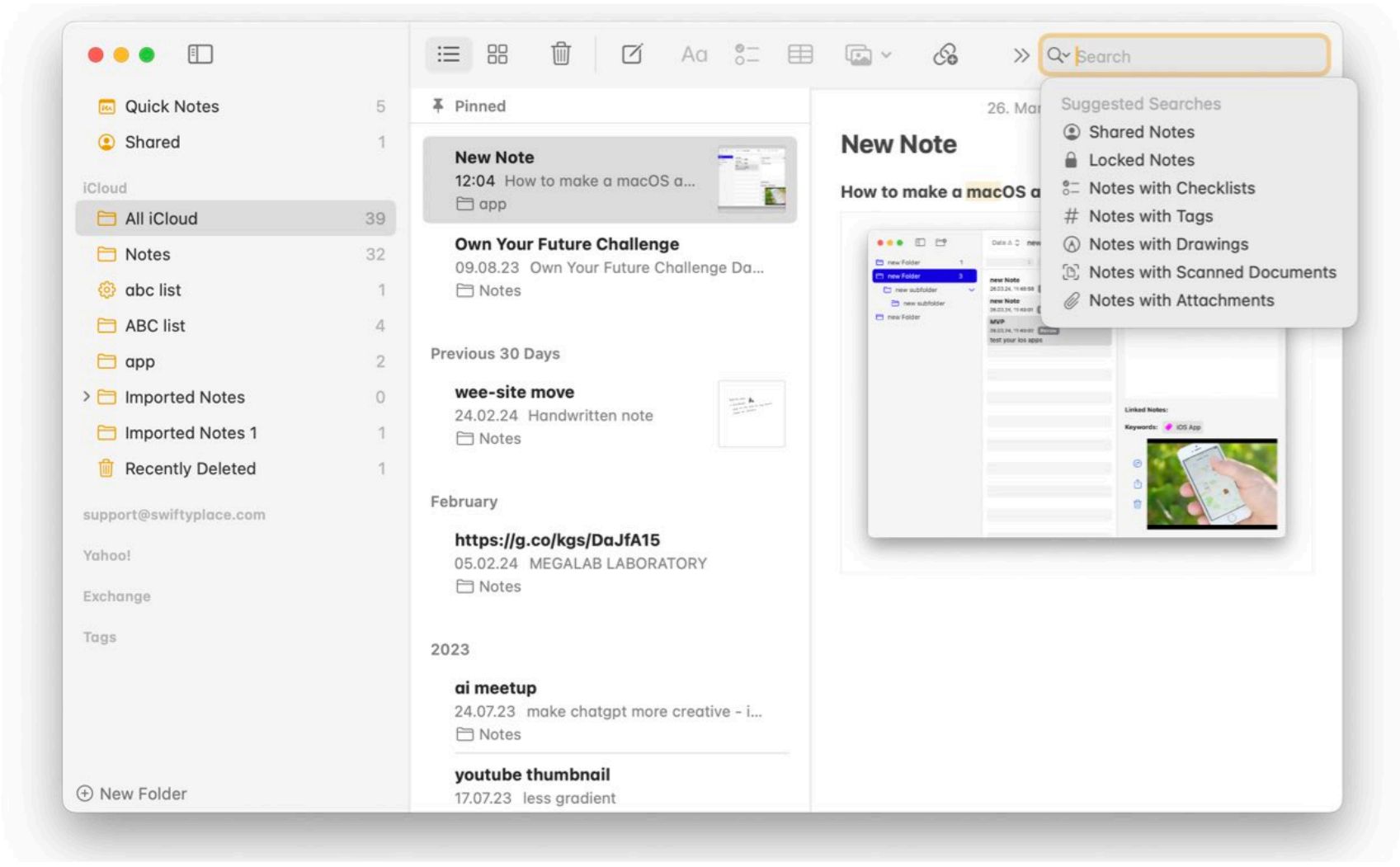


To further motivate you, let's look at two example applications that make extensive use of sorting and filtering: the Notes app and the Finder app on macOS.

The Notes App

In the Notes app, you can select a folder on the left side, which filters the notes to only show those belonging to the selected folder. The notes are then sorted by date and displayed in sections, such as the last seven days, the previous month, and so on.

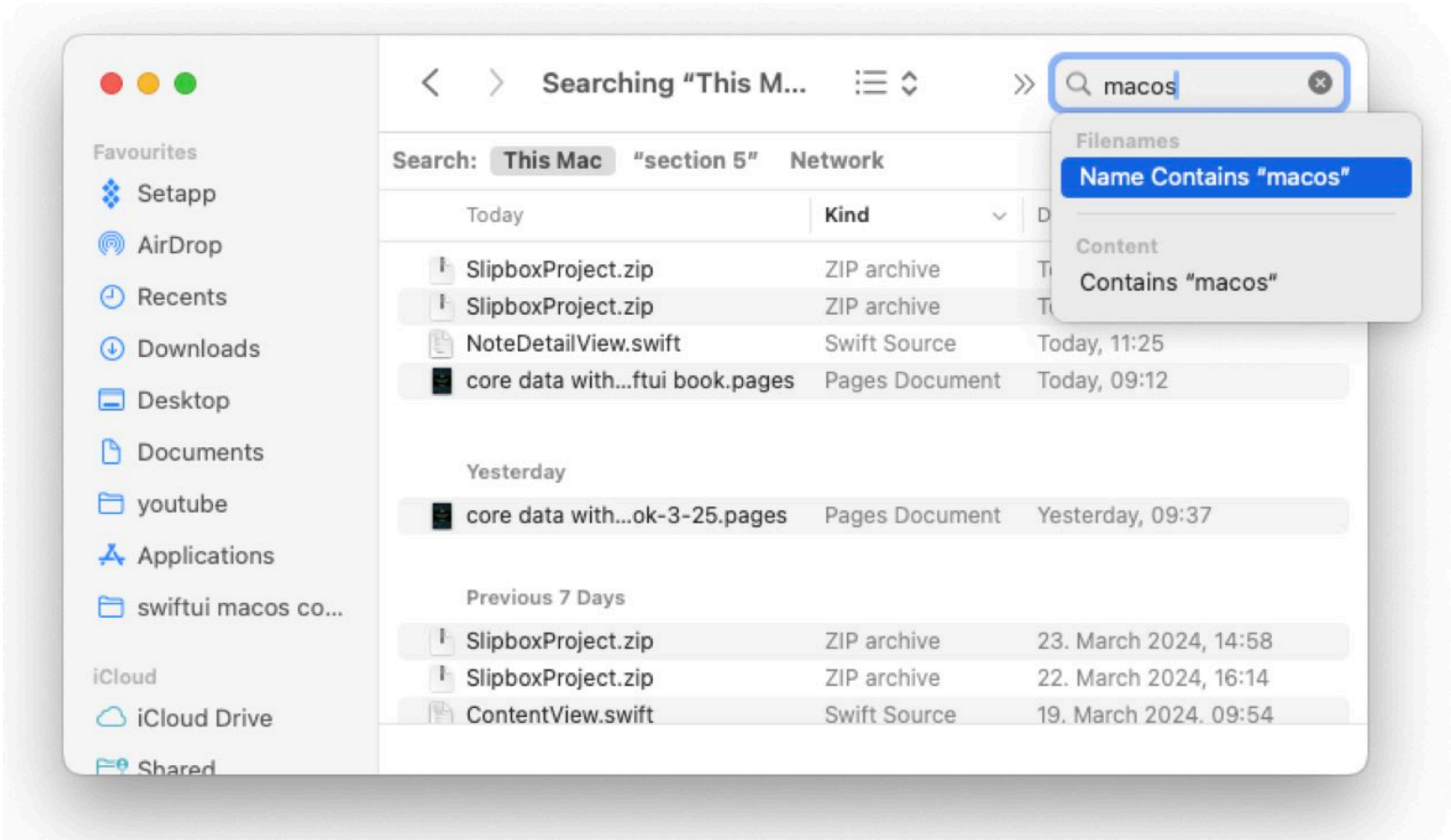
The search functionality in the Notes app offers sectioned sorting and highlights the search term within the notes. When you start typing in the search field, the app suggests searches and allows you to use tokens to refine your search. For example, you can filter by notes that have tags or attachments.



The Finder App

The Finder app's search feature is crucial for locating files and documents. You can specify whether to search by file name or content, and you can filter by file type, location, and more. Finder also allows you to sort results alphabetically or by other attributes and offers a column or list view.

Both apps utilize sectioned sorting, which groups items based on certain criteria, such as file type or date added. This is a feature that we can also implement in our iOS app using SwiftUI's new built-in support for sectioned fetch results.



Implementing Core Data Filtering and Sorting in SwiftUI

Throughout this section, I’ll focus on how to create NSPredicates for filtering and NSSortDescriptors for sorting within Core Data fetch requests. We’ll learn how to set up these features in Core Data and connect them to our SwiftUI views.

With iOS 15 and later, the @FetchRequest property wrapper has become more dynamic, allowing us to update the configuration of fetch requests at runtime. I’ll show you how to use this property wrapper, as well as how to initialize fetch requests with specific configurations.

We will explore three main areas in our app:

1. **Filtering** topmost folders in the folder section.

2. **Sorting** and searching keywords.

3. **Searching and sorting** notes by creation date, with additional sectioned sorting based on the note’s status.

By the end of this section, you should have a clear understanding of how to implement sorting and filtering in Core Data for SwiftUI. I’ll provide you with practical examples and unit tests to ensure you grasp the full range of filtering capabilities. Then, we’ll apply this knowledge to enhance our SwiftUI views with dynamic and powerful data presentation.

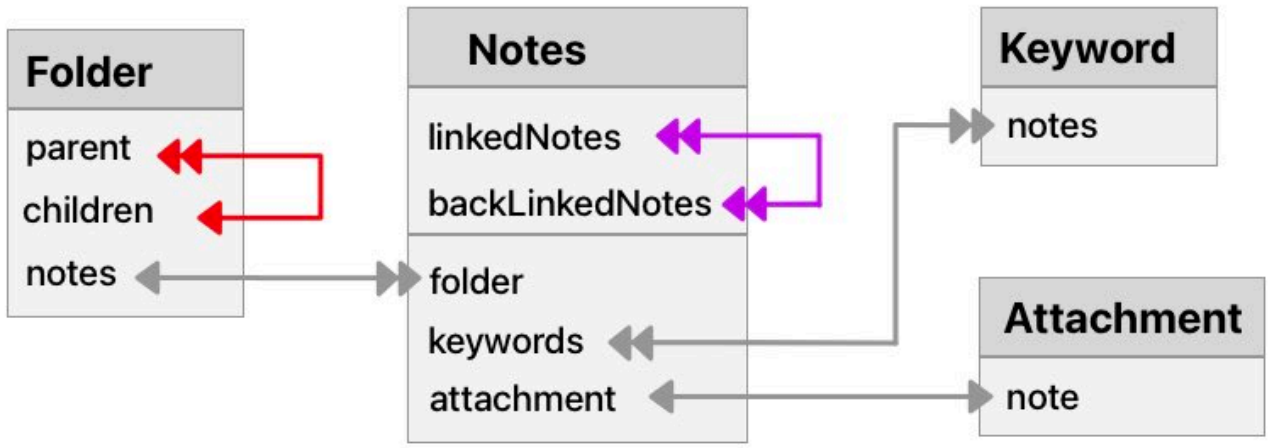
5.2 FETCH TOP LEVEL FOLDERS

In this lesson, I will show you how to fetch top-level folders using Core Data predicates. Predicates allow us to filter and search for specific data in our Core Data database. To practice and demonstrate this, I will be using unit tests in my FolderTests class.

To begin, let's take a look at the format of predicates. Predicates must be in a specific format and can be quite specific depending on the case.

NSPredicate To Search for Top Most Folders

Now, let's focus on fetching the topmost folders from our database. In our database, we have set up a parent-child relationship for folders, where a folder can have a parent. The parent relationship is optional, meaning a folder without a parent is considered a top folder. To fetch the topmost folders, we need to create a predicate that checks if the parent is either nil or non-existent.



To demonstrate this, I will write a unit test called topFolderFetch in my folder test class. In this test, I will create two folders, named "parent" and "child". I set the "child" folder to be a child of "parent" folder.

```
final class FolderTests: XCTestCase {
     …

    func test_top_folder_fetch()  {
        let parent = Folder(name: "parent", context: context)
        let child = Folder(name: "", context: context)
        parent.children.insert(child)

        // fetch
        // check parent folders
    }
}
```

If I want to fetch for parent folders, these are folders that do not have a parent. That means the folders parent property is nil:

```
topFolder.parent = nil
```

The predicate that looks for folders without a parent is as follows:

```
let predicate = NSPredicate(format: "parent == nil”)
```

I want to use this predicate later in SwiftUI views. Therefor I am adding a function in Folder that generates a fetch requests and sets this predicate to the request:

```
extension Folder {
    …

    static func topFolderFetch() -> NSFetchRequest<Folder> {
        let predicate = NSPredicate(format: "%K == nil", FolderProperties.parent)
        return Folder.fetch(predicate)
    }
}
```

To make your life easier and to avoid hardcoding strings, you can use string constants for your attribute names. Here’s an example of how you can structure these constants:

```
extension Folder {
  …
}
//MARK: - define my string constants

struct FolderProperties {
    static let parent = "parent"
    static let children = "children_"
    static let name = "name_"
}
```

Then, I will fetch the folders that don’t have a parent and verify that only the topmost folder is returned.

Here’s the code for the unit test:

```
final class FolderTests: XCTestCase {
     …

    func test_top_folder_fetch()  {

        let parent = Folder(name: "parent", context: context)
        let child = Folder(name: "", context: context)
        parent.children.insert(child)

        //let fetchRequest = Folder.fetch(.all) //will return 2, child and parent
        let fetchRequest = Folder.topFolderFetch()
        let retrievedFolders = try! context.fetch(fetchRequest)

        XCTAssertTrue(retrievedFolders.count == 1)
        XCTAssertTrue(retrievedFolders.contains(parent))
    }
}
```

Performance Optimization: Fetch Limit

When it comes to performance optimization, consider using fetchLimit to limit the number of objects fetched. If we wanted to show a list of folder in SwiftUI, you would probably only show 10-20 folders initially. If your database is large and has 100reds of folders, you would maybe want to limit how much data to fetch.

You can use the fetchLimit property of fetch request, to define how many objects are returned. In this example I will retrieve 2 folders only:

```
    func test_fetch_first_2_folders() {

        _ = Folder(name: "parent", context: context)
        _ = Folder(name: "parent", context: context)
        _ = Folder(name: "parent", context: context)

        let fetchRequest = Folder.topFolderFetch()
        fetchRequest.fetchLimit = 2

        let retrievedFolders = try! context.fetch(fetchRequest)

        XCTAssertTrue(retrievedFolders.count == 2)
    }
```

You can also use fetchBatchSize to load objects in batches, which is particularly useful for large data sets:

```
    func test_batch_size() {
        _ = Folder(name: "parent", context: context)
        _ = Folder(name: "parent", context: context)
        _ = Folder(name: "parent", context: context)

        let fetchRequest = Folder.topFolderFetch()
        fetchRequest.fetchBatchSize = 2

        let retrievedFolders = try! context.fetch(fetchRequest)

        XCTAssertTrue(retrievedFolders.count == 3)
    }
```

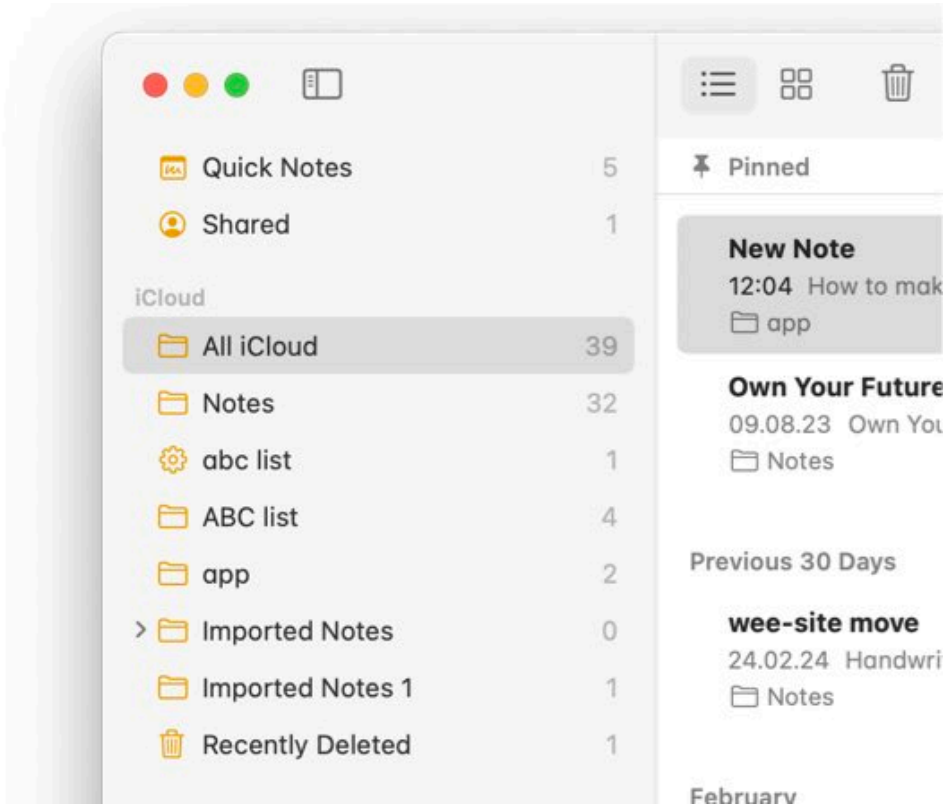
Performance Optimization: Result Type

In some situation, you dont want all the data and its properties. Maybe you only care for the number of objects in the database. For example, how many folders fulfil a search createrion. In the below example, I set the result type to “countResultType”:

```
    func test_fetch_count_top_folders() {
        _ = Folder(name: "parent", context: context)
        _ = Folder(name: "parent", context: context)
        _ = Folder(name: "parent", context: context)

        let fetchRequest = Folder.topFolderFetch()
        fetchRequest.resultType = .countResultType

        let retrievedFoldersCount = try! context.count(for: fetchRequest)
        XCTAssertTrue(retrievedFoldersCount == 3)
    }
```

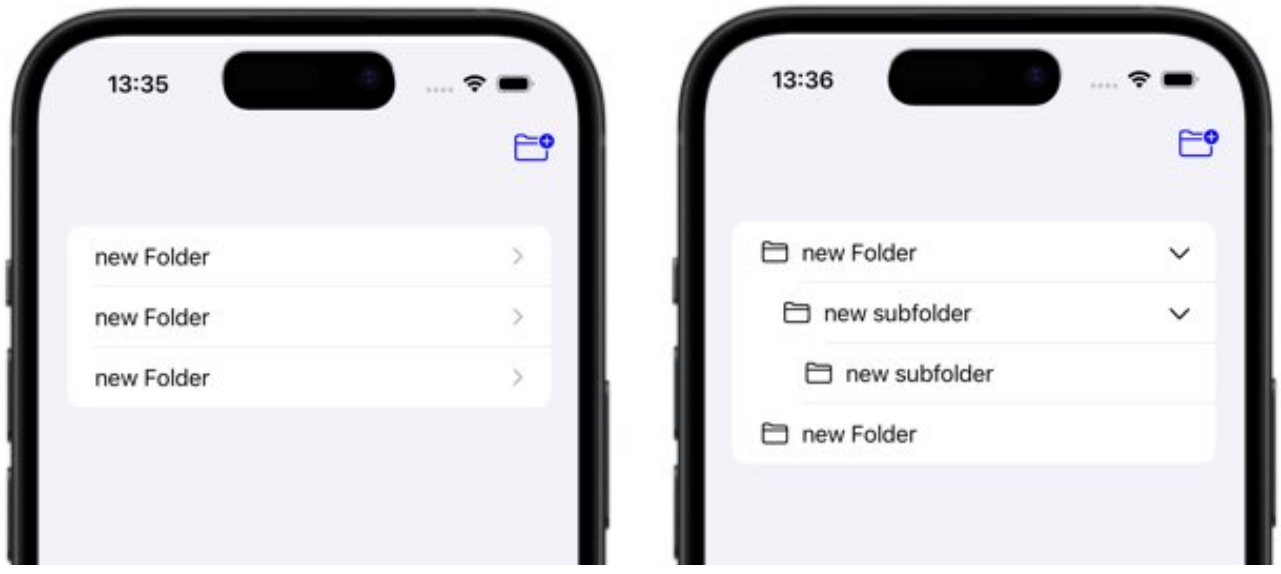
You tell the context to count for the fetch request and the result is the number of objects.

Which you could use to show the number of notes for each folder like in the notes app:



5.3 FOLDER LIST VIEW

In this section, I'll guide you through the process of modifying the folder list in our SwiftUI app to display a nested folder structure. Initially, our app shows all folders without any hierarchy. We aim to change this to represent parent-child relationships among folders. Let's dive into the necessary changes step by step.



Creating Nested Folder Examples

The preview data I'm currently using is from the persistent store's preview, which does not reflect any nested structure. To demonstrate nesting, I'll create an example in the FolderHelper:

```
extension Folder {
    …

    static func nestedFolderExample(context: NSManagedObjectContext) -> Folder {
        let parent = Folder(name: "parent", context: context)
        let child1 = Folder(name: "child 1", context: context)
        let child2 = Folder(name: "child 2", context: context)
        let child3 = Folder(name: "child 3", context: context)
        parent.children.insert(child1)
        parent.children.insert(child2)
        child2.children.insert(child3)

        return parent
    }
}
```

Which I use in FolderListView for the preview:

```
#Preview {
    let context = PersistenceController.createEmpty().container.viewContext
    let nestedFolder = Folder.nestedFolderExample(context: context)
    _ = Folder(name: "second", context: context)

    return NavigationView {
        FolderListView(selectedFolder: .constant(nestedFolder))
            .environment(\.managedObjectContext, context)
    }
}
```

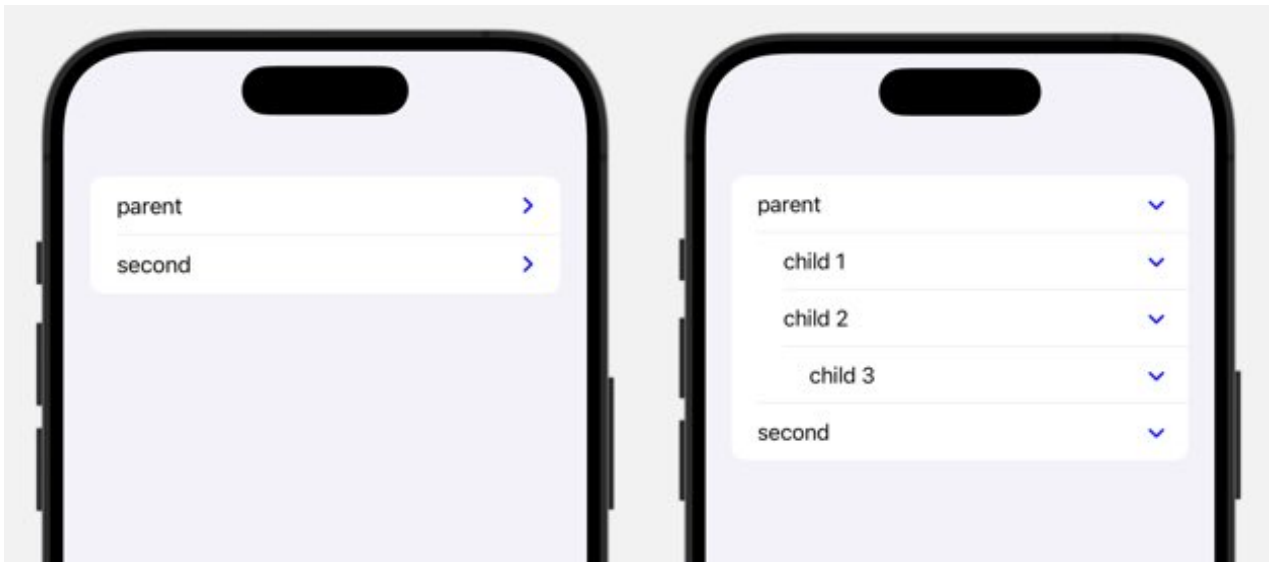
Hierarchical List

Currently, only the fetched folders are shown. But I want to add toggles to show the child folders as well. This is what folders are shown currently:

```
struct FolderListView: View {
   …
    @FetchRequest(fetchRequest: Folder.topFolderFetch())
    private var folders: FetchedResults<Folder>

      var body: some View {
        List(selection: $selectedFolder) {
            ForEach(folders) { folder in
                FolderRow(folder: folder)
            }
            .onDelete()
        }
    }
}
```

You can use a SwiftUI component either List with children or OutlineGroup:

```
List(Array(folders), children: \.sortedChildren) { folder in
    FolderRow(folder: folder)
}

List {
    OutlineGroup(Array(folders), children: \.sortedChildren) { folder in
        FolderRow(folder: folder)
    }
}
```



I need to pass an array of folders. The key path to the nesting has to be defined and needs to be of type optional [Folder]. I created a computed property to Folder that gives me exactly this type:

```
extension Folder {
    …

    var sortedChildren: [Folder]? {
        (children_ as? Set<Folder>)?.sorted()
    }
}
```

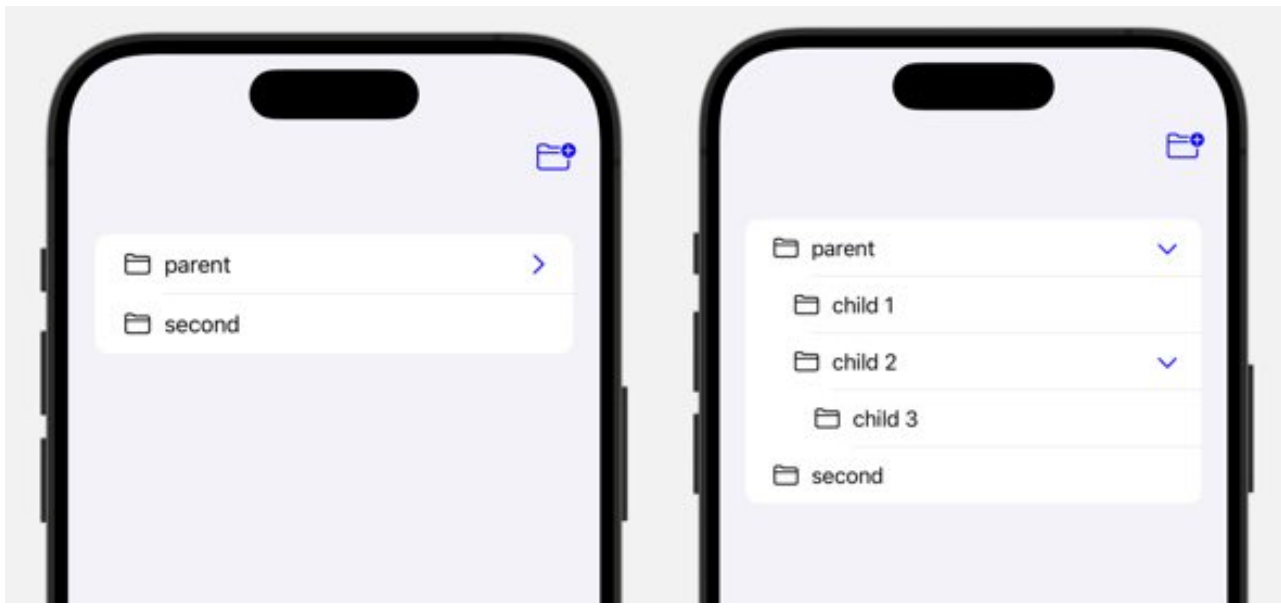
There are a few limitations to this approach. You can not add onDelete or onMove to change the folders. The styling of the hierarchical list is fixed with the discourse buttons.

Therefore, I choose to write a custom recursive list. This is relatively easy and strait forward.

Recursive Folder View

To display the nested folders, I'll create a custom recursive view instead of using the built-in List with children. This gives me more control over the UI and data structure:

```
List(selection: $selectedFolder) {
    ForEach(folders) { folder in
        RecursiveFolderView(folder: folder)
   }
   .onDelete()
}
```



The FolderListView leverages the RecursiveFolderView to show the hierarchy:

```
struct RecursiveFolderView: View {

    @ObservedObject var folder: Folder
    @State private var showSubfolders = true

    var body: some View {
        HStack {
            Image(systemName: "folder")
```

```
                FolderRow(folder: folder)

                Spacer()

                // Disclosure Button
                if folder.children.count > 0 {
                    Button {
                        withAnimation {
                            showSubfolders.toggle()
                        }

                    } label: {
                        Image(systemName: "chevron.right")
                            .rotationEffect(.init(degrees: showSubfolders ? 90 : 0))
                    }
                }
            }
            .tag(folder)

            if showSubfolders {
                ForEach(folder.children.sorted()) { subfolder in
                    RecursiveFolderView(folder: subfolder)
                        .padding(.leading)
                }
                .onDelete(perform: deleteFolders(offsets:))
            }

        }

        private func deleteFolders(offsets: IndexSet) {
            offsets.map { folder.children.sorted()[$0] }.forEach(Folder.delete(_:))
        }
    }
```

I am adding a button to show and hide the subfolders. This button is only shown if the current folder has subfolders. A state property “showSubfolders” is updated when I press the button.

If the “showFolders” property is true, I show another ForEach with the children of the current folder. Inside this ForEach, I call recursively the view I am in “RecursiveFolderView”.

I am adding again the onDelete modifier, which allows me to delete any subfolder.

5.4 NSPREDICATE FOR NOTES SEARCH TERM AND COMPOUND PREDICATES

In this section, we will explore various examples of using predicates to search for notes in Core Data. We will cover different conditions and how to combine predicates to create compound fetch requests.

To begin, let's create a new test class to separate our fetch tests:

```
import XCTest
@testable import SlipboxProject
import CoreData

final class NoteFetchTest: XCTestCase {

    var controller: PersistenceController!

    var context: NSManagedObjectContext {
        controller.container.viewContext
    }

    override func setUpWithError() throws {
        self.controller = PersistenceController.createEmpty()
    }

    override func tearDownWithError() throws {
        self.controller = nil
    }

    // testing searching for notes
}
```

I am also adding a storage in Note+Helper extension, that holds all the strings for the attributes of Note:

```
struct NoteProperties {
    static let title = "title_"
    static let bodyText = "bodyText_"
    static let status = "status_"
    static let creationDate = "creationDate_"

    static let folder = "folder"
    static let keywords = "keywords_"
}
```

Searching For Notes Title

Now, let's dive into the first example. Suppose you want to find notes that contain a specific search term in their title. Here's how you can write a test function for this:

```
func test_search_term_notes() {
    let note1 = Note(title: "Tast", context: context)
    let note2 = Note(title: "Dummy", context: context)
```

```
        let searchTerm = "tast"

        let predicate = NSPredicate(format: "%K CONTAINS[cd] %@“,
                                    NoteProperties.title ,
                                    searchTerm as CVarArg)

        let request = Note.fetch(predicate)
        let retrievedNotes = try! context.fetch(request)

        XCTAssertTrue(retrievedNotes.count == 1)
        XCTAssertTrue(retrievedNotes.contains(note1))
        XCTAssertFalse(retrievedNotes.contains(note2))
    }
```

In this snippet, **%K** is used as a placeholder for the attribute name, and %@ is used for the value of the search term. The [cd] in CONTAINS[cd] makes the search case-insensitive. I added 2 note instances with one note containing the search term. At the end, I check that I only get the note back that has the search term included.

Searching for Multiple Search Criteria

Moving on, let’s say you want to search for a term that could be in either the note’s title or body text. This is where compound predicates come into play. You can combine multiple predicates using logical OR. Here’s an example of how to do this:

```
    let searchTerm = "test"

    let predicates = [NSPredicate(format: "%K CONTAINS[cd] %@",
                                  NoteProperties.title,
                                  searchTerm as CVarArg),
                      NSPredicate(format: "%K CONTAINS[cd] %@“,
                                  NoteProperties.bodyText ,
                                  searchTerm as CVarArg)]

    let predicate = NSCompoundPredicate(orPredicateWithSubpredicates: predicates)

    let request = Note.fetch(predicate)
```

In this test, you create two predicates: one for the title and another for the body text. Then, you combine them with NSCompoundPredicate(orPredicateWithSubpredicates:) to perform a logical OR operation.

Searching for Multiple Search Terms

Imagine you want to search for multiple terms within the note’s title and ensure all terms are present. We will use a compound predicate with an AND operator:

```
    let searchTerms = ["Hello", "World", "and"]

    var predicates = [NSPredicate]()
```

```
    for term in searchTerms {
            let p = NSPredicate(format: "%K CONTAINS[cd] %@",
                                NoteProperties.title,
                                term as CVarArg)
            predicates.append(p)
    }

    let predicate = NSCompoundPredicate(andPredicateWithSubpredicates: predicates)

    let request = Note.fetch(predicate)
```

I create an array of predicates. Then I lop through the array of all search terms. For each search term, I create a predicate that filters for the title of the note. Lastly I create a compound predicate that combines this search term predicates. I use an “AND” operation. This means I want only notes that contain all search terms.

5.5 NOTES FETCH FOR BOOLEAN AND ENUM ATTRIBUTES

I’m going to show you how to search for notes based on different status conditions using enums. Enums are a powerful way to represent a fixed set of values, and they can be very useful when filtering data.

Let’s start by creating a new test function called “searchByStatus”. In this example, we’ll search for notes with a specific status.

```
    func test_filter_by_notes_status_default_draft() {
        let note1 = Note(title: "Hello and World", context: context)
        let note2 = Note(title: "test more world", context: context)
        let note3 = Note(title: "note 3", context: context)

        let filterStatus = Status.draft

        let predicate = NSPredicate(format: "%K == %@",
                                    NoteProperties.status,
                                    filterStatus.rawValue as CVarArg)

        let request = Note.fetch(predicate)
        let retrievedNotes = try! context.fetch(request)

        XCTAssertTrue(retrievedNotes.count == 3)
        XCTAssertTrue(retrievedNotes.contains(note1))
        XCTAssertTrue(retrievedNotes.contains(note2))
        XCTAssertTrue(retrievedNotes.contains(note3))
    }
```

To search for notes with a specific status, we’ll create a predicate using the enum value. The enum is stored as a String in the database. I am using the rawValue of the enum for NSPredicate.

In the above, I am testing that the initial value of the status property is set to draft. In the following, I am using 3 notes of which one has a status of “archived”:

```
    func test_filter_by_notes_status_archived() {
        let note1 = Note(title: "Hello and World", context: context)
        let note2 = Note(title: "test more world", context: context)
        let note3 = Note(title: "note 3", context: context)
        note3.status = .archived

        let filterStatus = Status.archived

        let predicate = NSPredicate(format: "%K == %@",
                                    NoteProperties.status,
                                    filterStatus.rawValue as CVarArg)
        let request = Note.fetch(predicate)
        let retrievedNotes = try! context.fetch(request)

        XCTAssertTrue(retrievedNotes.count == 1)
        XCTAssertFalse(retrievedNotes.contains(note1))
        XCTAssertFalse(retrievedNotes.contains(note2))
        XCTAssertTrue(retrievedNotes.contains(note3))
    }
```

Because I only added one note with the status “archived”, the retrieved notes should be only one.

Filtering With Boolean Properties

You might also search for favorite notes. Assuming you added a favorite attribute of note of type Bool:

```
    func test_search_for_favorite_notes() {
        let note1 = Note(title: "Hello and World", context: context)
        let note2 = Note(title: "test more world", context: context)
        note2.isFavorite = true

        let predicate = NSPredicate(format: "isFavorite == true")

        let request = Note.fetch(predicate)
        let retrievedNotes = try! context.fetch(request)

        XCTAssertTrue(retrievedNotes.count == 1)
        XCTAssertFalse(retrievedNotes.contains(note1))
        XCTAssertTrue(retrievedNotes.contains(note2))
    }
```

The predicate searches for notes with **"isFavorite == true”.** In the above test, I added only one favourite note. To verify the test, I check if the retrieved notes count is one.

5.6 FETCH NOTES FOR TIME PERIOD

You can also search for notes in a certain time interval. Let's say you wanted to search for notes that were created in the last 7 days:

```
    func test_fetch_notes_for_last_7_days() {
        let calendar = Calendar.current

        let beginDate = calendar.date(byAdding: .day, value: -7, to: Date())!

        let note1 = Note(title: "Hello and World", context: context)
        note1.creationDate_ = calendar.date(byAdding: .day, value: -2, to: Date())
        let note2 = Note(title: "test more world", context: context)
        note2.creationDate_ = calendar.date(byAdding: .day, value: -9, to: Date())

        let predicate = NSPredicate(format: "%K < %@",
                                    NoteProperties.creationDate,
                                    beginDate as NSDate)

        let request = Note.fetch(predicate)
        let retrievedNotes = try! context.fetch(request)

        XCTAssertTrue(retrievedNotes.count == 1)
        XCTAssertFalse(retrievedNotes.contains(note1))
        XCTAssertTrue(retrievedNotes.contains(note2))
    }
```

The main difficulty when searching for date properties, is working with Calendar and getting the right reference dates. In the above case, I first get the date for 7 days ago. Then I create 2 notes with different creation dates. Only one of them fulfills the search criterion and should be returned.

5.7 FETCHING NOTES IN RELATIONSHIP TO FOLDERS AND KEYWORDS

For the notes app, that we are building, I would like to create a fetch request that gives me the **notes that belong to a specific folder**. Here is an example of how to search for these notes:

```
    func test_search_notes_for_folder() {
        let note1 = Note(title: "note", context: context)
        let folder = Folder(name: "folder", context: context)
        folder.notes.insert(note1)

        let predicate = NSPredicate(format: "%K == %@", NoteProperties.folder, folder)

        let request = Note.fetch(predicate)
        let retrievedNotes = try! context.fetch(request)

        XCTAssertTrue(retrievedNotes.count == 1)
        XCTAssertTrue(retrievedNotes.contains(note1))
    }
```

You might also want to retrieve **any notes that contain a certain keyword**. You can use the "CONTAINS" terms to search for to-many relationships:

```
    func test_search_notes_with_keyword() {
        let keyword = Keyword(context: context)
        let note1 = Note(title: "dd", context: context)
        note1.keywords.insert(keyword)
        _ = Note(title: "note 2", context: context)
        _ = Note(title: "note 3", context: context)

        let predicate = NSPredicate(format: "%K CONTAINS %@",
                                    NoteProperties.keywords,
                                    keyword)

        let request = Note.fetch(predicate)
        let retrievedNotes = try! context.fetch(request)

        XCTAssertTrue(retrievedNotes.count == 1)
        XCTAssertTrue(retrievedNotes.contains(note1))
    }
```

Or if you have multiple keywords and you want to find **notes that are linked to any of these keywords:**

```
    var selectedKeywords = Set<Keyword>()
    selectedKeywords.insert(keyword2)
    selectedKeywords.insert(keyword1)

    let predicate = NSPredicate(format: "ANY %K in %@",
                                NoteProperties.keywords,
                                selectedKeywords)
```

Or you actually want to exclude these keywords and only want notes that are **not linked** to these keywords:

```
    let predicate = NSPredicate(format: "NONE %K in %@",
                                NoteProperties.keywords,
                               selectedKeywords)
```

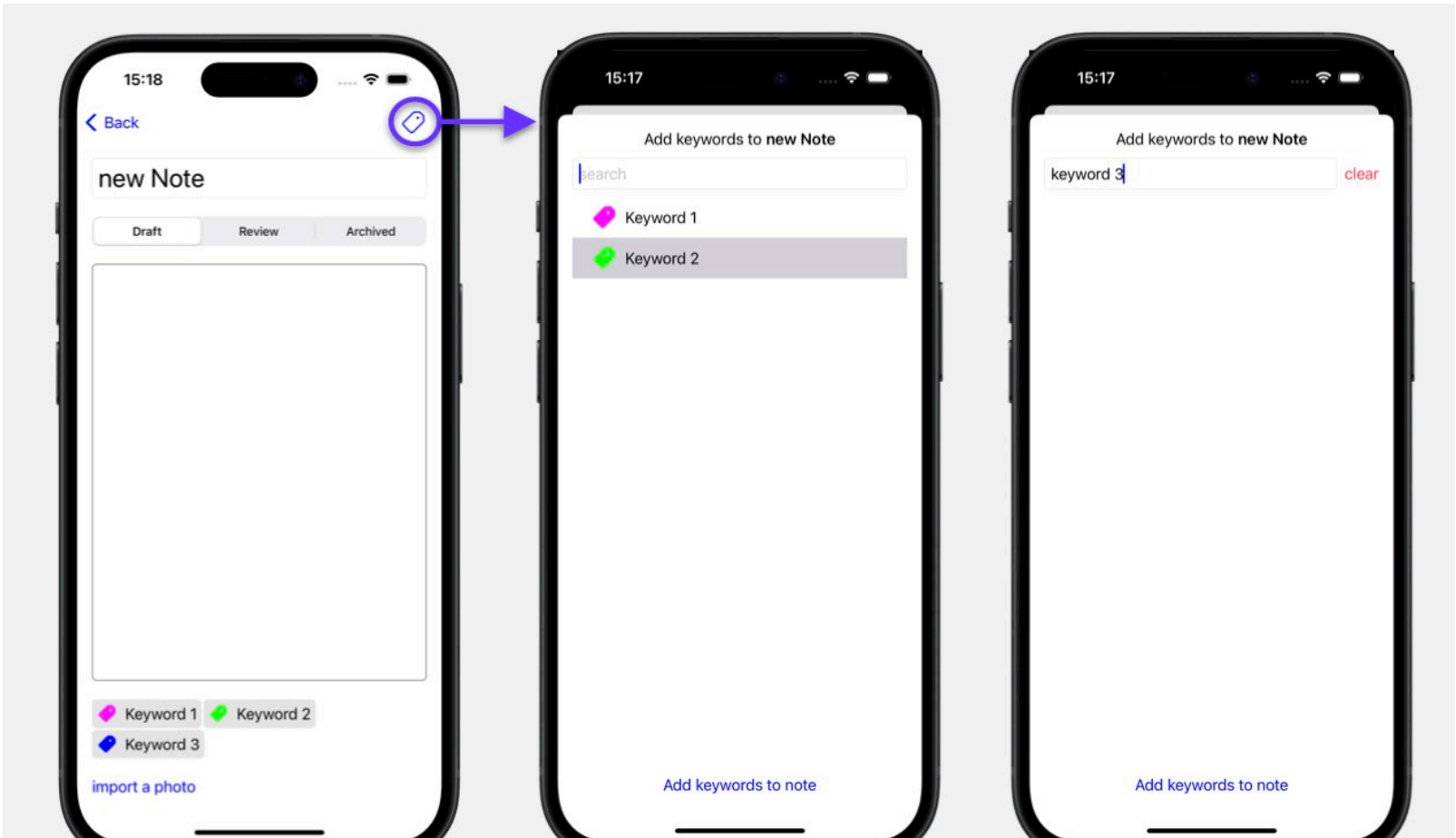
You can also search for **notes that have no keywords** added to them. In this case you can use the "@count" term:

```
    let predicate = NSPredicate(format: "keywords_.@count == 0")
```

This approach allows you to search for notes based on their relationships with other entities. You can modify the predicate to search for notes associated with different folders or even combine multiple predicates to perform more complex searches.

5.8 SHOWING KEYWORDS IN THE NOTES DETAIL VIEW

In this lesson, I will show you how to display keywords in the notes detail view. In the following section, we will also discuss how to add keywords using a keyword editor where all keywords are listed. The user can also search for existing keywords:



Preview Data

First, let's create an example note with some keywords. We'll add a function that returns an example note with default properties, including some keywords. The following creaets 10 example keywords:

```
extension Keyword {
    …

    static func exampleArray() -> [Keyword] {
        let context = PersistenceController.preview.container.viewContext

        var keys = [Keyword]()
        for index in 0...4 {
            let key = Keyword(name: "keyword \(index)", context: context)
            let colorValue = CGFloat(index) / 5
            key.color = Color(red: colorValue , green: 0.5, blue: colorValue)
            keys.append(key)
        }

        return keys
    }
}
```

Which I use for the example Note. The resulting note should have 4 keywords:

```
extension Note {
    …

    static func example() -> Note {
        let context = PersistenceController.preview.container.viewContext

        let note = Note(title: "my note", context: context)
        note.formattedBodyText = NSAttributedString(string: defaultText)

        let keys = Keyword.exampleArray()
        for key in keys {
            note.keywords.insert(key)
        }

        return note
    }
}
```

Showing the Keywords Linked to a Note

To begin, let's create a separate view to display the keywords in a nice flow layout. We'll call this view the "NotesKeywordsCollectionView" and create a separate file for it.

```
import SwiftUI

struct NotesKeywordsCollectionView: View {

    @FetchRequest(fetchRequest: Keyword.fetch(.none)) var keywords

    var body: some View {
        FlowLayout(alignment: .leading, spacing: 2) {
            ForEach(keywords) { keyword in
                Text(keyword.name)
        }
    }
}

#Preview {
    NotesKeywordsCollectionView(note: Note.example())
        .environment(\.managedObjectContext,
                     PersistenceController.preview.container.viewContext)
}
```

For preview purposes, I'll create an example note with some default text and keywords. Here's how you can set up a static function to create an example note.

Fetch Request for Keywords

To show the keywords for a specific note, we'll need to use a fetch request with an NSPredicate. Here's how you can create a fetch request for keywords associated with a note:

```
extension Keyword {
    …

    static func fetch(_ predicate: NSPredicate = .all) -> NSFetchRequest<Keyword> {
        let request = Keyword.fetchRequest()
         request.sortDescriptors = [NSSortDescriptor(keyPath: \Keyword.name_,
                                                     ascending: true)]
         request.predicate = predicate

         return request
     }

    static func fetch(for note: Note) -> NSFetchRequest<Keyword> {
        let predicate = NSPredicate(format: "%K CONTAINS %@",
                                    KeywordProperties.notes,
                                    note)
        return Keyword.fetch(predicate)
    }
}
```

I am adding this in the Keyword extension. This makes it easy to use it in the SwiftUI views as well as write unit tests.

Updating the UI

I can use the new fetch request in the initialiser of NotesKeywordsCollectionView and set it to @FetchRequest. Now only keywords that belong to the current note are shown:

```
struct NotesKeywordsCollectionView: View {

    init(note: Note) {
        self.note = note
        self._keywords = FetchRequest(fetchRequest: Keyword.fetch(for: note))
    }

    let note: Note

    @FetchRequest(fetchRequest: Keyword.fetch(.none)) var keywords

    var body: some View {
      …
    }
}
```

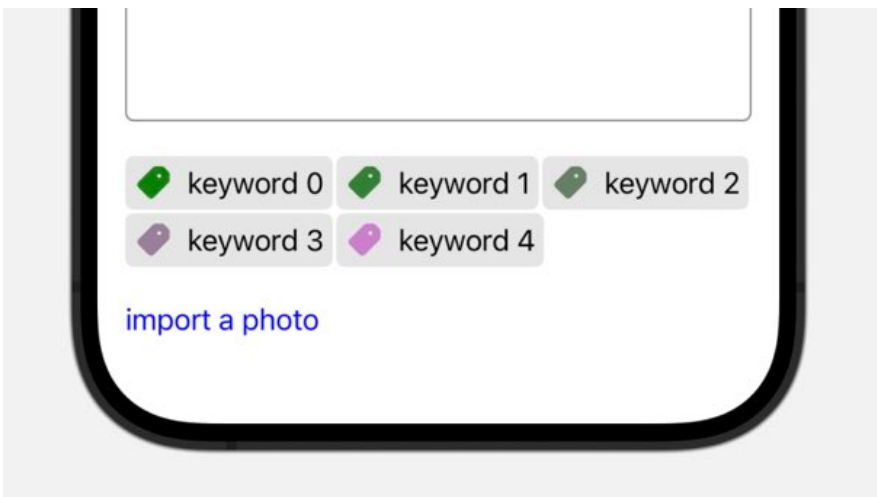
Now, let's update the Note Detail View to use the example note and display the keywords using the Notes Keywords Collection View:

```
struct NoteDetailView: View {
    …

    var body: some View {
        VStack(alignment: .leading, spacing: 20) {
            …

            NotesKeywordsCollectionView(note: note)

            if let attachment = note.attachment {
                NoteAttachmentView(attachment: attachment)
            }

            NotePhotoSelectorButton(note: note)
        }
    }
}
```

Adding Styling to the Keyword View

You can add more styling to the keywords and for example, use the color for the icon next to the keyword name and add a grey background:

```
ForEach(keywords) { keyword in
    HStack {
        Image(systemName: "tag.fill")
            .foregroundColor(keyword.color)
        Text(keyword.name)
    }
     .padding(5)
     .background(RoundedRectangle(cornerRadius: 5).fill(Color(white: 0.9)))
}
```

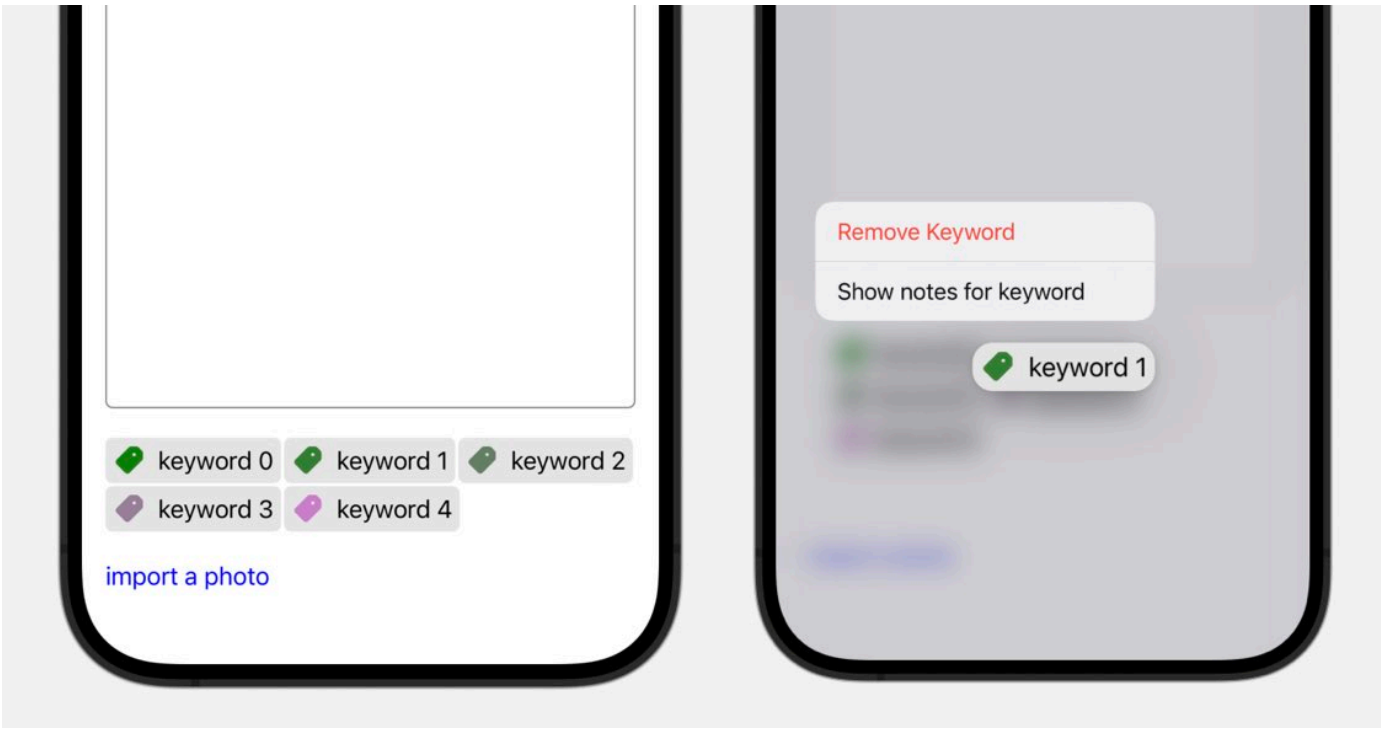


Context Menu to Keywords

You might want to allow the user to remove keywords. You could add a context menu to each keyword field and show a “remove keyword” option.

```
    ForEach(keywords) { keyword in
        // Keyword tat
            .contextMenu {
                Button {
                    keyword.notes.remove(note)
                } label: {
                    Text("Remove Keyword")
                }

                Button {
                    // open new window with keyword and notes
                } label: {
                    Text("Show notes for keyword")
                }
            }
    }
```

Note that I am not deleting the keyword. I only remove the link between the note and keyword.

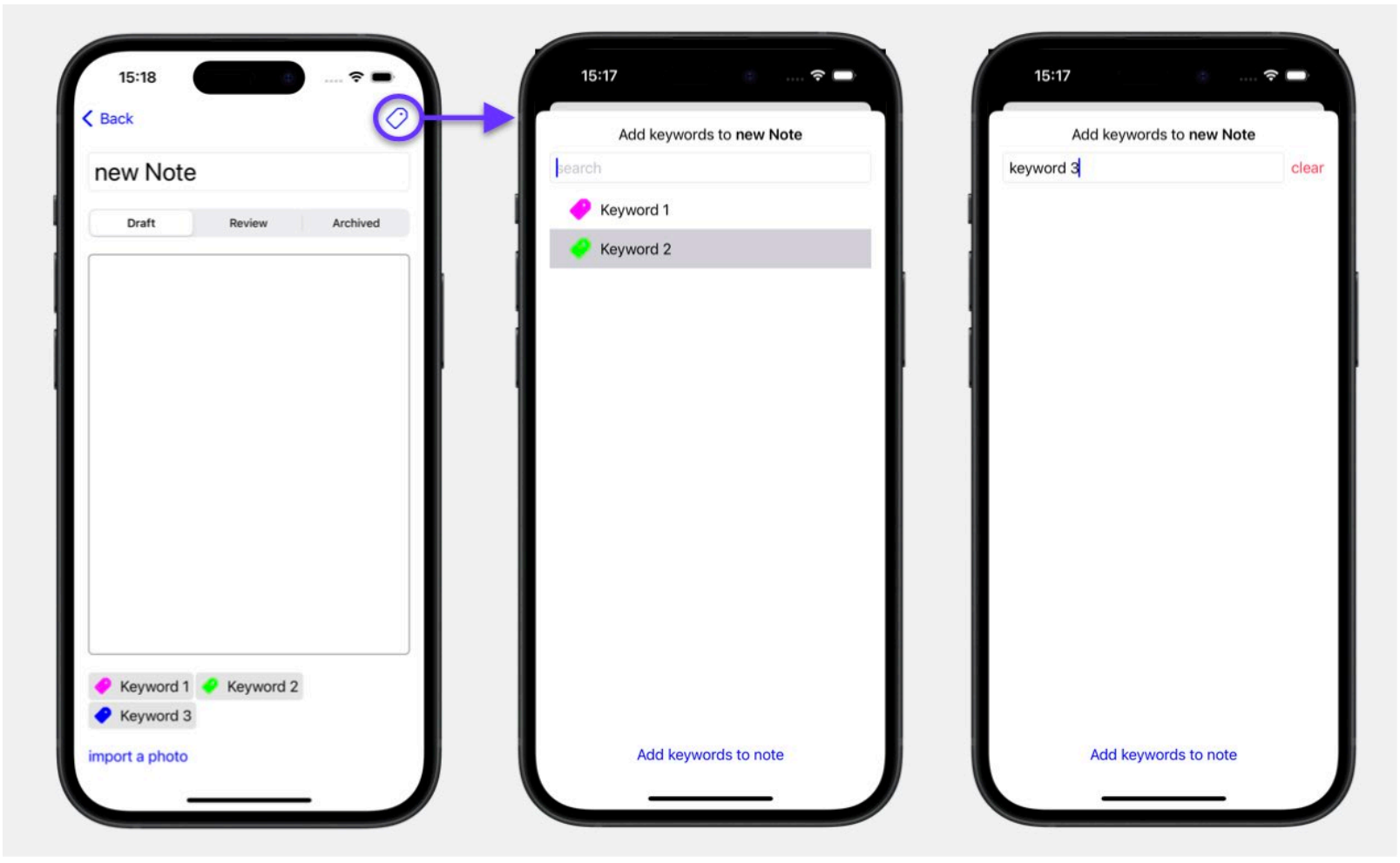


To add keywords, we’ll create a keyword editor view where users can find existing keywords or create new ones. We’ll discuss this in the next lesson.

5.9 VIEW TO ADD KEYWORD TO NOTES

In this section, you will allow the user to create Keywords and add them to a note. You will add a button in the toolbar that opens an editor. First all existing keywords are shown. When the user taps on a keyword, he can tap on the “Add keyword to note” button on the bottom of the screen.

He can also enter a search term in a text field to search for a specific keyword. If the user types a keyword that does not already exist, you will show a button to create as new keyword and link it to the current note:



Showing the Editor

To present the AddKeywordsToNoteView, use a button in the NoteDetailView and show the view as a popover or sheet:

```
struct NoteDetailView: View {

    @ObservedObject var note: Note
    @State private var showKeywordView = false

    var body: some View {
      VStack(alignment: .leading, spacing: 20) {
         …
      }
      .toolbar {
          ToolbarItem {
              Button {
                  showKeywordView.toggle()
              } label: {
```

```
                    Image(systemName: "tag")
                }
                .popover(isPresented: $showKeywordView) {
                    AddKeywordsToNoteView(note: note)
                        .environment(\.managedObjectContext, context)
                }
            }
        }
    }
}
```

Next, let's set up a new view where you can add keywords to a note. I'll call it **AddKeywordsToNoteView**. This view requires a reference to the note you're working with, so we know which notes we should add the keywords to:

```
struct AddKeywordsToNoteView: View {

    let note: Note
    @FetchRequest(fetchRequest: Keyword.fetch(.all)) var keywords
    @State private var selectedKeywords = Set<Keyword>()

    var body: some View {
        VStack {
            HStack(spacing: 0) {
                Text("Add keywords to ")
                Text(note.title).bold()
            }

            List(selection: $selectedKeywords) {
                ForEach(keywords) { keyword in
                    HStack {
                        Image(systemName: "tag.fill")
                            .foregroundColor(keyword.color)
                        Text(keyword.name)
                    }
                    .tag(keyword)
                }
            }
        }
    }
}
```

Above, I am fetching all keywords and show them in a list. I created a state property "selectedKeywords" that holds all the selected keywords from the list.

If at least one keyword is selected, I will show a button "Add keywords to note"

```
    if selectedKeywords.count > 0 {
        Button {
            selectedKeywords.forEach { key in
                note.keywords.insert(key)
            }
            dismiss()
        } label: {
            Text("Add keywords to note")
        }
    }
```

I am dismissing the sheet when the user presses on the “Add” button.

Searching for a Keyword

You’ll need a TextField to input the search term and a list to display the keywords. Add a textfield before the list of keywords:

```
struct AddKeywordsToNoteView: View {
    …

    @State private var searchTerm: String = ""
    @FetchRequest(fetchRequest: Keyword.fetch(.all)) var keywords

    var body: some View {
        VStack {
            …
           TextField("search", text: $searchTerm)

           List(keywords, selection: $selectedKeywords) {
               …
           }
        }
        .onChange(of: searchTerm) { newValue in
            if !newValue.isEmpty {
                keywords.nsPredicate = NSPredicate(format: "%K CONTAINS[cd] %@“,
                                                   KeywordProperties.name ,
                                                   searchTerm as CVarArg)
            } else {
                keywords.nsPredicate = nil
            }
        }
    }
```

For the list of keywords, use a FetchRequest to dynamically fetch keywords as the user types. With iOS 15 and newer, you can update the fetch request’s predicate using the onChange view modifier.

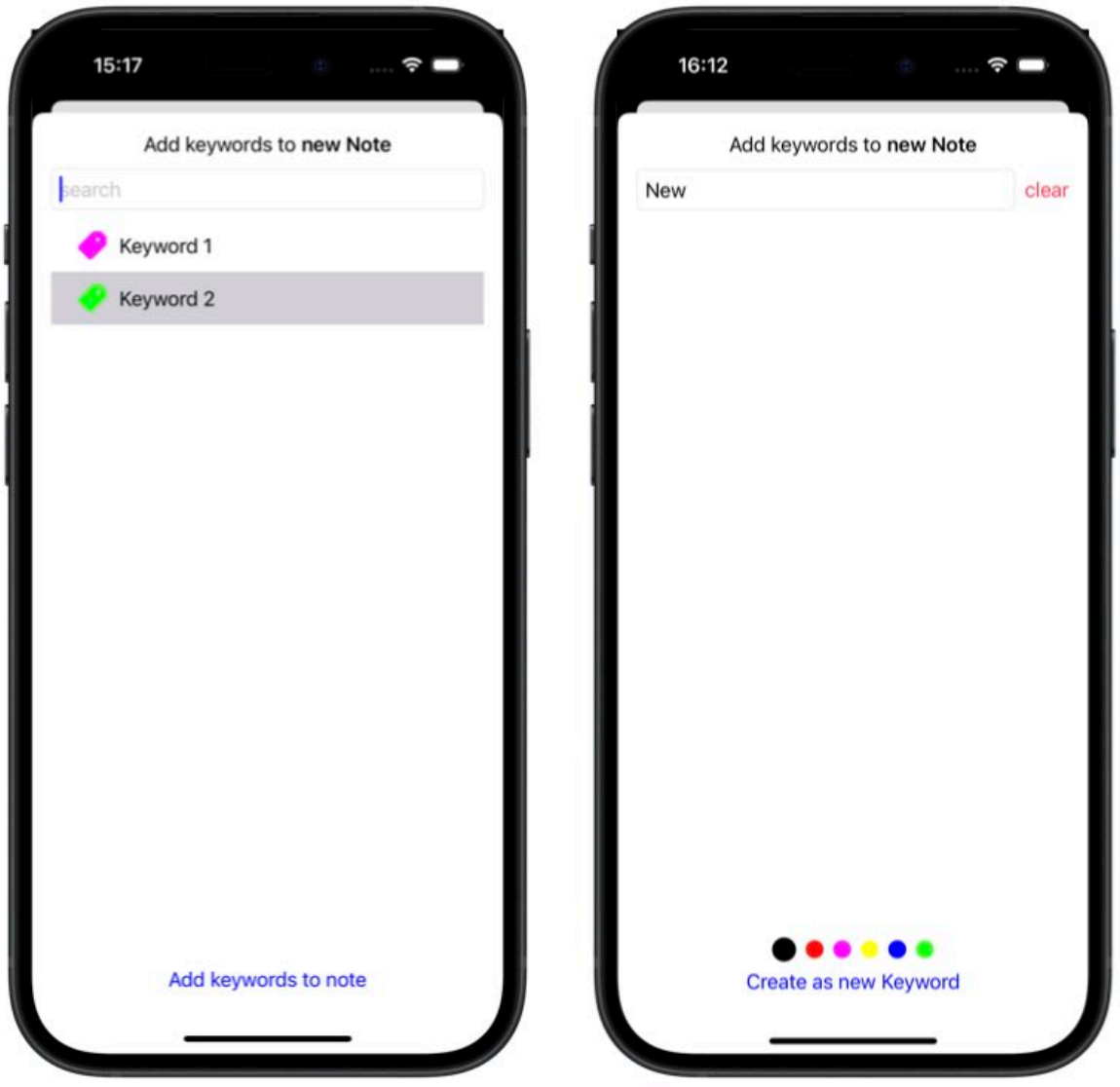
If the search term is not empty, I am creating a new predicate with the search term ad update the @FetchRequest for keywords.

Otherwise, I dont have any thing to search for and set the predicate to nil. Which means that all keywords are fetched.

Creating New Keywords

If the search doesn’t find any existing keywords, you should offer the option to create a new one. Here’s a button for that purpose which is only shown if no keyword is shown in the list:

```
if selectedKeywords.count > 0 {
    Button(“Add keywords to note”)
} else if keywords.isEmpty {
   NewKeywordView(note: note, searchTerm: searchTerm)
}
```



I extracted this logic in a new view, that handles creating new keywords:

```
struct NewKeywordView: View {

    let note: Note
    let searchTerm: String

    let colorOptions = [Color(red: 0, green: 0, blue: 0),
                        Color(red: 1, green: 0, blue: 0),
                        Color(red: 1, green: 0, blue: 1),
                        Color(red: 1, green: 1, blue: 0),
                        Color(red: 0, green: 0, blue: 1),
                        Color(red: 0, green: 1, blue: 0)]

    @Environment(\.managedObjectContext) var context
    @Environment(\.dismiss) var dismiss

    @State private var selectedColor = Color(red: 0, green: 0, blue: 0)

    var body: some View {
        VStack {
            HStack {
               // showing color options for the keyword tag
                ForEach(colorOptions, id: \.self) { color in
                    Circle().fill(color)
                        .frame(width: selectedColor == color ? 20 : 15)
                        .onTapGesture {
                            selectedColor = color
                        }
```

```
                    }
                }

                Button {
                    let key = Keyword(name: searchTerm, context: context)
                    key.color = selectedColor
                    note.keywords.insert(key)

                    dismiss()
                } label: {
                    Text("Create as new Keyword")
                }
                .disabled(searchTerm.count == 0)
            }
        }
    }
```

I also added 6 color circles, that the user can select from. This will be set to the color of the newly created keyword.

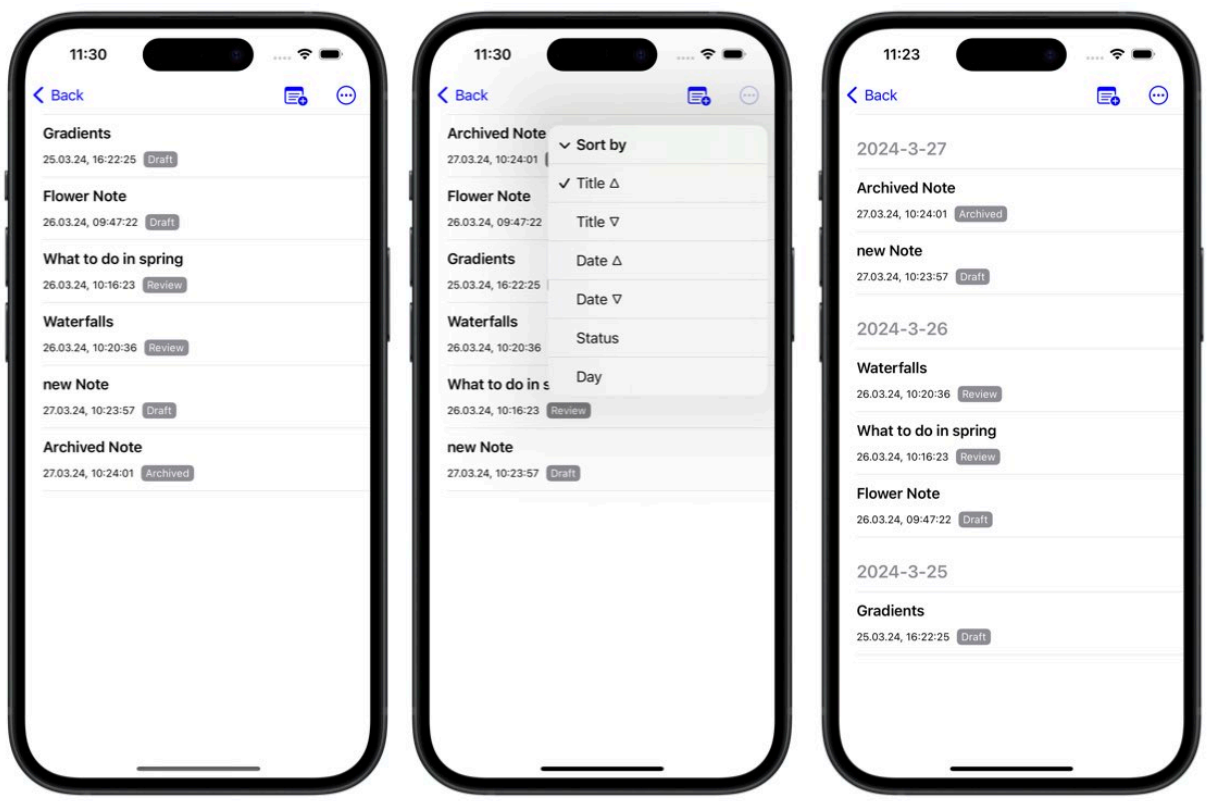
Afterwards, the keywords is added to the notes keyword list and the sheet is dismissed.

6. NOTES SORTING AND SEARCHING

6.1 INTRODUCTION TO SECTION

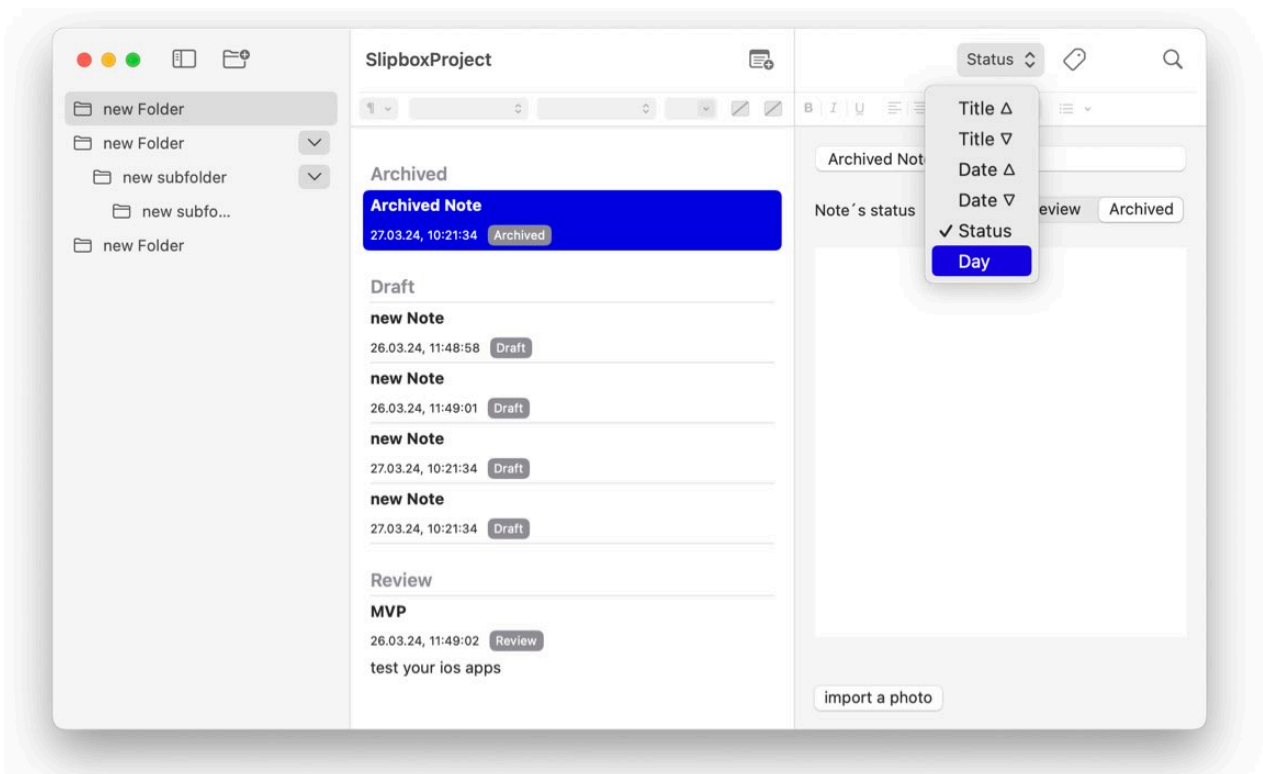
In this section, I'm going to dive into sorting. I'll use the note list as our example because when you're dealing with a multitude of notes, having robust search and filter options becomes essential.

To give you a clear picture of what we're going to tackle, you can see the iOS app below that uses a picker in the toolbar to switch between different sorting:

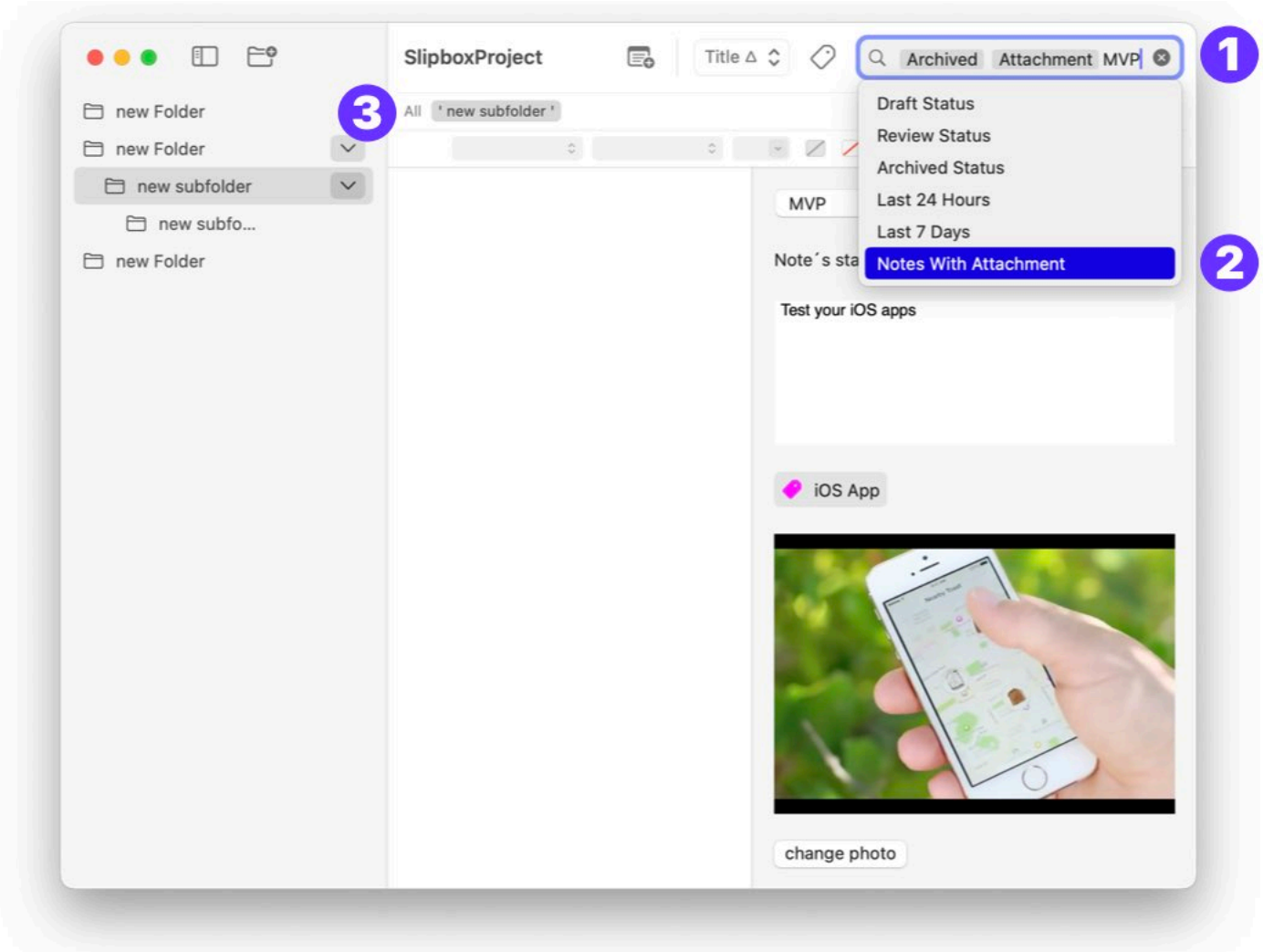


The sorting toggle presents six options. You can sort by title, either alphabetically or in reverse. By default, I've set it to sort by creation date. The last two options are more nuanced; they allow you to section your sorting. For instance, you can have sections for different days. Within each section, the notes are further sorted by time.

Mac users, in particular, expect a high level of sorting and filtering functionality. In the following screenshot, I sorted the notes injections by note status:



In addition to sorting, I want to incorporate filtering. Let's say you want to search through note titles or body text. I've prepared some dummy notes to illustrate this. You can add more filters, like showing only drafts or notes added in the last 24 hours. You can even filter by attachments.



The search functionality needs to be compatible with the sorted lists and work across all the options provided. At the bottom, you'll notice search suggestions. These are there to guide you in how you might want to search.

For a more advanced search, I'm using a newer SwiftUI feature with **search tokens (2**). This allows us to add multiple filter categories, greatly enhancing the app's functionality. We also have a **search scope (3).** On the toolbar, you'll see an additional line for search, where you can specify if you want to search within a particular folder or across all notes.

You might add more scopes, like child folders or specific tags, depending on your app's structure. This is what we call search scope – it defines the environment within which you want to search.

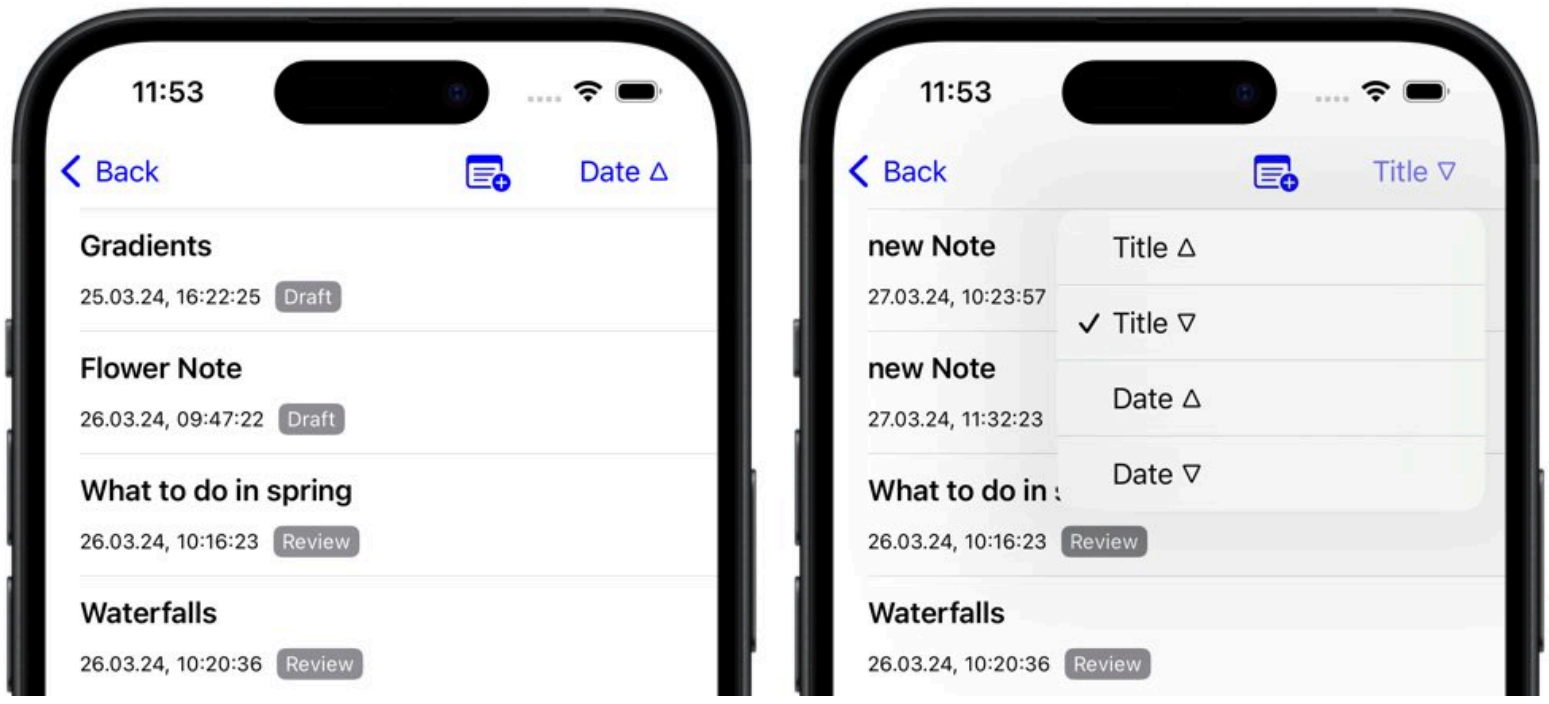
As you can see, we're stepping into a more complex level with a multitude of options for searching, filtering, and sorting. This complexity is necessary for our notes application, where robust search capabilities are crucial.

SwiftUI has introduced several new search features, like tokens and search scopes, which we'll explore. Some aspects of implementing these features are more about UX design than coding in SwiftUI, and I'll guide you through various implementations to give you a comprehensive understanding.

We'll ensure all these features work together smoothly, so you can display your notes in the most organized and user-friendly way possible.

6.2 SORTING NOTES BY TITLE OR DATE

In this section, I'll guide you through implementing sorting functionality for your notes. You'll learn how to create a picker that allows users to sort notes by various criteria, such as title or date.



Preview Data

To test sorting in previews, I create dummy notes with varying titles and dates. Here's a quick way to generate an array of dummy notes:

```
extension Note {
    …
    static func exampleArray(context: NSManagedObjectContext) -> [Note] {
        var notes = [Note]()
        let calendar = Calendar.current

        for index in 0..<10 {
            let newNote = Note(title: "note \(index)", context: context)
            newNote.creationDate_ = calendar.date(byAdding: .hour,
                                                  value: -(index * 10),
                                                  to: Date())

            if index > 6 {
                newNote.status = .review
            } else if index > 3 {
                newNote.status = .archived
            }

            notes.append(newNote)
        }

        return notes
    }
}
```

I use the example note to create a folder with these 10 notes:

```
extension Folder {
    …
    static func exampleWithNotes(context: NSManagedObjectContext) -> Folder {
        let folder = Folder(name: "my folder", context: context)

        let notes = Note.exampleArray(context: context)
        for note in notes {
            note.folder = folder
        }

        return folder
    }
}
```

The folder with notes can be used for the preview of NoteListView like so:

```
#Preview {
    let context = PersistenceController.preview.container.viewContext
    let folder = Folder.exampleWithNotes(context: context)

    return NavigationStack {
        NoteListView(selectedFolder: folder,
                     selectedNote: .constant(nil))
    }
    .environment(\.managedObjectContext, context)
}
```

I embedded the NoteListView in a NavigationStack, so that we will see the toolbar in the preview. Next I will add a sort picker in the toolbar.

Creating a Picker for Sorting

Now that we have the data for testing, let's move on to implementing the picker for sorting. I added a new file called "Sorting.swift" where I define an enum with cases for sorting by title, creation date, and status:

```
enum NoteSorting: CaseIterable, Identifiable {
    case titleAsc
    case titleDes
    case creationDateAsc
    case creationDateDes
    case status
    case day

    var id: Self { self }

    func title() -> String {
        switch self {
            case .titleAsc: return "Title △"
            case .titleDes: return "Title ▽"
            case .creationDateAsc: return "Date △"
            case .creationDateDes: return "Date ▽"
            case .status: return "Status"
            case .day: return "Day"
        }
    }
```

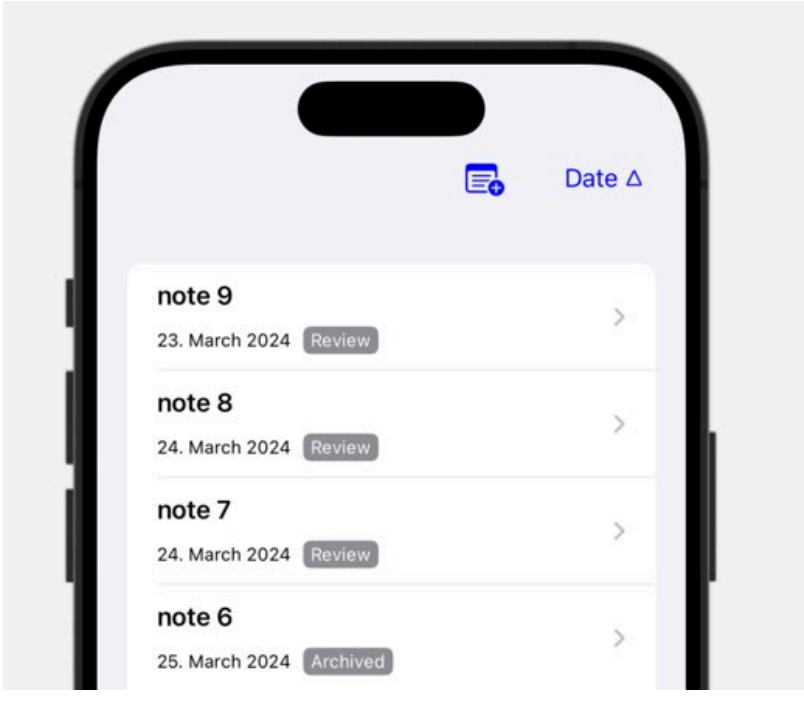
For convenience, I added a function “title” that generates a string, which I will use for the picker view.

In the NoteListView, below the toolbar item for adding a new note, I will add a picker with the label “Sort by”. I will also create a state property called **selectedNoteSorting** to keep track of the selected sorting option:

```
struct NoteListView: View {
    …
    @ObservedObject var selectedFolder: Folder
    @State private var selectedNoteSorting = NoteSorting.creationDateAsc

    var body: some View {
        List(selection: $selectedNote) {
             ForEach(selectedFolder.notes.sorted()) { note in
                …
            }
        }
        .toolbar {
            …

            ToolbarItem {
                Picker("Sort by",
                       selection: $selectedNoteSorting) {
                    ForEach(NoteSorting.allCases) { sorting in
                        Text(sorting.title())
                    }
                }
            }
        }
}
```

Next, let’s add an indicator to show the status of each note. In the NoteRow view, I will add a text element that displays the raw value of the note’s status. To make it more visually appealing, I will style the indicator with a smaller font size, white foreground color, and a gray background rectangle.

To make the layout more organized, I will wrap everything in a VStack. Additionally, I will add the timestamp next to the note’s title in an HStack:



```
struct NoteRow: View {

    @ObservedObject var note: Note

    var body: some View {
        VStack(alignment: .leading) {
            Text(note.title)
                .bold()

            HStack {
                Text(note.creationDate, style: .date)
                    .font(.caption)
                Text(note.status.rawValue)
                    .font(.caption)
                    .foregroundColor(.white)
                    .padding(.horizontal, 5)
                    .padding(.vertical, 2)
                    .background(
                        RoundedRectangle(cornerRadius: 5, style: .continuous)
                         .fill(Color.gray)
                    )
            }

            if note.bodyText.count > 0 {
                Text(note.bodyText)
                    .lineLimit(3)
            }
        }
        .tag(note)
    }
}
```

Sorting Notes in Core Data

Currently, I am showing the notes of the selected folder like so:

```
    List(selection: $selectedNote) {
        ForEach(selectedFolder.notes.sorted()) { note in
                    …
        }
    }
```

I could handle the sorting here and use the **sorted(by**) function for arrays. But this would handle the sorting on the array level. Instead, I would like Core Data to handle the sorting with fetch requests. This is much more efficient and memory-friendly. This approach is important if you have a large data set with complex searching and filtering.

First, I will need to create a fetch request that returns only notes for a specific folder. I add a function in the extension of Note, which you could also include in the Unit tests:

```
extension Note {
   …

   static func fetch(for folder: Folder) -> NSFetchRequest<Note> {
        let folderPredicate = NSPredicate(format: "%K == %@“,
                                          NoteProperties.folder,
                                          folder)
         return Note.fetch(folderPredicate)
    }
}
```

Next, I need to change NoteListView and use @FetchRequest property wrapper to fetch notes:

```
struct NoteListView: View {
     …

    init(selectedFolder: Folder, selectedNote: Binding<Note?>) {
        self.selectedFolder = selectedFolder
        self._selectedNote = selectedNote

        self._notes = FetchRequest(fetchRequest: Note.fetch(for: selectedFolder),
                                   animation: .bouncy)
    }

    @FetchRequest(fetchRequest: Note.fetch(.none))
    private var notes: FetchedResults<Note>

    var body: some View {
        List(selection: $selectedNote) {
            ForEach(notes) { note in
                NavigationLink(value: note) {
                    NoteRow(note: note)
                }
            }
        }
    }
```

In the initializer, I am updating the fetch request of notes to fetch all notes for the selected Folder. The retrieved notes are shown in the list view.

Showing the Sorted List

Now, that we have @FetchRequest of notes in place, I can update the sort descriptor, when the selectedNoteSorting state property changes. To do the updates, I am using onChange modifier:

```
struct NoteListView: View {

    …
    @FetchRequest(fetchRequest: Note.fetch(.none))
private var notes: FetchedResults<Note>
    @State private var selectedNoteSorting = NoteSorting.creationDateAsc

    var body: some View {
        …

        .onChange(of: selectedNoteSorting) { newValue in
            switch newValue {
                case .creationDateAsc:
                notes.nsSortDescriptors = [NSSortDescriptor(keyPath:
                                                \Note.creationDate, ascending: true)]
                case .creationDateDes:
                    notes.nsSortDescriptors = [NSSortDescriptor(keyPath:
                                               \Note.creationDate, ascending: false)]
                case .titleAsc:
                    notes.nsSortDescriptors = [NSSortDescriptor(keyPath:
                                                    \Note.title_, ascending: true)]
                case .titleDes:
                    notes.nsSortDescriptors = [NSSortDescriptor(keyPath:
                                                   \Note.title_, ascending: false)]
                case .status, .day:
                    // other cases omitted for brevity
                    return
            }
        }
    }
```

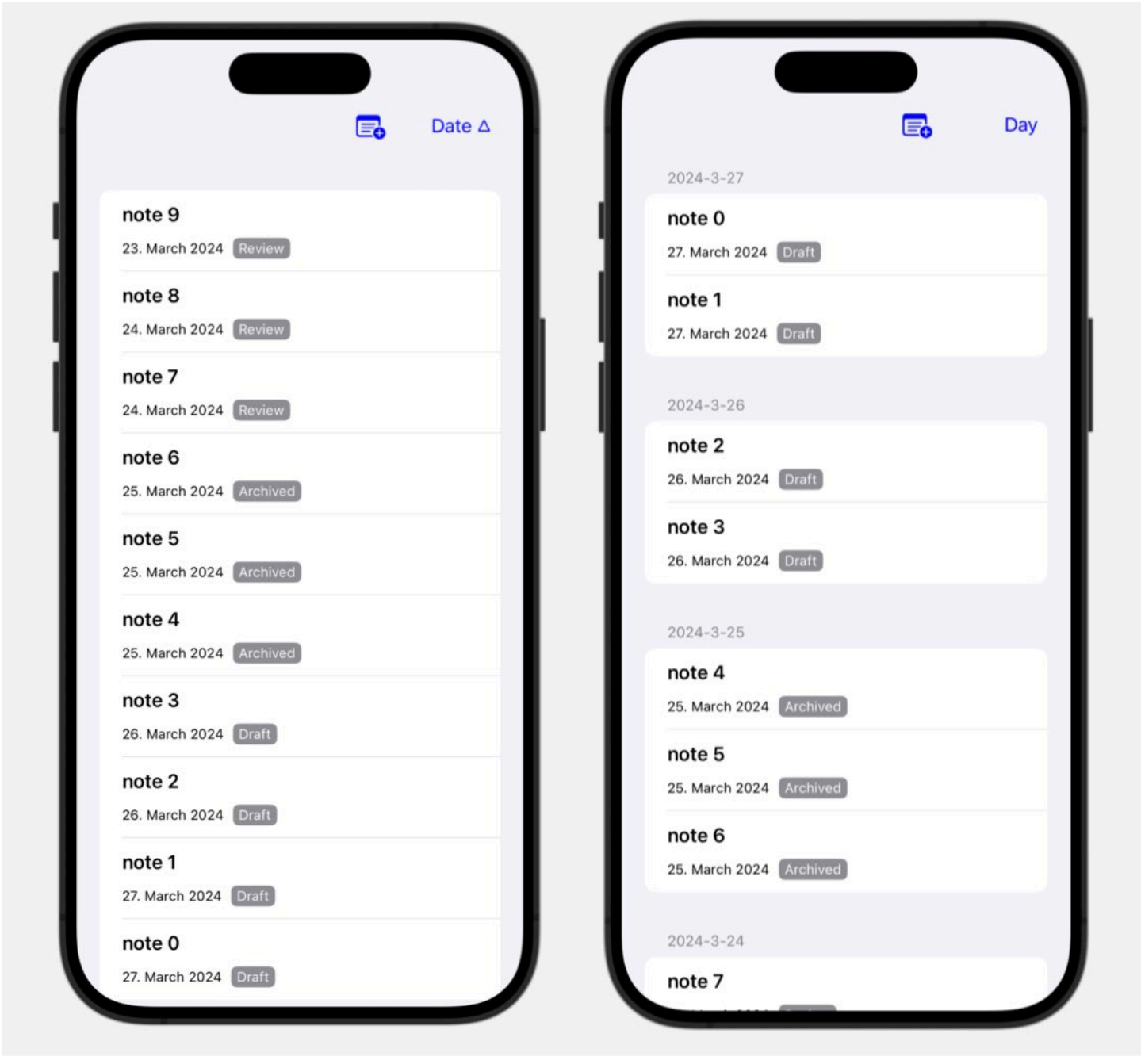
Depending on the selected sorting, I update the nsSortDesciptors property of notes. You need to specify the key path for the sorting property in the database. Make sure to use “title_” instead of “title”. Otherwise the app will crash.

I show sorting for title and creation date. In the next sections, you will learn about sectioning for sorting. You will show sorting by status and day with sectioning.

6.3 SECTIONEDFETCHREQUEST BY DAY

In this lesson, I'm going to tackle the challenge of sorting and sectioning our notes by day using Core Data and SwiftUI. We'll be implementing a SectionedFetchRequest that groups our notes based on the day they were created.

Down below, you can see the sorting by date on the left. On the right, you can see the notes sections. Each section has a header showing the day. For the preview data that you used in the last section, each section has 2 notes:



Creating a Computed Property for Day

First, we need to define a property we can section by. Since CoreData's **@SectionedFetchRequest** doesn't work well with custom types, I suggest using a string for simplicity. You might have an optional string attribute in your database, but SectionedFetchRequest won't like optional either. To handle this, we'll create a non-optional property.

You need to create a computed property that represents the day. Our Note entity has a creationDate attribute, which includes the time down to seconds. For our purposes, we want to group notes by the day, so we need to strip the time and only focus on the year, month, and day.

Here's how you create the computed property in the Note extension:

```
extension Note {
    …

   @objc var day: String {
        let components = Calendar.current.dateComponents([.year, .month, .day],
                                                         from: creationDate)
        return "\(components.year!)-\(components.month!)-\(components.day!)"
    }
}
```

I am formatting the resulting string as "Yeay-Month-Day" e.g. "2024-3-25".

Note to add **@objc** to make it compatible with Core Data and Objective-C.

Sorting with SectionedFetchRequest

Next, you need to set up your SectionedFetchRequest. This will use the day property to group notes and sort them accordingly. Create a new view "NotesSectionedByDayView". Use @SectionedFetchRequest to fetch notes:

```
struct NotesSectionedByDayView: View {

    @SectionedFetchRequest<String, Note>(
        sectionIdentifier: \.day,
        sortDescriptors: [SortDescriptor(\.creationDate_, order: .forward)]
    )

    var body: some View {
       …
    }
}
```

You need to specify the sectionIdentifier key path. In the above case, I set the identifier to "day", which will use the computed property that we just define

I am passing the type of the section identifier as string for the Note class like:

```
@SectionedFetchRequest<String, Note>
```

Lastly, I set the sorting by creation date in forward order. This is used to sort the notes within each section.

Showing Notes for the Selected Folder

I want to use this sectioned list together with the folder to notes list. Therefor, I need to change the fetch request to only show the notes that belong to the selected folder. In the following the predicate for the fetch request is set in the initializer of the view:

```
struct NotesSectionedByDayView: View {

    init(selectedFolder: Folder) {
        self.selectedFolder = selectedFolder

        let request = Note.fetch(for: selectedFolder)
        self._notesSections = SectionedFetchRequest(fetchRequest: request,
                                                    sectionIdentifier: \.day)
    }

    var selectedFolder: Folder

    @SectionedFetchRequest<String, Note>(
        sectionIdentifier: \.day,
        sortDescriptors: [SortDescriptor(\.creationDate_, order: .forward)]
    )
    private var notesSections: SectionedFetchResults<String, Note>

    var body: some View {
       …
    }
}
```

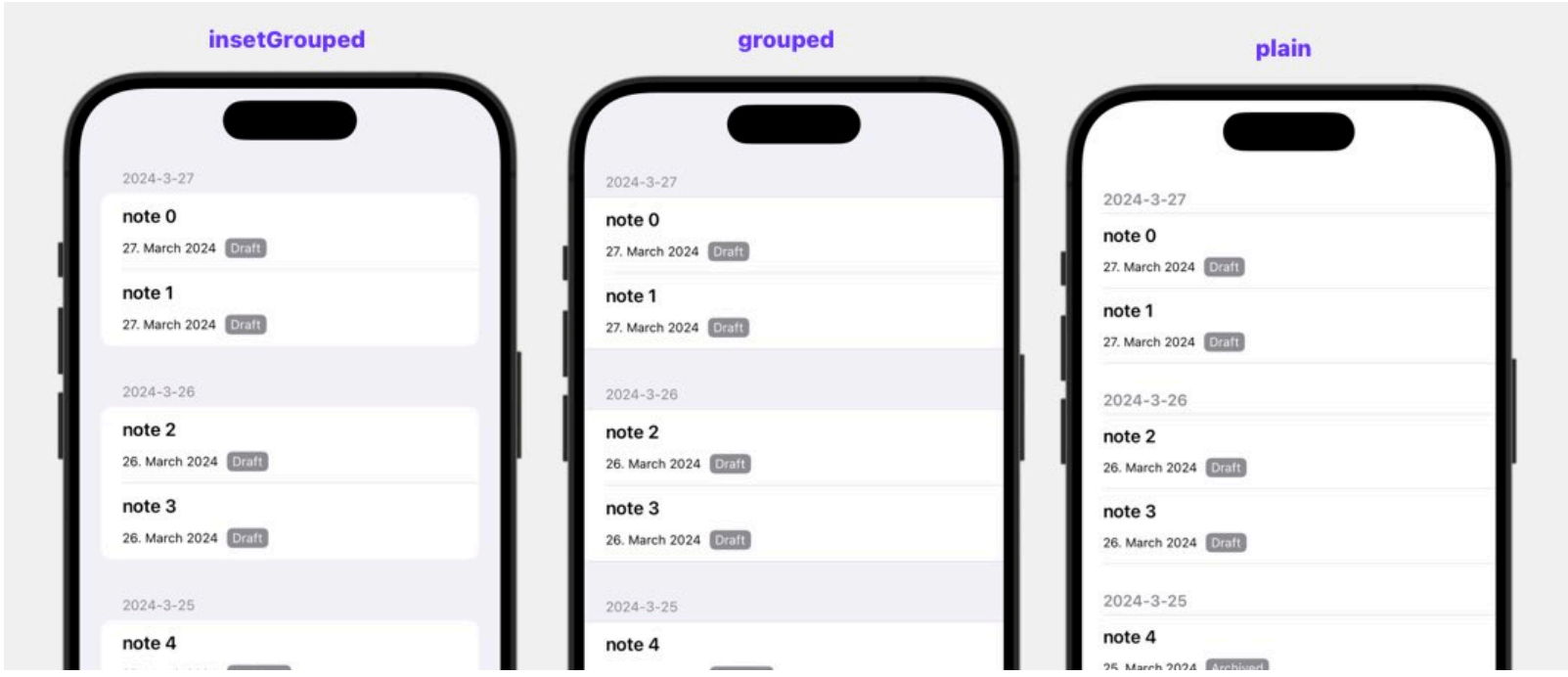
To display the notes in a list, grouped by day, you can iterate over **notesSections** and create sections in your SwiftUI view:

```
List {
  ForEach(sectionedNotes) { section in
    Section(section.id) {
        ForEach(section) { note in
            NoteRow(note: note)
        }
    }
  }
}
```

To test this in the preview canvas update the preview to use the example data:

```
#Preview {
    let context = PersistenceController.preview.container.viewContext
    let folder = Folder.exampleWithNotes(context: context)

    return NotesSectionedByDayView(selectedFolder: folder,
                                   selectedNote: .constant(nil))
    .environment(\.managedObjectContext, context)
}
```

List per default will show the sections in an insetGroup style. You can also show the list as grouped or plain:



Customizing Date Display

If you want to format the date differently, adjust the computed property day to return the date string in the desired format. For example, if you don't want the dashes:

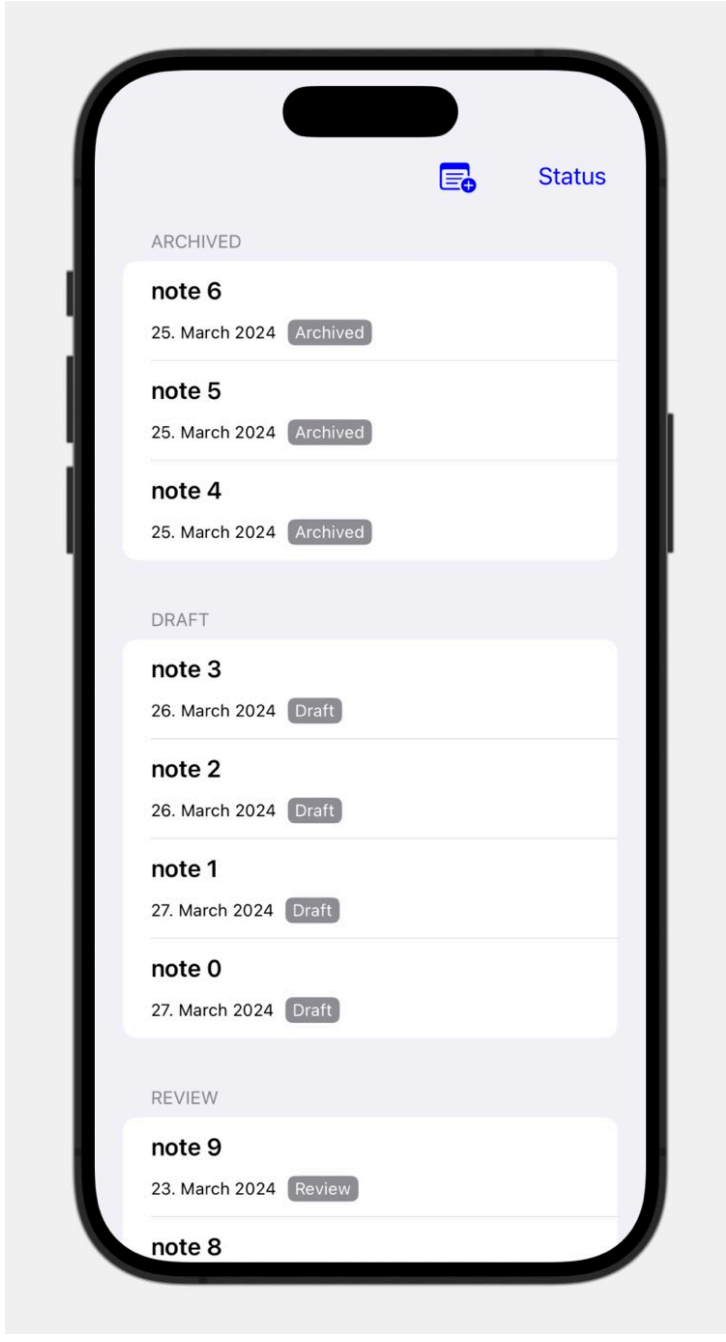
```
extension Note {
    …

   @objc var day: String {
        let components = Calendar.current.dateComponents([.year, .month, .day],
                                                         from: creationDate)
        return "\(components.year!) \(components.month!) \(components.day!)"
    }
}
```

Handling Large Datasets

One important note is that SectionedFetchRequest can have performance implications. If you're working with a large dataset, or the computation of section values is complex, you might want to use a stored property in the database instead. Derived properties could also be a better option for performance.

6.4 SECTIONEDFETCHREQUEST BY BY STATUS

In this lesson, I'm going to show you how to create a sectioned list based on the status of our notes. We'll have separate sections for notes in 'Draft', 'Archived', and 'In Review' status. To achieve this, we need to create a property that we can section by.



Creating a Computed Property for Status

Since CoreData's SectionedFetchRequest isn't friendly with custom types, I'll use a String to represent the status. The attribute status_ in the database is a String, but it's optional, and SectionedFetchRequest doesn't work well with optional.

To handle the optional, I create a computed property:

```
extension Note {
    …

    @objc var sectionStatus: String {
        status_ ?? Status.review.rawValue
    }
}
```

Sorting with SectionedFetchRequest

Now, let's create a SwiftUI view, NoteSectionedByStatusView, that will use this property to display notes in sections.

```
struct NotesSectionedByStatusView: View {

    init(selectedFolder: Folder) {
        self.selectedFolder = selectedFolder

        let request = Note.fetch(for: selectedFolder)
        self._notesSections = SectionedFetchRequest(fetchRequest: request,
                                                    sectionIdentifier: \.sectionStatus)
    }

    @SectionedFetchRequest<String, Note>(
        sectionIdentifier: \.sectionStatus,
        sortDescriptors: [SortDescriptor(\.status_)]
    )
    private var sectionedNotes: SectionedFetchResults<String, Note>

    var body: some View {
        List {
            ForEach(sectionedNotes) { section in
                Section(section.id) {
                    ForEach(section) { note in
                        NoteRow(note: note)
                    }
                }
            }
        }
    }
```

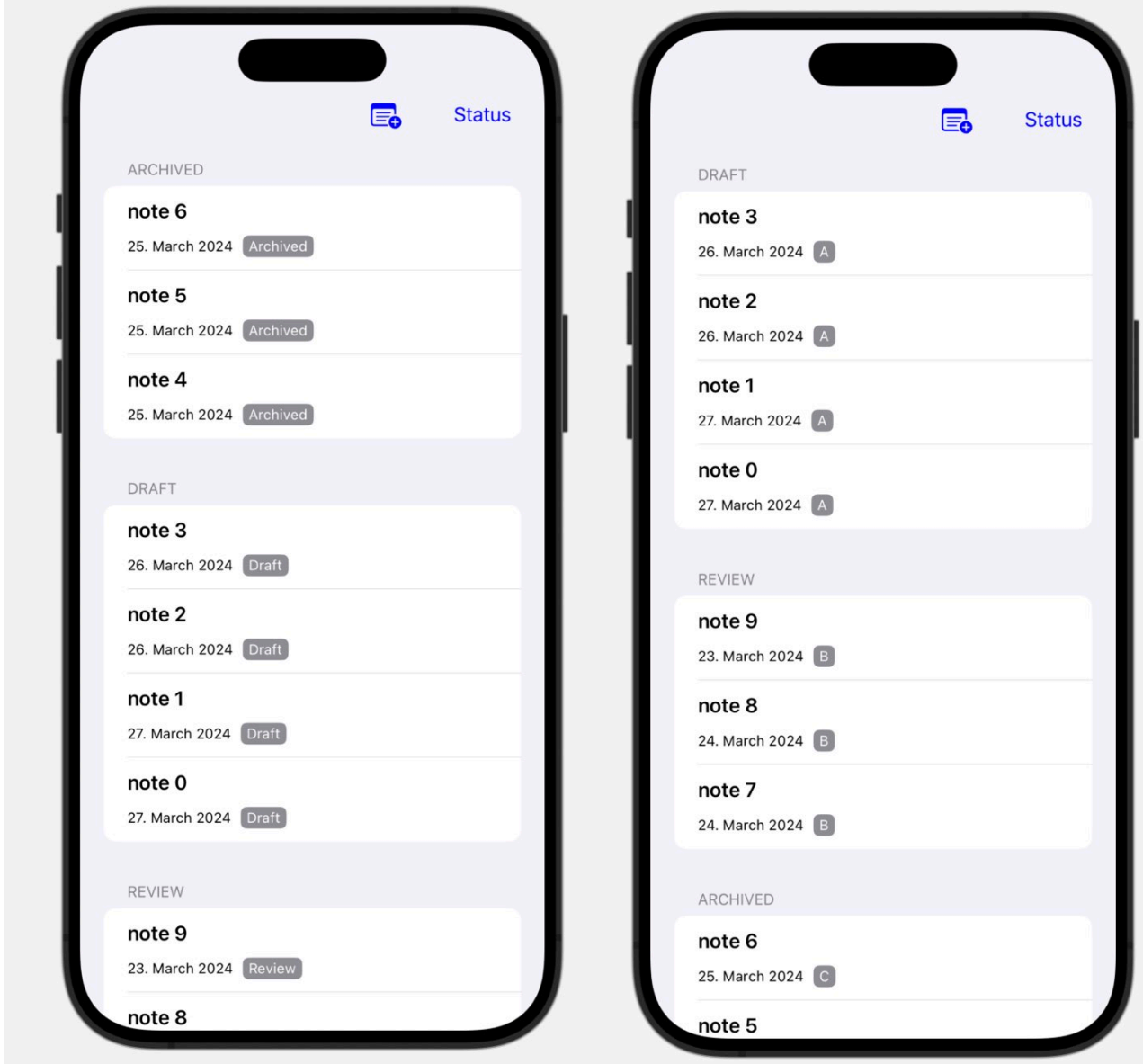
This is very similar to the previous section where I sectioned by day. The only difference is that I set the section identifier to "sectionStatus"

Changing the Sorting of the sections

One issue you might encounter is the alphabetical sorting of the sections themselves. The are shown in the order "Archived", "Draft" and "Review". I would prefer the order "Draft", "Review" and Archived".

If you want a specific order that isn't alphabetical, you'll have to work around this limitation. One way is to change the string values in your database to something that sorts alphabetically in the order you want, like "A", "B", and "C" instead of "Draft", "Review", and "Archived".

In the image below, you see that on the right the sort order is changed to "Draft", "Review", and last "Archived":



Here is how you can update the Status enum that uses String:

```
enum Status: String, Identifiable, CaseIterable {

    case draft = "A"
    case review = "B"
    case archived = "C"

    var id: String {
        return self.rawValue
    }

    var displayName: String {
        switch self {
            case .draft: return "Draft"
            case .review: return "Review"
            case .archived: return "Archived"
        }
    }
}
```

I also included a “displayName” property that I use to show the title for each section:

```
    struct NotesSectionedByStatusView: View {
        …
        var body: some View {
                ForEach(sectionedNotes) { section in
                    Section(sectionName(id: section.id)) {
                        ForEach(section) { note in
                            NoteRow(note: note)
                        }
                    }

                }
        }

        func sectionName(id: String) -> String {
            let status = Status(rawValue: id)
            return status?.displayName ?? Status.draft.displayName
        }
    }
```

However, if you’re in the middle of development or have existing users, changing the data model isn’t ideal. You could lose data, and it’s more of a hack than a solution.

To overcome this limitation, I suggest adding a custom way of sectioning. Instead of using a single SectionedFetchRequest, you can write three separate fetch requests, each with a different predicate for the status you’re looking for. This approach is more work, but it gives you complete flexibility.

6.5 SECTIONING WITH MULTIPLE FETCH REQUESTS

In this section, I'll guide you through an alternative approach for sectioning your data in SwiftUI when using Core Data. Instead of a single-sectioned fetch request, you can use multiple fetch requests with specific predicates for each section. This can be particularly useful when you need to display different data statuses or categories in separate sections.

Creating Multiple Fetch Requests

When you want to section your data, you'll need to set up distinct fetch requests for each section. Each fetch request will have its own predicate to filter the data accordingly. Let's say you have a list of notes with different statuses (e.g., archived, draft, etc.), and you want to display each status in its section. You would create a separate fetch request for each status.

Here's an example of how you can set up these fetch requests:

```
extension Note {
    …

    static func fetch(for folder: Folder,
                      status: Status? = nil) -> NSFetchRequest<Note> {
        let folderPredicate = NSPredicate(format: "%K == %@",
                                          NoteProperties.folder,
                                          folder)
        if let status = status {
            let statusPredicate = NSPredicate(format: "%K == %@",
                                              NoteProperties.status,
                                              status.rawValue as CVarArg)
            let predicate = NSCompoundPredicate(andPredicateWithSubpredicates:
                                                    [folderPredicate, statusPredicate])
            return Note.fetch(predicate)
        } else {
            return Note.fetch(folderPredicate)
        }
    }
}
```

In this code snippet, I first create a predicate to filter notes by the folder they belong to. Then, if a status is provided, I create another predicate for filtering by status and combine them using NSCompoundPredicate. This gives me a fetch request tailored to a specific folder and status.

Implementing the Sectioned View

Once you have your fetch requests set up, you can use them to create a sectioned view. For each section, you'll create a subview that uses its fetch request.

First, create a subview that takes in the selectedFolder and status as an argument an uses these to create the fetch request with the corresponding filtering:

```
    private struct SectionedView: View {

        init(selectedFolder: Folder, status: Status, title: String) {
            self.title = title
            self._notes = FetchRequest(fetchRequest: Note.fetch(for: selectedFolder,
                                                                status: status))
        }

        @FetchRequest(fetchRequest: Note.fetch(.none))
                        private var notes: FetchedResults<Note>
        let title: String

        var body: some View {
            Section(title) {
                ForEach(notes) { note in
                    NoteRow(note: note)
                }
            }
        }
    }
```

In the NoteListView, I show a list with a ForEach of all cases for the Status enum. For each case, I show a SectionedView that will filter for the individual status:

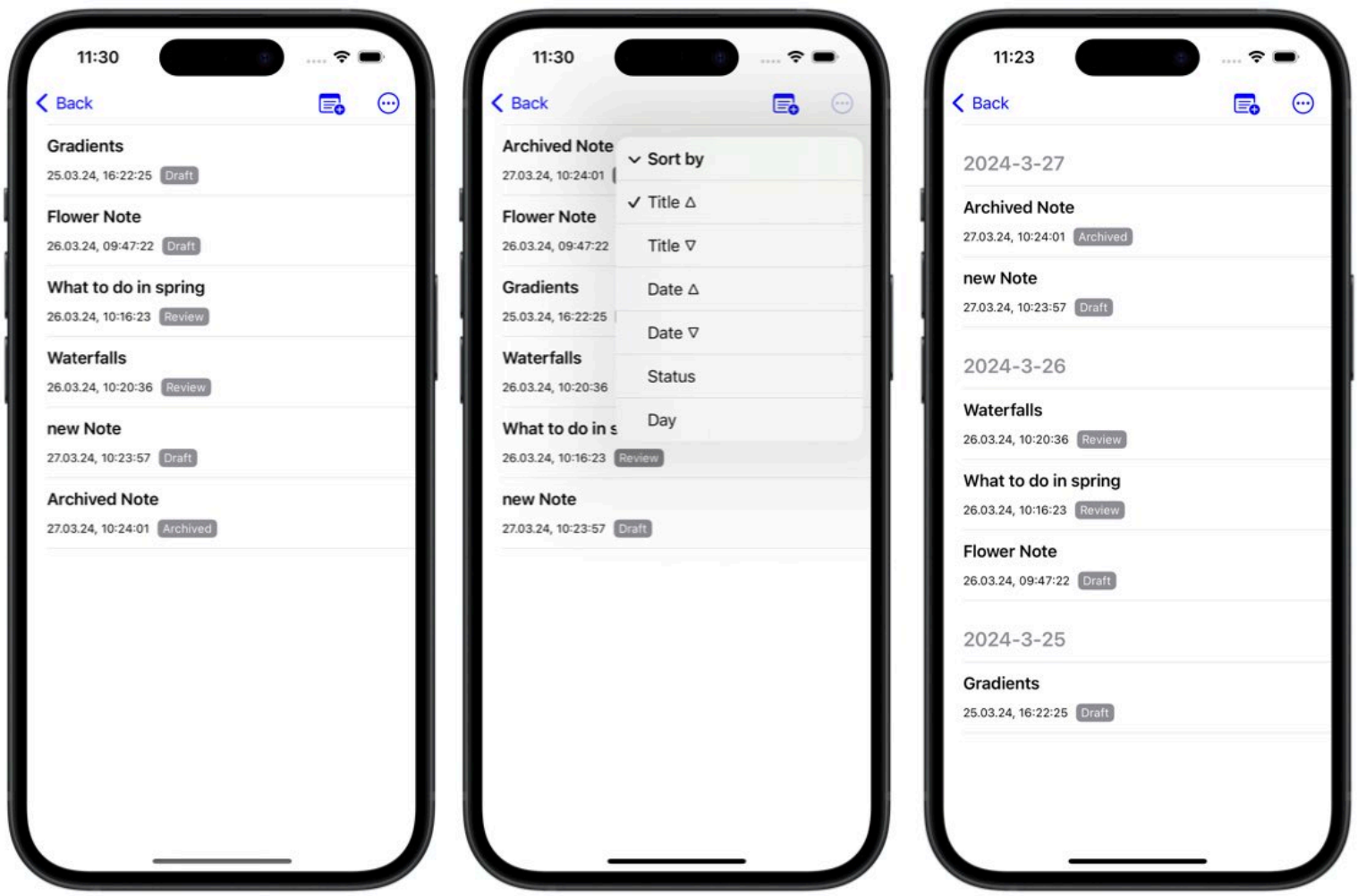
```
    struct NotesSimpleSectionedByStatusView: View {

        let selectedFolder: Folder

        var body: some View {
          List {
             ForEach(Status.allCases) { status in
                SectionedView(selectedFolder: selectedFolder,
                              status: status,
                              title: status.rawValue)
             }
          }
        }
    }
```

The order of the section is defined by the order of “allCases” which is defined by the order of the cases defined in the Status enume:

```
    enum Status: String, Identifiable, CaseIterable {

        case draft = "Draft"
        case review = "Review"
        case archived = "Archived"

        var id: String {
            return self.rawValue
        }
    }
```

6.6 COMBINING THE DIFFERENT SORTING VIEWS

So far, in the app you can only sort by title or creation date. I also want to show the sectioned sorting for status and day.



As you can imaging this is a more complex data flow, where we have to update and show the list differently.

First, let's update NoteListView. Depending on the selected sorting, I want to show different styles in the view:

```
struct NoteListView: View {

    let selectedFolder: Folder
    @Binding var selectedNote: Note?

    var body: some View {
        List(selection: $selectedNote) {
            switch selectedNoteSorting {
                case .titleAsc, .titleDes, .creationDateAsc, .creationDateDes:
                   // show sorting with @FetchRequest
                case .status:
                     NotesSectionedByStatusView(selectedFolder: selectedFolder)
                case .day:
                     NotesSectionedByDayView(selectedFolder: selectedFolder)
            }
        }
```

```
            .toolbar {
                 …

                ToolbarItem {
                    Picker("Sort by", selection: $selectedNoteSorting.animation()) {
                        ForEach(NoteSorting.allCases) { sorting in
                            Text(sorting.title())
                        }

                    }
                }
            }
        }
    }
```

In the case of “status” I can reuse the existing view “NotesSectionedByDayView”. Since the List is already implemented in NoteListView, I only make a small change and remove it from “NotesSectionedByDayView”:

```
    struct NotesSectionedByDayView: View {
       …
       var body: some View {
            ForEach(notesSections) { section in
                Section(section.id) {
                    …
                }
            }
        }
    }
```

Similarly, I would update “NotesSectionedByStatusView”:

```
    struct NotesSectionedByStatusView: View {
        …

        var body: some View {
                ForEach(sectionedNotes) { section in
                    Section(section.id) {
                    …
                }
        }
    }
```

Extracting the Sorting by Date and Title into a Separate Subview

Lastly, I am extracting the note list for sorting by title or creation date into a new subview "NoteSingleSortedView":

```
    struct NoteSingleSortedView: View {

        init(selectedFolder: Folder) {
            self.selectedFolder = selectedFolder
            self._notes = FetchRequest(fetchRequest: Note.fetch(for: selectedFolder))
        }

        @FetchRequest(fetchRequest: Note.fetch(.none))
        private var notes: FetchedResults<Note>

        var body: some View {
             ForEach(notes) {
                 NoteRow(note: $0)
            }
        }
    }
```

This subview uses the selected folder to filter the notes. I can then show "NoteSingleSortedView" for the case of sorting by either title or creation date:

```
    List(selection: $selectedNote) {
        switch selectedNoteSorting {
            case .titleAsc, .titleDes, .creationDateAsc, .creationDateDes:
                NoteSingleSortedView(selectedFolder: selectedFolder)
            case .status:
                NotesSectionedByStatusView(selectedFolder: selectedFolder)
            case .day:
                NotesSectionedByDayView(selectedFolder: selectedFolder)
        }
    }
```

Now I am able to switch between sorting by title, status or day. But I need to also distinguish between the cases of title ascending / descending and creation date. I want to handle this in the initialiser of "NoteSingleSortedView". I pass the selected note sorting:

```
    struct NoteSingleSortedView: View {

      init(selectedFolder: Folder, noteSorting: NoteSorting) {
        let request = Note.fetch(for: selectedFolder)
        let defaultSorting = NSSortDescriptor(keyPath: \Note.creationDate_,
                                              ascending: true)
        switch noteSorting {
            case .creationDateAsc:
                request.sortDescriptors = [defaultSorting]
            case .creationDateDes:
                request.sortDescriptors = [NSSortDescriptor(keyPath: \Note.creationDate_,
                                                            ascending: false)]
            case .titleAsc:
                request.sortDescriptors = [NSSortDescriptor(keyPath: \Note.title_,
                                                           ascending: true),
```

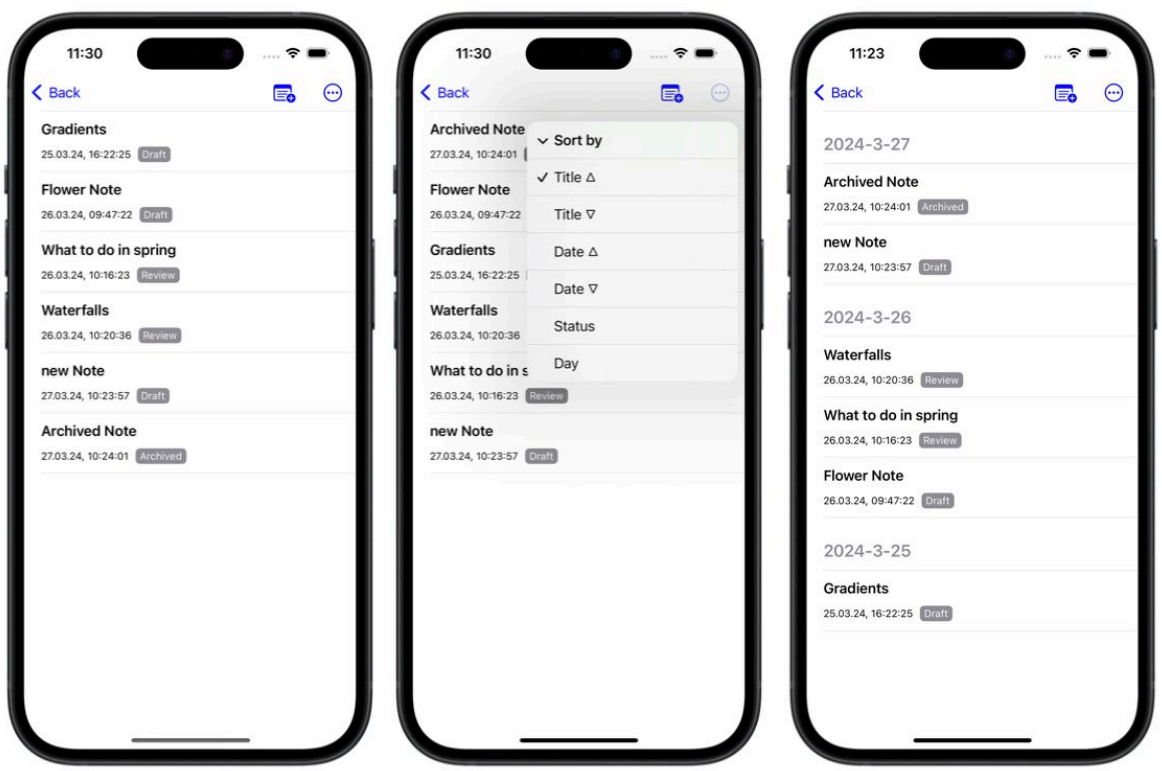
```
                                            defaultSorting]
            case .titleDes:
                request.sortDescriptors = [NSSortDescriptor(keyPath: \Note.title_,
                                                            ascending: false),
                                           defaultSorting]
            case .status, .day:
                return
        }

        self._notes = FetchRequest (fetchRequest: request, animation: .bouncy)
    }

    @FetchRequest(fetchRequest: Note.fetch(.none))
    private var notes: FetchedResults<Note>

    …
}
```

This is the same code that was previously handled in the onChange in NoteListView.

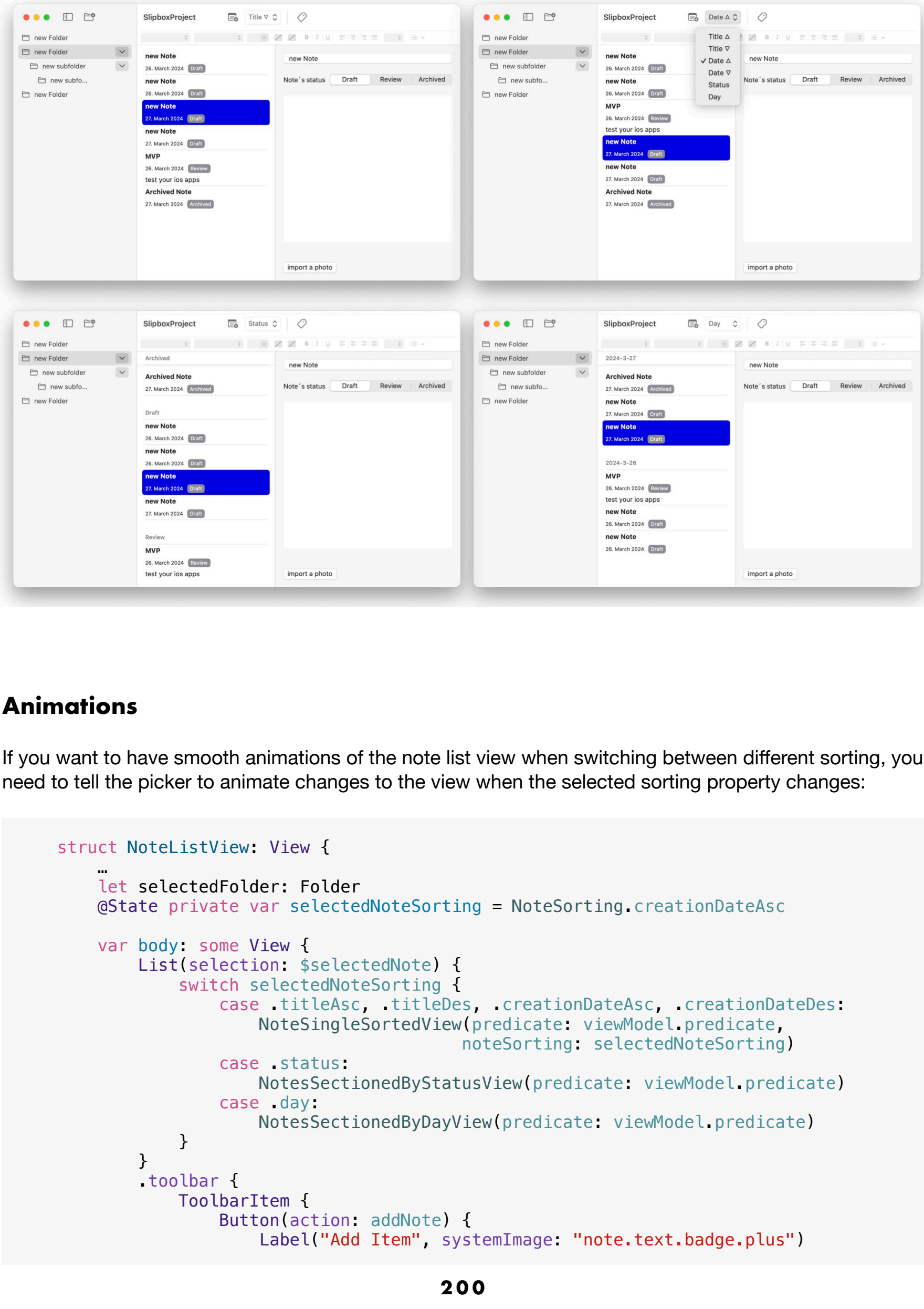
Now I can update the body of NoteListView to pass all necessary parameters like selected folder and sorting to the subviews:

```
    List(selection: $selectedNote) {
        switch selectedNoteSorting {
            case .titleAsc, .titleDes, .creationDateAsc, .creationDateDes:
                NoteSingleSortedView(selectedFolder: selectedFolder,
                                     noteSorting: selectedNoteSorting)
            case .status:
                NotesSectionedByStatusView(selectedFolder: selectedFolder)
            case .day:
                NotesSectionedByDayView(selectedFolder: selectedFolder)
        }
}
```

Build and run the project, you should be able to switch between the different sorting cases:



Similarly, this should also work for macOS:



Animations

If you want to have smooth animations of the note list view when switching between different sorting, you need to tell the picker to animate changes to the view when the selected sorting property changes:

```
struct NoteListView: View {
    …
    let selectedFolder: Folder
    @State private var selectedNoteSorting = NoteSorting.creationDateAsc

    var body: some View {
        List(selection: $selectedNote) {
            switch selectedNoteSorting {
                case .titleAsc, .titleDes, .creationDateAsc, .creationDateDes:
                    NoteSingleSortedView(predicate: viewModel.predicate,
                                         noteSorting: selectedNoteSorting)
                case .status:
                    NotesSectionedByStatusView(predicate: viewModel.predicate)
                case .day:
                    NotesSectionedByDayView(predicate: viewModel.predicate)
            }
        }
        .toolbar {
            ToolbarItem {
                Button(action: addNote) {
                    Label("Add Item", systemImage: "note.text.badge.plus")
```

```
                    }
                }

                ToolbarItem {
                    Picker("Sort by", selection: $selectedNoteSorting.animation()) {
                        ForEach(NoteSorting.allCases) { sorting in
                            Text(sorting.title())
                        }
                    }
                }
            }
        }
```

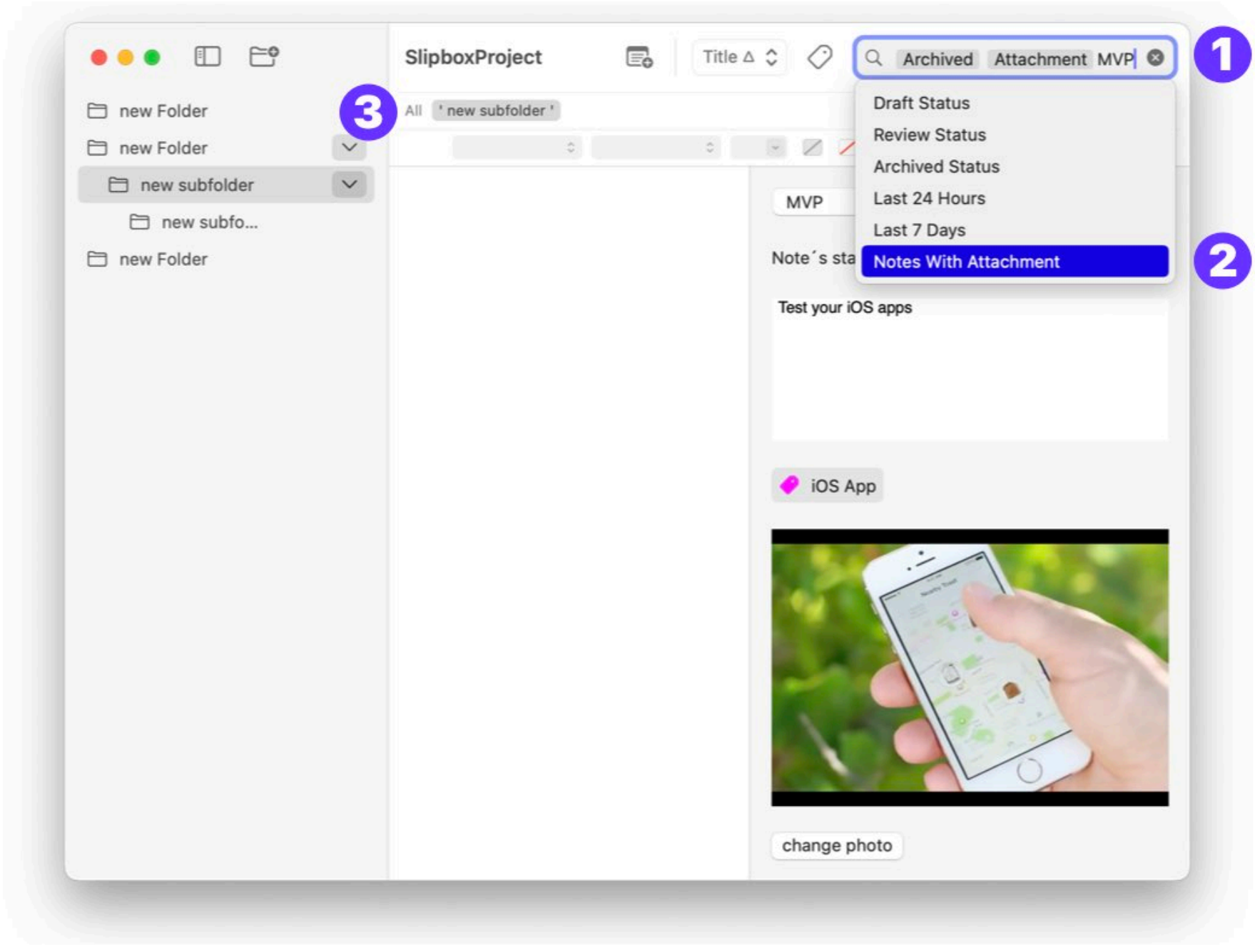
For previous iOS and macOS versions, some of these animations did not show. For example, going from @FetchRequest to @SectionedFetchRequest. For iOS 17 and macOS 14, they seem to work smoothly and stable.

6.7 SEARCHABLE VIEW MODIFIER, TOKENS AND SCOPE

In this section, we're going to tackle the implementation of a search feature for notes. SwiftUI provides us with a range of UI elements, some of which are relatively new as of iOS 16 and macOS 13. You'll learn how to utilize these elements to create a user-friendly search interface.

Here are the 3 components you will learn about:

1. Search text field
2. Search tokens
3. Search scope



Adding a Searchable Modifier

First, let's add a search text field to our NoteListView. This is achieved by using the searchable view modifier. Here's how you can do it:

```
    struct NoteListView: View {
        …

        @State private var searchText: String = “”

        var body: some View {
            List(selection: $selectedNote) {
                …
            }
            .searchable(text: $searchText)
    }
```

When you add the .searchable modifier and run the live preview, you'll see the search text field appear. If you tap on it, the toolbar will collapse, allowing you to type your query. Tapping 'Cancel' will dismiss the search and bring back the full toolbar.

Search Suggestions

The searchable modifier also allows you to provide search suggestions and tokens. Let's add some static suggestions as an example:

```
.searchSuggestions {
     Text(“Hello")
       .searchCompletion(“hello”)
     Text("World")
 }
```

Tapping on the search field now displays these suggestions.

Search Tokens

To provide more advanced filtering options, let's define an enum for our search tokens:

```
    enum NoteSearchToken: CaseIterable, Identifiable {

        case draftStatus
        case reviewStatus
        case archivedStatus
        case last24Hours
        case last7Days
        case withAttachment
```

```
        var id: Self { self }

        var name: String {
            switch self {
                case .draftStatus:
                    return "Draft"
                case .reviewStatus:
                    return "Review"
                case .archivedStatus:
                    return "Archived"
                case .last24Hours:
                    return "24h"
                case .last7Days:
                    return "7d"
                case .withAttachment:
                    return "Attachment"
            }
        }

        var fullName: String {
            switch self {
                case .draftStatus:
                    return "Draft Status"
                case .reviewStatus:
                    return "Review Status"
                case .archivedStatus:
                    return "Archived Status"
                case .last24Hours:
                    return "Last 24 Hours"
                case .last7Days:
                    return "Last 7 Days"
                case .withAttachment:
                    return "Notes With Attachment"
            }
    }
```

I added cases to search for notes with draft, review or archived status or notes that have an attachement image. You can also show a search option for a certain time limit e.g. notes that were created on the last day or last week.

For convenience, I added computed properties that give me a text that I can show in the UI as search suggestions. One short and one long form.

Create a state property to hold the selected tokens “searchTokens”. Because the user might want to pick multiple filter options, this property is an array of **NoteSearchToken.** You will use them later to filter the note list.

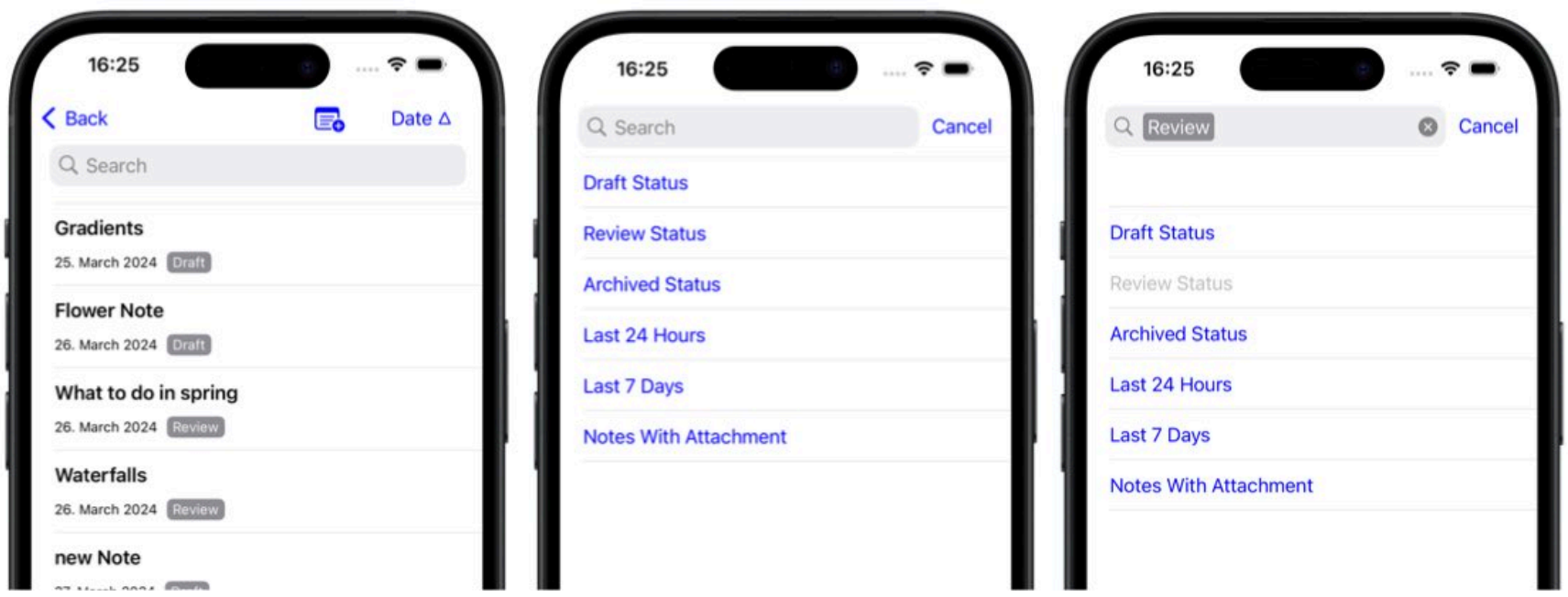
```
    struct NoteListView: View {
        …

        @State private var searchText: String = ""
        @State private var searchTokens = [NoteSearchToken]()

        var body: some View {
          …
       }
    }
```

Now, we can use these tokens with the **searchable** modifier:

```
struct NoteListView: View {
    …

    @State private var searchText: String = ""
    @State private var searchTokens = [NoteSearchToken]()

    var body: some View {
        List(selection: $selectedNote) {
            …
        }
        .searchable(text: $searchText,
                    tokens: $searchTokens,
                    token: { token in
            Text(token.name)
        })
    }
}
```

If pull on the note list screen, the search bar will appear. Tap on it and the search tokens should be shown. Tap on a token and it is added inside the search text field. If your descriptions are very long, they might fill up the text field. You can instead pass a short version of the token name that will be shown in the text field.

In the below scree, I selected “Review Status” from the suggestions list. In the text field a shorter version is shown “Review”



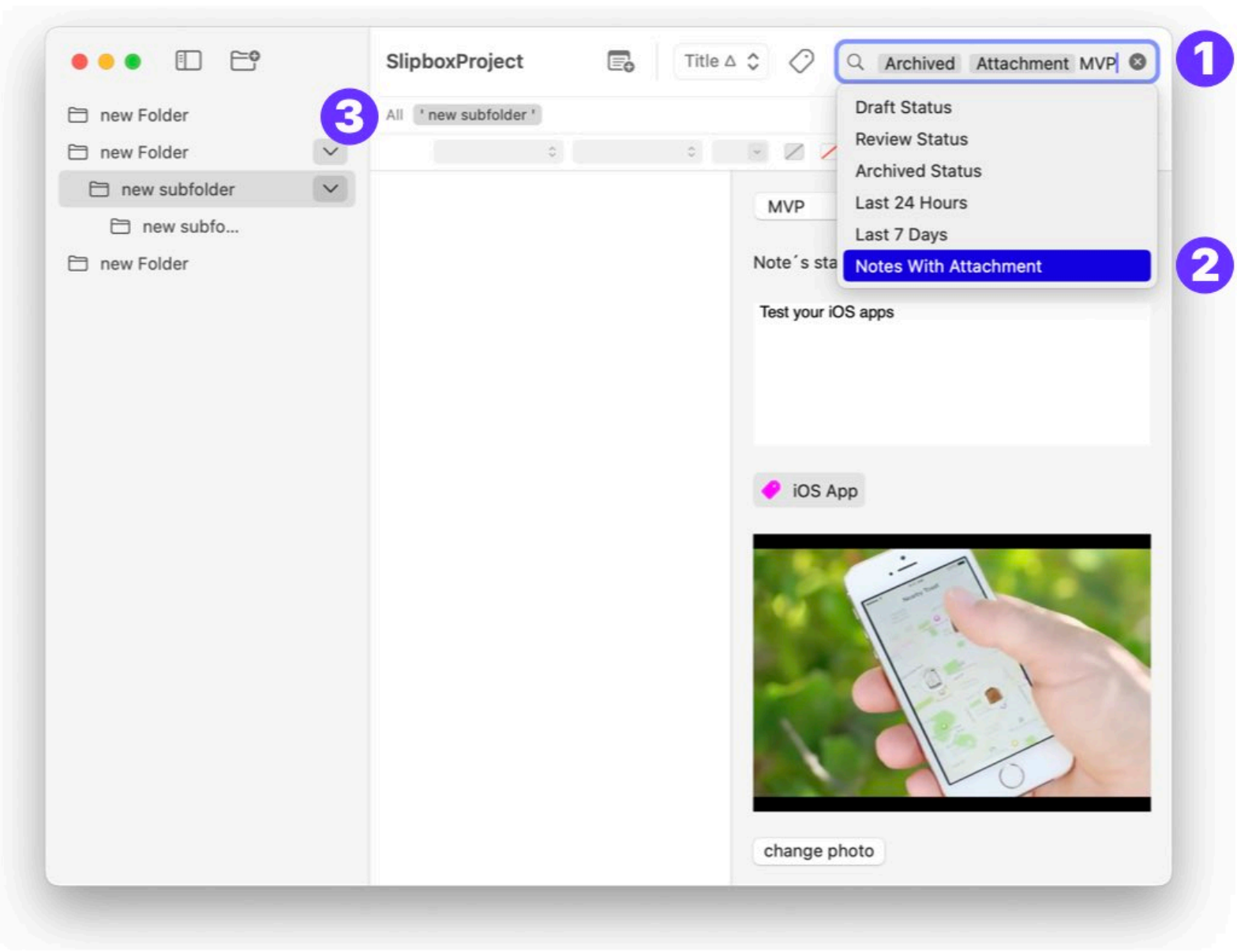
To have 2 different text versions shown for the suggestion list and tag in the text field, use a search suggestion modifier like so:

```
struct NoteListView: View {
    …

    @State private var searchText: String = ""
    @State private var searchTokens = [NoteSearchToken]()
```

```
        var body: some View {
            List(selection: $selectedNote) {
                …
            }
            .searchable(text: $searchText,
                        tokens: $searchTokens,
                        token: { token in
                Text(token.name)
            })
            .searchSuggestions({
                ForEach(NoteSearchToken.allCases) { token in
                    Button {
                        searchTokens.append(token)
                    } label: {
                        Text(token.fullName)
                    }
                }
            })
    }
```

SwiftUI Search Scope

Sometimes, you might want to provide **(3) search scope** options for your search. This is particularly useful for macOS, where the user might have a lot of complex data to handle:



For this, we'll create another enum with 2 cases for the currently selected folder and for all notes independent of the folder:

```
enum NoteSearchScope: CaseIterable, Identifiable {

    case all
    case selectedFolder

    var id: Self { self }

    func name(folder: Folder) -> String {
        switch self {
            case .all:
                return "All"
            case .selectedFolder:
                return "\' \(folder.name) \'"
        }
    }
}
```

You can then create another state property "searchScope" and connect it to the **searchScopes** view modifier:

```
struct NoteListView: View {
    …

    @State private var searchScope = NoteSearchScope.selectedFolder

    var body: some View {
        List(selection: $selectedNote) {
            …
        }
        .searchable(text: $viewModel.searchText,
            …
        })
        .searchSuggestions({
            …
        })
        .searchScopes($searchScope, scopes: {
            ForEach(NoteSearchScope.allCases) { scope in
                Text(scope.name(folder: selectedFolder))
            }
        })
    }
}
```

With this setup, users can choose to search within all folders or a specific one they have selected.

By now, you've learned how to add a searchable view modifier to your NoteListView and how to implement search suggestions, tokens, and scope to refine the search experience. Each of these elements plays a crucial role in creating an intuitive and effective search feature for your notes app.

In the next lesson, we'll dive into the creation of an NSPredicate using all the search inputs we've gathered. This will allow us to filter our notes based on the user's search criteria.

6.8 COMBINING ALL SEARCH PARAMETERS TO ONE PREDICATE

So far, I only implemented the UI for searching. In this section, I want to update the note list to show the correct search results. We have to consider the search text, search tokens and scope. These are currently added to the UI in NoteListView with 3 state properties:

```
struct NoteListView: View {
    …

    @State private var selectedNoteSorting = NoteSorting.creationDateAsc

    let selectedFolder: Folder

    @State private var searchText: String = ""
    @State private var searchTokens = [NoteSearchToken]()
    @State private var searchScope = NoteSearchScope.selectedFolder

    var body: some View {
        List(selection: $selectedNote) {
              switch selectedNoteSorting {
                case .titleAsc, .titleDes, .creationDateAsc, .creationDateDes:
                    NoteSingleSortedView(predicate: viewModel.predicate,
                                         noteSorting: selectedNoteSorting)
                case .status:
                    NotesSectionedByStatusView(predicate: viewModel.predicate)
                case .day:
                    NotesSectionedByDayView(predicate: viewModel.predicate)
            }
            .searchable(text: $viewModel.searchText,
                …
            })
            .searchSuggestions({
                …
            })
            .searchScopes($searchScope, scopes: {
               …
            })
        }
    }
```

I need to create NSPredicates from these search criteria and use it with @FetchRequest and @SectionedFetchRequest in the subviews NoteSingleSortedView, NotesSectionedByStatusView and NotesSectionedByDayView.

ViewModel for Search Logic

To create an NSPredicate from all these search inputs, I will need to add quite a bit of code and I want to make sure I can write unit tests for them. Thus I will create a separate class that handles this logic.

Create a new swift file “NoteSearchViewModel.swift”.

Move the properties for search from NoteListView to the new class:

```
import Foundation

class NoteSearchViewModel: ObservableObject {

    @Published private var selectedFolder: Folder? = nil

    @Published var searchText: String = ""
    @Published var searchTokens = [NoteSearchToken]()
    @Published var searchScope = NoteSearchScope.selectedFolder

    // create combined predicate
}
```

And update NoteListView to use the new view model for searching:

```
struct NoteListView: View {

      let selectedFolder: Folder
      @Binding var selectedNote: Note?

      @State private var selectedNoteSorting = NoteSorting.creationDateAsc

      @StateObject var viewModel = NoteSearchViewModel()

     var body: some View {
          …
            .searchable(text: $viewModel.searchText,
                        tokens: $viewModel.searchTokens,
                        token: { token in
                Text(token.name)
            })
            .searchSuggestions({
                ForEach(NoteSearchToken.allCases) { token in
                    Button {
                        viewModel.searchTokens.append(token)
                    } label: {
                        Text(token.fullName)
                    }
                }
            })
            .searchScopes($viewModel.searchScope, scopes: {
                ForEach(NoteSearchScope.allCases) { scope in
                    Text(scope.name(folder: selectedFolder))
                }
            })
      }
}
```

Next, I need to make sure that the view model is also correctly updating when the selected folder changes. At the end of NoteListVie, add onAppear and onChange to pass the values for the selected folder to the view model:

```
struct NoteListView: View {
     …

    let selectedFolder: Folder

    var body: some View {
        …
        .onAppear {
            viewModel.selectedFolder = selectedFolder
        }
        .onChange(of: selectedFolder) { newValue in
            viewModel.selectedFolder = newValue
        }
    }
}
```

Now that we have all the data in NoteSearchViewModel, we can start creating the NSPredicate that describes all the filtering options for Core Data.

First, I will create functions that create predicates for each individual search filter. Afterwards, I will combine all predicates to one combined predicate.

Search Predicate for Search Text

The **searchTerm** string property should be used to search in the notes title or body property. Here is the function that generates the predicate:

```
class NoteSearchViewModel: ObservableObject {
  …

   func createSearchTextPredicate(text: String) -> NSPredicate? {
        guard !text.isEmpty else { return nil }

        let predicates = [NSPredicate(format: "%K CONTAINS[cd] %@",
                                      NoteProperties.title,
                                      text as CVarArg),
                          NSPredicate(format: "%K CONTAINS[cd] %@",
                                      NoteProperties.bodyText ,
                                      text as CVarArg)]

        return NSCompoundPredicate(orPredicateWithSubpredicates: predicates)
    }
}
```

I check if the search term is not empty otherwise I return nil because I don’t have any thing to search for. Then I create 2 separate predicates to search in the title or body text of the note. Last I combine the 2 predicates with an “OR”. This means the search text should be either in the title or body text.

Predicate for Search scope

Next, I want to filter for the search scope, which had 2 cases: search in all folders or the selected folder. In case of “all folders”, I don’t need to filter. So I return nil. Only for the “selected folder” case will the following function return a NSPedicate that searches for notes in the selected folder:

```
    func createScopePredicate(scope: NoteSearchScope, folder: Folder?) -> NSPredicate? {
        if scope == .selectedFolder,
            let folder = folder {
            return NSPredicate(format: "%K == %@",
                               NoteProperties.folder,
                               folder)
        } else {
            return nil
        }
    }
```

Predicate for Search Tokens

I set the search token up to search for a multitude of different criteria including notes by time interval, status, and if they have an attachment. I will create functions for these tokens individually.

First, let’s describe a predicate that filters for notes in the last week and last day:

```
    func createLast7DaysPredicate() -> NSPredicate {
        let calendar = Calendar.current
        let beginDate = calendar.date(byAdding: .day, value: -7, to: Date())!
        return NSPredicate(format: "%K > %@",
                           NoteProperties.creationDate,
                           beginDate as NSDate)
    }
```

```
    func createLast24HoursPredicate() -> NSPredicate {
        let calendar = Calendar.current
        let beginDate = calendar.date(byAdding: .hour, value: -24, to: Date())!
        return NSPredicate(format: "%K > %@",
                           NoteProperties.creationDate,
                           beginDate as NSDate)
    }
```

I also create a function that searches for notes which have an image attachment, In this case the attachment relationship should not be nil:

```
    func createAttachmentPredicate() -> NSPredicate {
        NSPredicate(format: "%K != nil", NoteProperties.attachment)
    }
```

Additionally, I am creating a function that filter for a given status. I want to return notes that have a certain status like “review”, “draft” or “archievd”:

```
func createStatusPredicate(status: Status) -> NSPredicate {
    return NSPredicate(format: "%K == %@",
                       NoteProperties.status,
                       status.rawValue as CVarArg)
}
```

The user can select more than one token. I want to handle the case of the token array. In the following, I write a function that takes an array of NoteSearchToken and returns an optional predicate. If the token array is empty, I return nil, which means I am not filtering.

```
func createTokenPredicates(tokens: [NoteSearchToken]) -> NSPredicate? {

    // start collecting all individual predicates
    var predicates = [NSPredicate]()

    // generate the individual NSPredicates for each token and add
    // it to the predicates array
    for token in tokens {
        switch token {
            case .draftStatus:
                predicates.append(createStatusPredicate(status: .draft))
            case .reviewStatus:
                predicates.append(createStatusPredicate(status: .review))
            case .archivedStatus:
                predicates.append(createStatusPredicate(status: .archived))
            case .last24Hours:
                predicates.append(createLast24HoursPredicate())
            case .last7Days:
                predicates.append(createLast7DaysPredicate())
            case .withAttachment:
                predicates.append(createAttachmentPredicate())
        }
    }

    if predicates.count == 0 {
        // no predicates, nothing to filter
        return nil
    } else if predicates.count == 1 {
        // only have one predicate, can return this one predicate here
        return predicates.first
    } else {
        // compounding all individual predicates to one search
        return NSCompoundPredicate(andPredicateWithSubpredicates: predicates)
    }
}
```

After I created all individual predicates for each token, I combine them with a compound predicate with “AND”. I only want to retrieve notes that fulfil all search token criteria.

Combing All Predicates

We have no the predicates for search term, search token and search scope for the selected folder. It is time to combine them into on single predicate.

Add another function that goes through all properties “searchText”, “searchScope”, “searchFolder” and “searchToken” and uses the predicate functions from above.

```
    func createFullPredicate() -> NSPredicate {

        var predicates = [NSPredicate]()

        // getting the individual predicates
        if let searchTextPredicate = createSearchTextPredicate(text: searchText) {
            predicates.append(searchTextPredicate)
        }

        if let folderScopePredicate = createScopePredicate(scope: searchScope,
                                                           folder: selectedFolder) {
            predicates.append(folderScopePredicate)
        }

        if let tokenPredicate = createTokenPredicates(tokens: searchTokens) {
            predicates.append(tokenPredicate)
        }

        // returning one search filter for the generated predicates:
        if predicates.count == 0 {
            return createScopePredicate(scope: .selectedFolder,
                                        folder: selectedFolder) ?? .none
        } else if predicates.count == 1 {
            return predicates.first ?? .none
        } else {
            return NSCompoundPredicate(andPredicateWithSubpredicates: predicates)
        }
    }
```

After I created all individual predicates, I combine them with a compound predicate with “AND”. This filter describes all user input from the search.

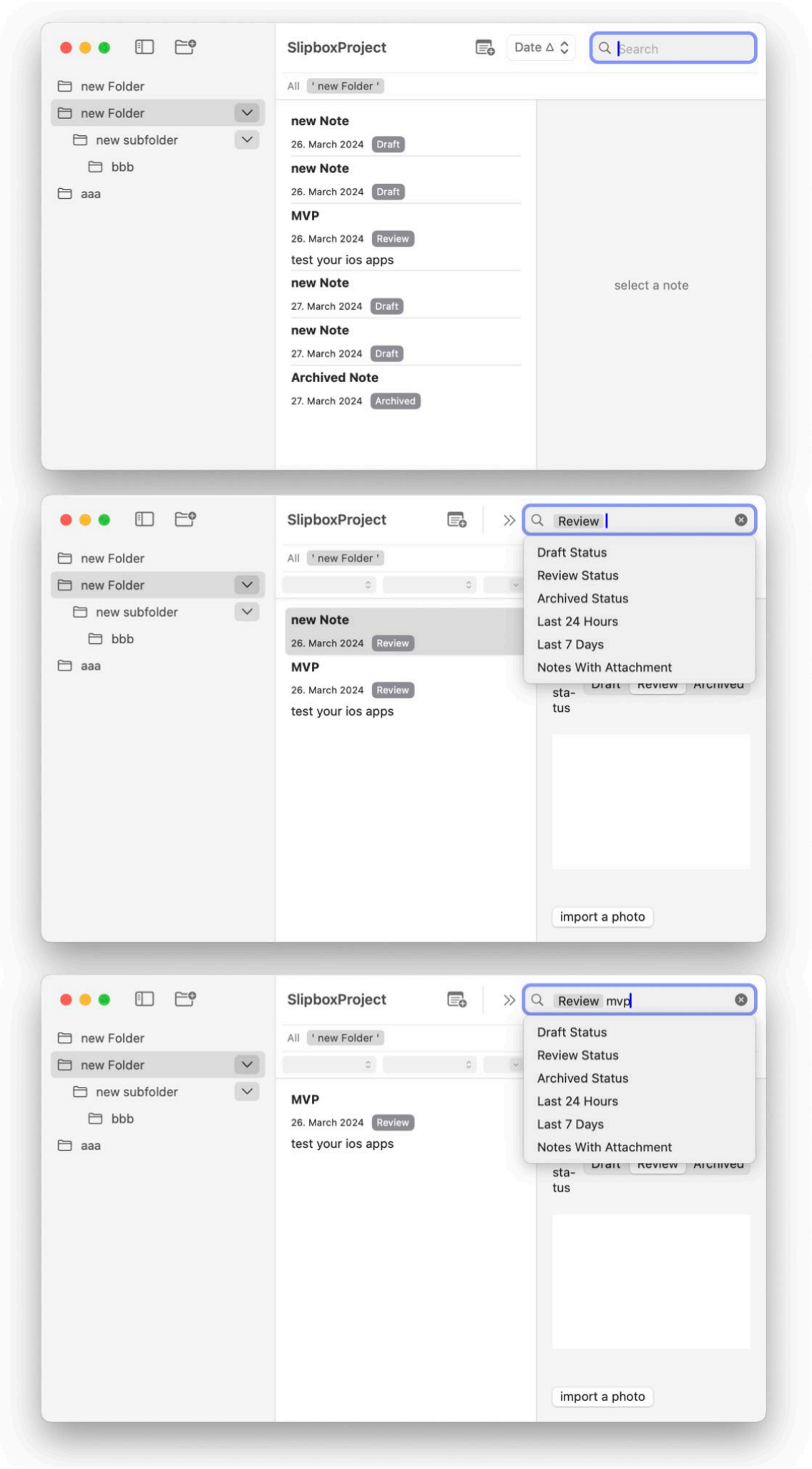
In the next section, we will update the UI of NoteListView to pass this predicate to show the filtered notes.

But before that, I need to call the function “createFullPredicate” which updates the predicate. I want to execute this function, every time one of the filter properties changes. In the following, I use a computed property:

```
class NoteSearchViewModel: ObservableObject {

    @Published var selectedFolder: Folder? = nil

    @Published var searchText: String = ""
    @Published var searchTokens = [NoteSearchToken]()
    @Published var searchScope = NoteSearchScope.selectedFolder

    var predicate: NSPredicate {
       createFullPredicate()
    }

    …
}
```

6.9 UPDATING THE NOTE LIST VIEWS WITH THE SEARCH PREDICATE

In this section, I'll walk you through how to update the UI with a new search predicate, which reflects the search text, tokens, and scope you've entered in the NoteSearchViewModel. The goal is to ensure that the list view shows only the notes that match your search criteria. By the end of this section, the note list should reflect correctly when you change the search on macOS. This will also work together with sorting.



I want to use the compound predicate from NoteSearchViewModel.

```
class NoteSearchViewModel: ObservableObject {
    …

    @Published var predicate: NSPredicate = .none

    …
}
```

To integrate the search predicate, we need to look at the list view that's currently displayed.

```
struct NoteListView: View {

      let selectedFolder: Folder
      @Binding var selectedNote: Note?

      @State private var selectedNoteSorting = NoteSorting.creationDateAsc

      @StateObject var viewModel = NoteSearchViewModel()

     var body: some View {
        List(selection: $selectedNote) {
             switch selectedNoteSorting {
               case .titleAsc, .titleDes, .creationDateAsc, .creationDateDes:
                   NoteSingleSortedView(predicate: viewModel.predicate,
                                        noteSorting: selectedNoteSorting)
               case .status:
                   NotesSectionedByStatusView(predicate: viewModel.predicate)
               case .day:
                   NotesSectionedByDayView(predicate: viewModel.predicate)
            }
        }
         // search …
      }
}
```

First, consider a view like NoteSingleSortedView, where you typically start by creating a fetch request with a selected folder:

```
struct NoteSingleSortedView: View {

  init(selectedFolder: Folder, noteSorting: NoteSorting) {
    let request = Note.fetch(for: selectedFolder)

    // sorting …
    self._notes = FetchRequest (fetchRequest: request, animation: .bouncy)
  }

   @FetchRequest(fetchRequest: Note.fetch(.none))
   private var notes: FetchedResults<Note>

   …
}
```

Instead of creating a new fetch request with a specific predicate, you can now pass down the comprehensive predicate from the NoteSearchViewModel. This predicate includes all the necessary conditions, including the selected folder.

Updating the Fetch Request

Here's how you modify the fetch request to use the new predicate:

```
    struct NoteSingleSortedView: View {

        init(predicate: NSPredicate, noteSorting: NoteSorting) {
            let request = Note.fetch(predicate)
            // sorting …

            self._notes = FetchRequest(fetchRequest: request)
        }

        @FetchRequest(fetchRequest: Note.fetch(.none))
        private var notes: FetchedResults<Note>

        …
    }
```

```
    struct NotesSectionedByStatusView: View {

        init(predicate: NSPredicate) {
            let request = Note.fetch(predicate)
              // sorting …

            self._sectionedNotes = SectionedFetchRequest(fetchRequest: request,
                                                         sectionIdentifier: \.sectionStatus)
        }

        …
    }
```

```
    struct NotesSectionedByDayView: View {

        init(predicate: NSPredicate) {
            let request = Note.fetch(predicate)
              // sorting …

            self._notesSections = SectionedFetchRequest(fetchRequest: request,
                                                        sectionIdentifier: \.day)
        }
        …
    }
```

```
    struct NoteListView: View {

        let selectedFolder: Folder
```

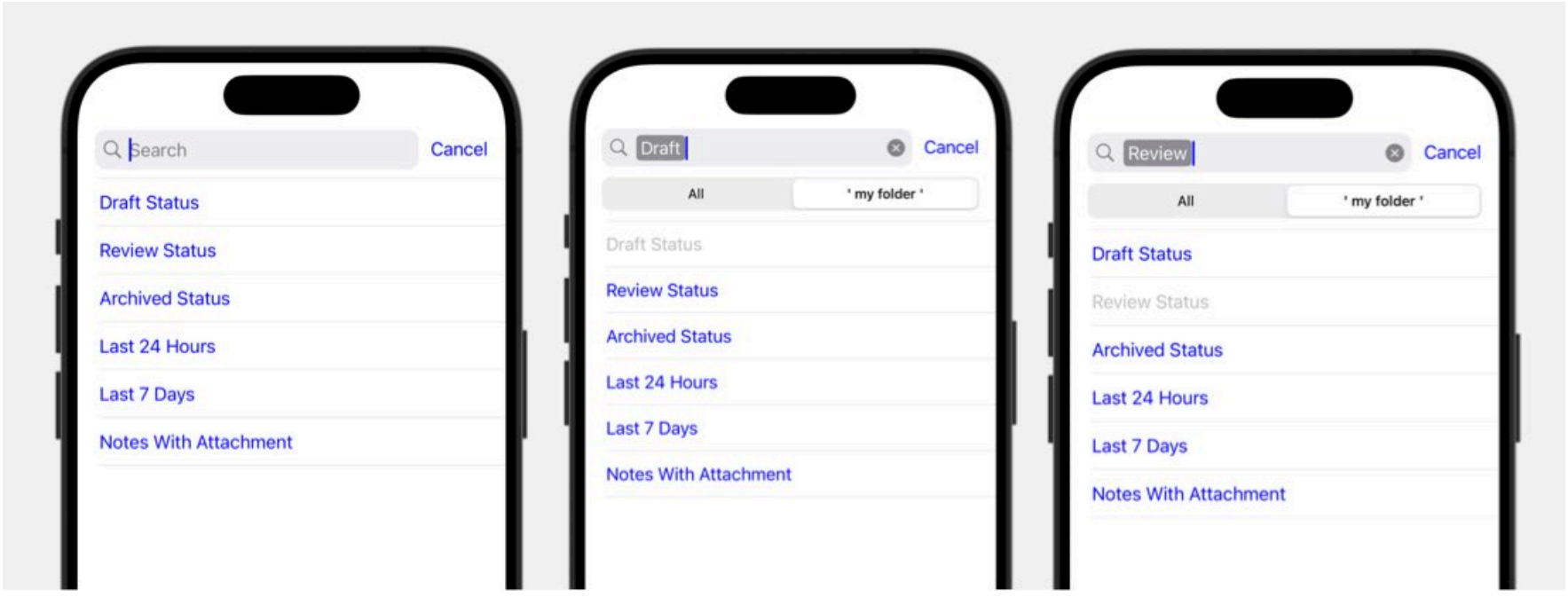
```
        @Binding var selectedNote: Note?

        @Environment(\.managedObjectContext) private var viewContext

        @State private var selectedNoteSorting = NoteSorting.creationDateAsc
        @StateObject var viewModel = NoteSearchViewModel()

        var body: some View {
            List(selection: $selectedNote) {

                switch selectedNoteSorting {
                    case .titleAsc, .titleDes, .creationDateAsc, .creationDateDes:
                        NoteSingleSortedView(predicate: viewModel.predicate,
                                             noteSorting: selectedNoteSorting)
                    case .status:
                        NotesSectionedByStatusView(predicate: viewModel.predicate)
                    case .day:
                        NotesSectionedByDayView(predicate: viewModel.predicate)
                }
            }
            // searching …
        }
    }
```

Testing the Search Functionality

You can test the search functionality on macOS, where the note list should update accordingly. On iOS, you will not see much, since I am releasing the search suggestions correctly. We will fix this soon.

Handling Status Tokens

When dealing with status tokens such as draft, review, or archived, it's important to ensure that they don't conflict. For example, a note can't be both in "Draft" and "Review" at the same time. Instead, when I already selected a status token e.g. "Draft", and then I tap on another status token e.g. "Review". I am removing the first "Draft" token and only showing the last selected token:



Here's how you could handle the addition of status tokens. I add an "add token" function to NoteSearchViewModel that handles the tokens. If I add a token that is status type, I check the existing tokens and if necessary remove tokens:

```
class NoteSearchViewModel: ObservableObject {
     ...

    func add(token: NoteSearchToken) {
        guard searchTokens.contains(where: { $0 == token}) == false else { return }

        if token.isStatusToken() {
            for existingToken in searchTokens {
                if existingToken.isStatusToken(),
                   let index = searchTokens.firstIndex(of: existingToken) {
                    searchTokens.remove(at: index)
                }
            }
        }

        searchTokens.append(token)
    }

    func isTokenSelected(token: NoteSearchToken) -> Bool {
        searchTokens.firstIndex(of: token) != nil
    }
}
```

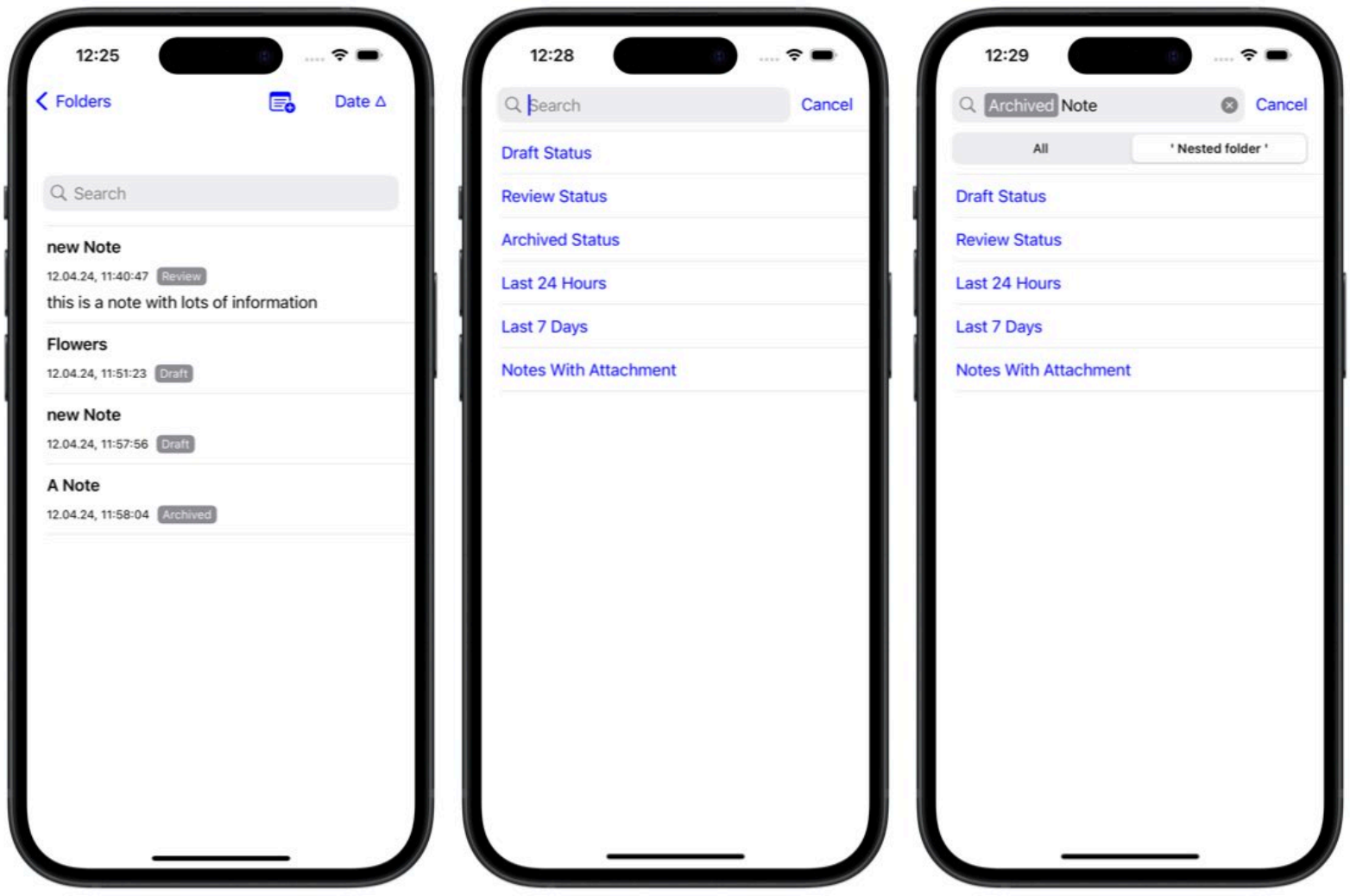
In NoteListView, when the user selects a token, I call the add token function of the view model:

```
struct NoteListView: View {
    …

    @StateObject var viewModel = NoteSearchViewModel()

    var body: some View {
        List(selection: $selectedNote) {
          …
        }
        .searchable(…)
        .searchSuggestions({
            ForEach(NoteSearchToken.allCases) { token in
                Button {
                    viewModel.add(token: token)
                } label: {
                    Text(token.fullName)
                }
                .disabled(viewModel.isTokenSelected(token: token))
            }
        })
     }
}
```

In addition, I disable the selected tokens in the suggestions list with another function "isTokenSelectded" of the view model.

6.10 SHOW SEARCH RESULTS ON IOS

In this section, we will focus on enhancing the search experience on iOS. In this lesson, we will address the issue of search suggestions overlaying the search results on iOS. Currently, when tapping on the search text field, the search suggestions appear and cover the search results below. This makes it impossible to see the results:



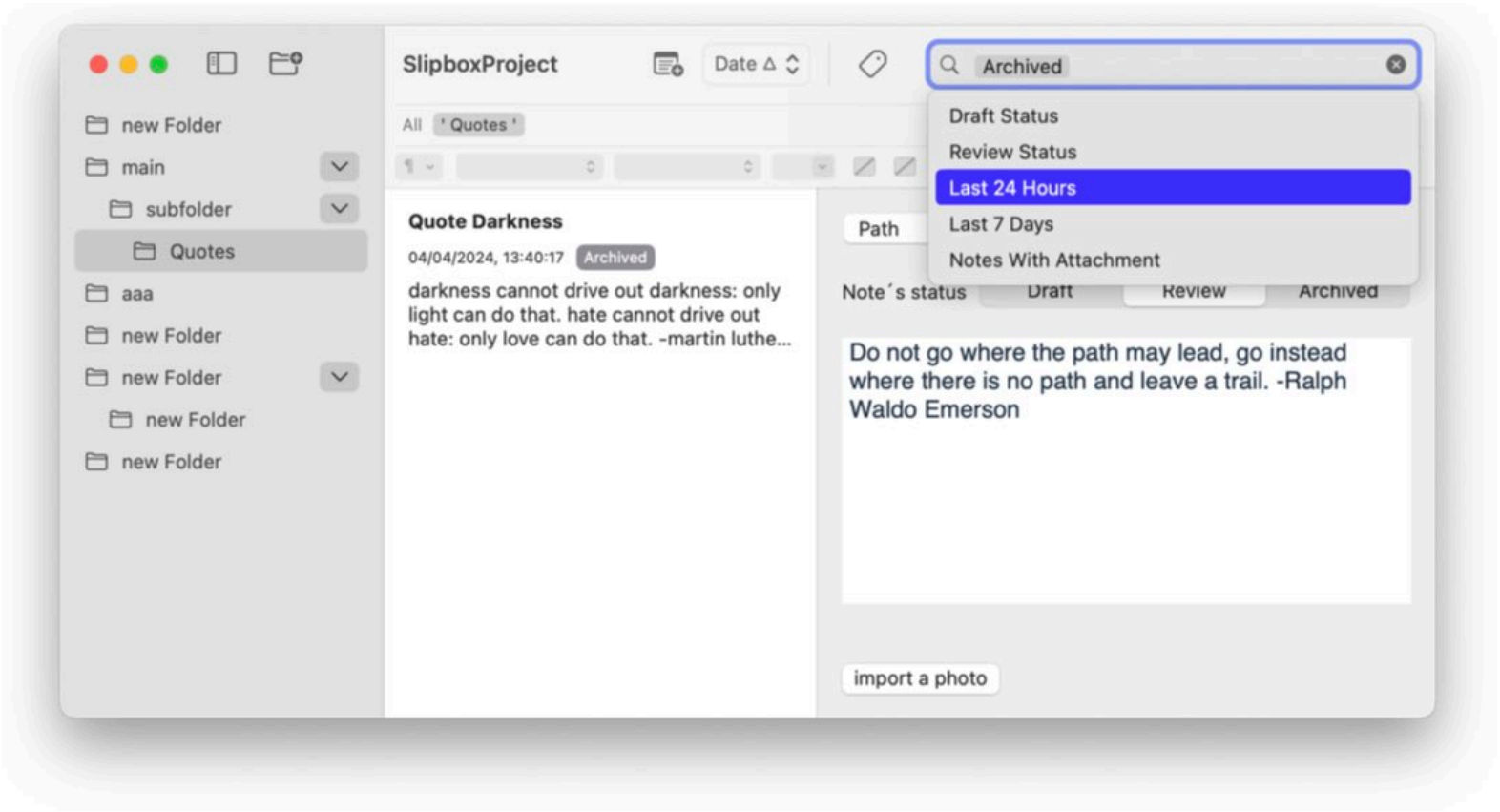
Currently, the search is implemented in the NoteListView:

```
struct NoteListView: View {
    …

    var body: some View {
        List(selection: $selectedNote) {
            …
        }
        .searchable(…)
        .searchSuggestions({
            …
        })
        .searchScopes($viewModel.searchScope, scopes: {
            …
        })
    }
}
```

To resolve this issue, we need to make some modifications. First, we will add a platform check to only use the search suggestions modifier on macOS.

```
#if os(macOS)
.searchSuggestions({
    …
})
.searchScopes($viewModel.searchScope, scopes: {
   …
})
#endif
```

On macOs, search suggestions are shown as a drop down menu below the search text field, which is suitable for that platform:



On iOS, we can now see the search results when tapping on the text field. Next we will display the search results manually inside the list.

```
struct NoteListView: View {
    …

    var body: some View {
        List(selection: $selectedNote) {
          // show search suggestions on iOS
           …
        }
        .searchable(…)
    }
}
```

To achieve this, we will create a subview called “SearchSuggestionsView”. We will use the “isSearching” environment property to determine if the user is currently searching.

```
fileprivate struct SearchSuggestionsView: View {

    @ObservedObject var viewModel: NoteSearchViewModel
    @Environment(\.isSearching) var isSearching

    var body: some View {
        if isSearching {
            Section("Search Suggestions") {
                SearchSuggestionsList(viewModel: viewModel)
            }
        }
    }
}
```

To avoid code repetition for the custom search suggestion view and the search suggestion modifier, create a separate view that show all search suggestions:

```
fileprivate struct SearchSuggestionsList: View {
    @ObservedObject var viewModel: NoteSearchViewModel

    var body: some View {
        ForEach(NoteSearchToken.allCases) { token in
            if !viewModel.isTokenSelected(token: token) {
                Button {
                    viewModel.add(token: token)
                } label: {
                    Text(token.fullName)
                }
                #if os(iOS)
                .foregroundColor(.cyan)
                #endif
            }
        }
    }
}
```

Modify NoteListView and show the SearchSuggestionsView above the search results but only for iOS:

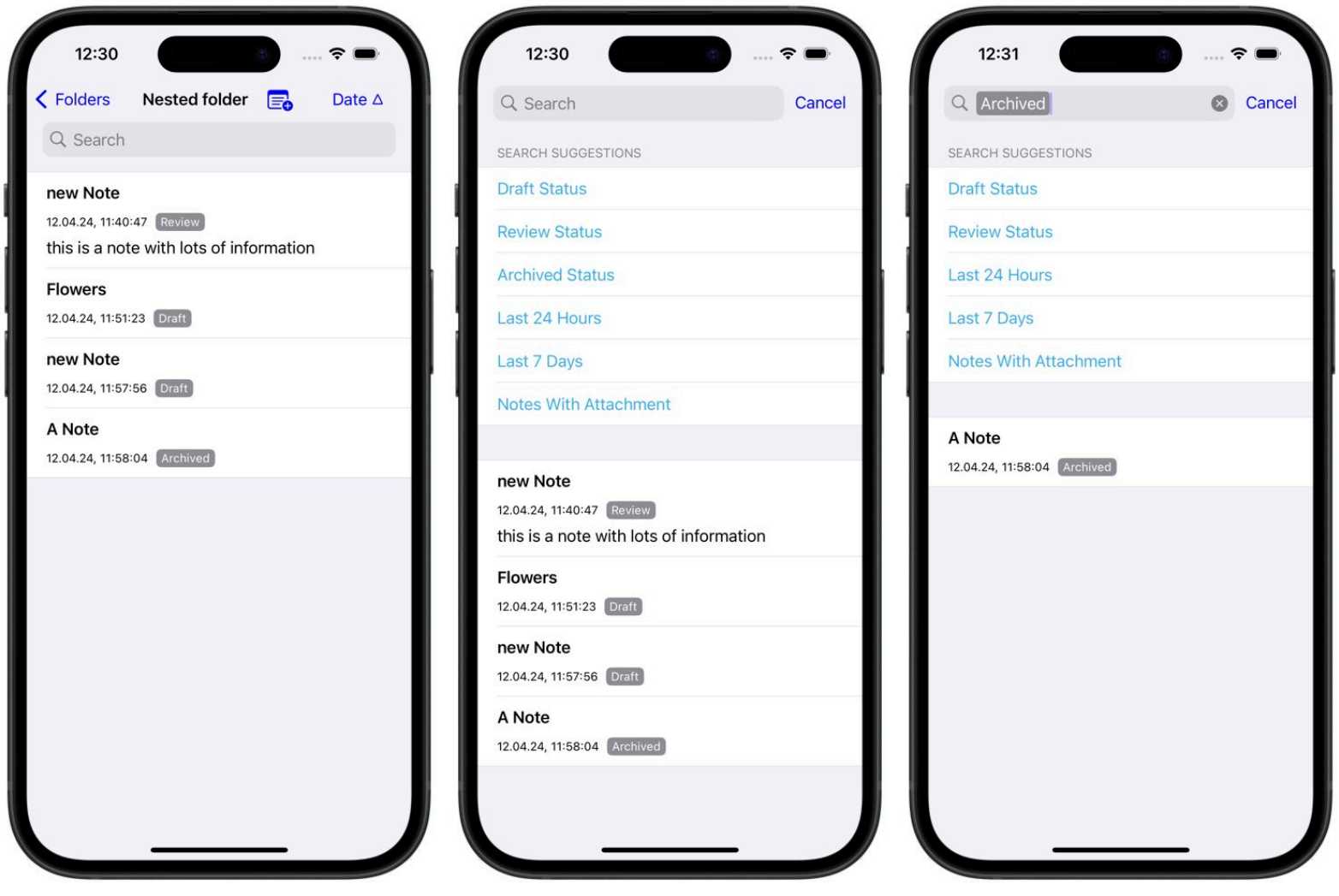
```
List(selection: $selectedNote) {
    #if os(iOS)
    SearchSuggestionsView(viewModel: viewModel)
    #endif

    switch selectedNoteSorting {
       …
    }
}
.searchable(…)
#if os(macOS)
.searchSuggestions({
    SearchSuggestionsList(viewModel: viewModel)
})
.searchScopes(… )
#endif
```

Now, when tapping on the search text field, the search suggestions will appear above the search results. You can customize the appearance of the search suggestions view by adding styling, such as changing the foreground color or font.

List Styling

Additionally, I will modify the list style on iOS to use the grouped style instead of the plain style. This can be done by using the if os(iOS) condition and setting the list style to grouped:

```
List(selection: $selectedNote) {
  …
}
.searchable(…)
#if os(macOS)
.searchSuggestions({
    SearchSuggestionsList(viewModel: viewModel)
})
.searchScopes($viewModel.searchScope, scopes: {
  …
})
#else
.listStyle(.grouped)
.navigationTitle(selectedFolder.name)
.navigationBarTitleDisplayMode(.inline)
#endif
```

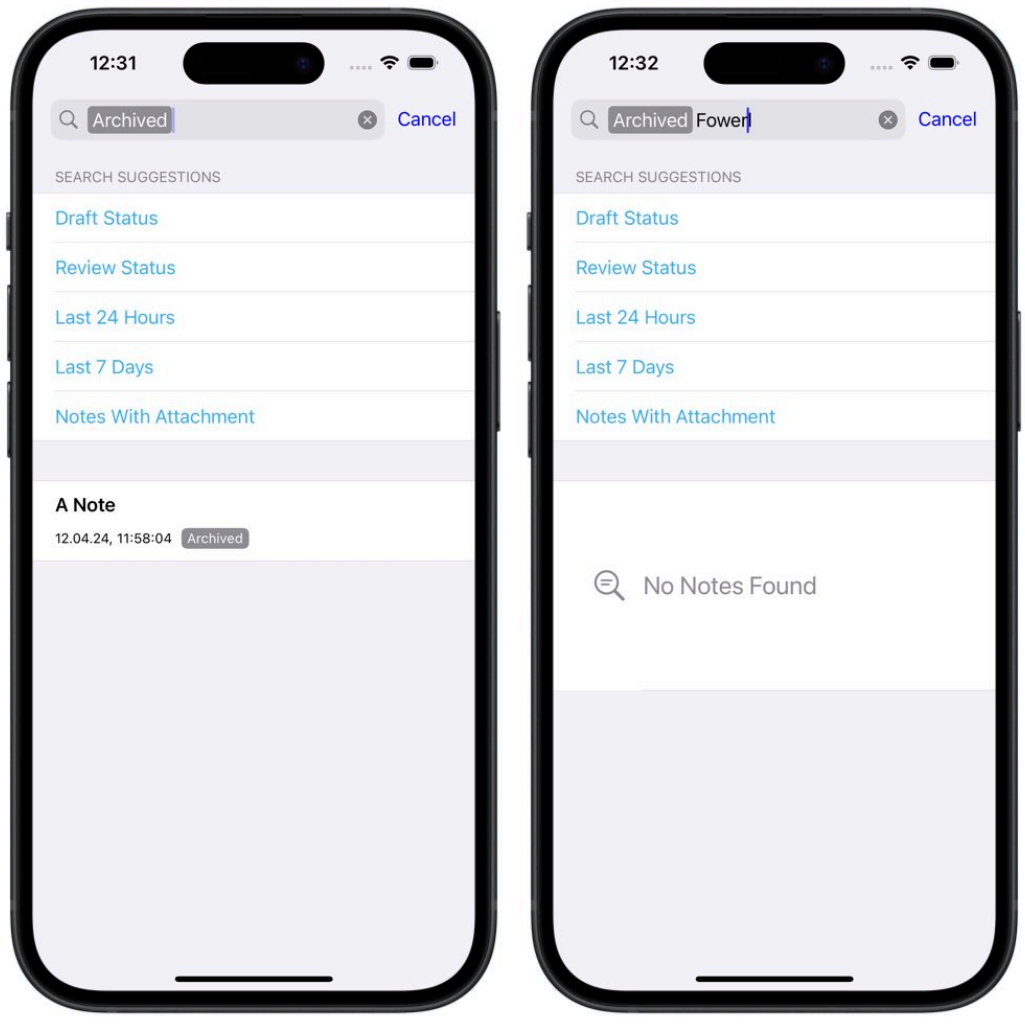


Showing a Custom Message When No Notes were Found

Additionally, we can improve the user experience by showing a message when no search results are found. To achieve this, we will create another subview called “NoResultsView”. To improve performance, I am creating a fetch request with “countResultsType” and perform the fetch in the initialiser:

```
fileprivate struct NoResultsView: View {

    init(predicate: NSPredicate, context: NSManagedObjectContext) {
        let request = Note.fetch(predicate)
        request.resultType = .countResultType
        self.count = (try? context.count(for: request)) ?? 0
    }

    let count: Int
    @Environment(\.isSearching) var isSearching

    var body: some View {
        if count == 0,
           isSearching {
            Label("No Notes Found”, systemImage:"text.magnifyingglass")
                .foregroundColor(.gray)
                .font(.title2)
                .padding(.vertical, 50)
        }
    }
}
```

I am only showing the feedback message, if the count of the search results is zero and if the user is currently searching.



I can then use “NoResultsView” in the “NoteListView” below the search suggestions:

```
    List(selection: $selectedNote) {

        #if os(iOS)
        SearchSuggestionsView(viewModel: viewModel)
        #endif

        switch selectedNoteSorting {
           …
        }

        NoResultsView(predicate: viewModel.predicate,
                      context: viewContext)
    }
```

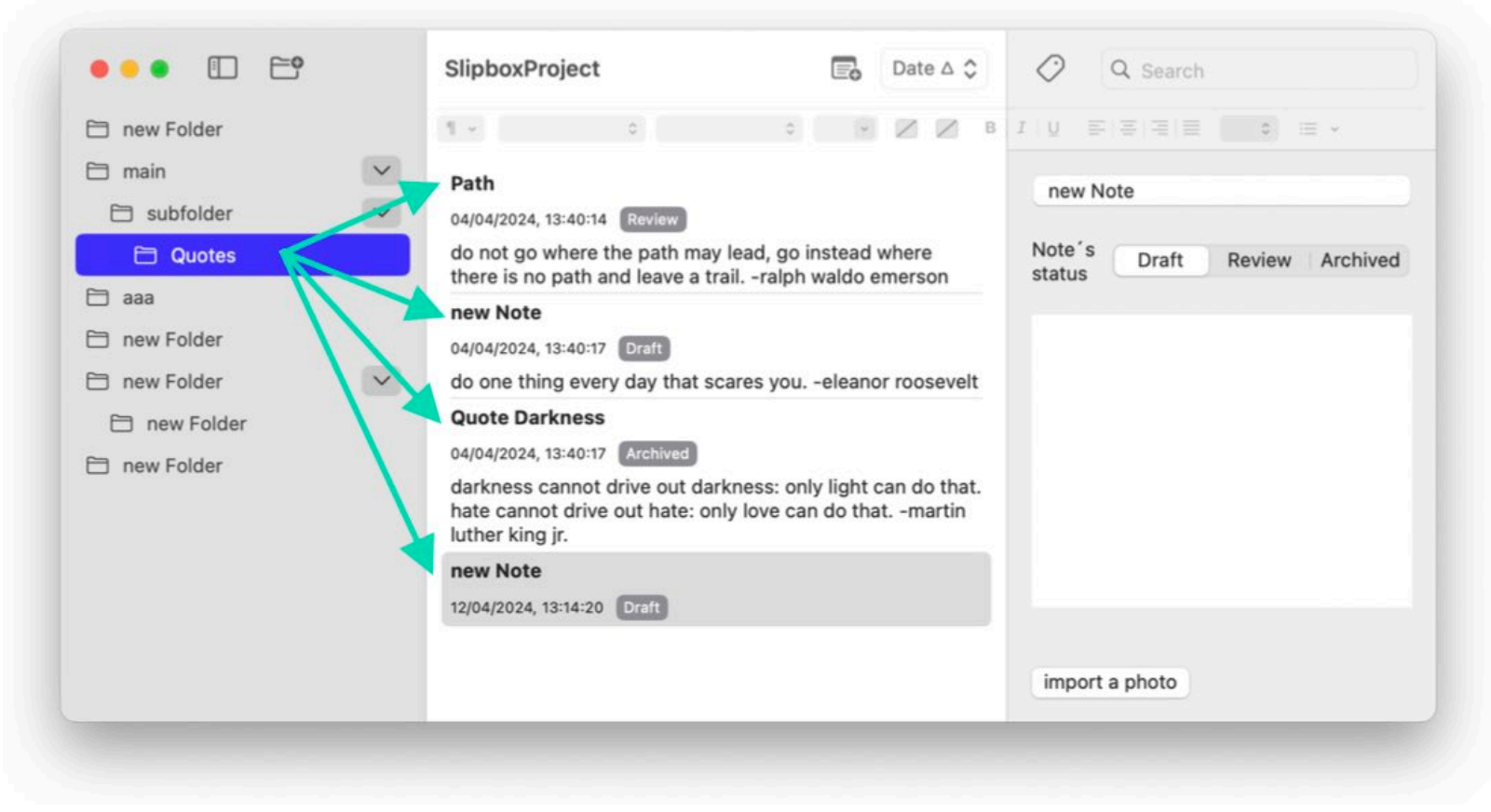
Additionally, I can show a feedback message when the user is opening an empty folder. For example, I can show a “This Folder is empty”. The following modification in “NoResultsView”, will check if the user is searching or not to show the different feedback messages:

```
    fileprivate struct NoResultsView: View {

         …

        let count: Int
        @Environment(\.isSearching) var isSearching

        var body: some View {
            if count == 0 {
                Label(isSearching ? "No Notes Found" : "This Folder is empty.",
                      systemImage: isSearching ? "text.magnifyingglass" :
                                                 "plus.rectangle.on.folder")

            }
        }
    }
```

In summary, by implementing the changes mentioned above, we have improved the search experience on iOS. The search suggestions now appear above the search results, and a message is displayed when no results are found.

6.11 BUG FIX: DATA FLOW WITH ONCHANGE

In this section, I'm going to walk you through a bug that you might encounter when your app is running in Navigation Split View mode. Let's say you're working within a folder in your app, and you add a new note. You expect this new note to appear in the list instantly, but it doesn't. This can be quite frustrating, right?



This issue arises because the NoteListView isn't updating as it should. After digging into the problem, I discovered that the culprit is the onchange modifier that's attached to the end of the NoteListView. It turns out that this modifier doesn't play well with updates in certain scenarios.

Here's the problematic code snippet:

```
struct NoteListView: View {
     …

    let selectedFolder: Folder

    @StateObject var viewModel = NoteSearchViewModel()

    var body: some View {
        List(selection: $selectedNote) {
            …
        }
          …
        }
        .onAppear {
            viewModel.folderChanged(to: selectedFolder)
        }
        .onChange(of: selectedFolder) { newValue in
            viewModel.folderChanged(to: newValue)
        }
    }
}
```

In iOS 16 and macOS 14, Apple introduced a new variant of the onchange modifier that provides both old and new values, which can be quite handy. Here's how you can use it for the selectedFolder:

```
.onChange(of: selectedFolder, { newValue, oldValue in
     viewModel.folderChanged(to: newValue)
})
```

But what if you want to support earlier iOS versions and still fix the bug? The solution is to use the @ObservedObject property wrapper for the selectedFolder. SwiftUI will then track the changes correctly and update the view as expected.

Here's how you can apply this fix:

```
struct NoteListView: View {

     …

    @ObservedObject var selectedFolder: Folder

    @StateObject var viewModel = NoteSearchViewModel()

    var body: some View {
        List(selection: $selectedNote) {
             …
        }
        …
        }
            .onAppear {
                viewModel.folderChanged(to: selectedFolder)
            }
            .onChange(of: selectedFolder) { newValue in
                viewModel.folderChanged(to: newValue)
            }
        }
    }
```

Once you've made these changes, run your app again. Navigate to a folder, add a new note, and you should see it appear in the list immediately. This fix ensures that your NoteListView updates correctly, even when using core data in conjunction with SwiftUI's view models.

Remember, sometimes SwiftUI or Core Data can have updating issues, especially when connecting data to view models. When you encounter such issues, it's essential to identify what's causing the problem. In this case, it was the onchange modifier not behaving as expected.

It's important to note that as SwiftUI evolves, certain modifiers may become deprecated, and it's always a good idea to stay updated with the latest changes and use the new modifiers when possible.

By understanding and applying these fixes, you can ensure a smoother user experience, free from frustrating update bugs.