Gagan Diwan

Adopting SwiftData: A Simplified Approach to Persistence in Native iOS Application

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication

Bachelor’s Thesis

6 May 2024

Abstract

Author:	Gagan Diwan
Title:	Adopting SwiftData: A Simplified Approach to Persistence in Native iOS Application
Number of Pages:	53
Date:	6 May 2024

Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Development
Supervisors:	Janne Salonen (Director of School)

---

iOS development is ever evolving and has kept pace with recent shift towards declarative programming practices by introducing new frameworks such as SwiftUI and SwiftData that enables persistence using declarative code.

This thesis explores the seamless integration of SwiftData, a persistent framework, with SwiftUI applications. By leveraging Swift's native types for data modeling, SwiftData bridges the gap between code and data definition.

The adoption process involves five steps: building models, importing SwiftData & annotating model class, defining relationships and property attributes, setting model container and finally, accessing model context object to work with data.

The adoption process is explored through three demo applications. A book application demonstrates one-to-one relationships and core SwiftData adoption steps. A messaging application showcases one-to-many relationships, while a learning diary application discusses many-to-many relationships and the final adoption step. Additionally, the thesis explores CRUD operations (Create, Read, Update, Delete) alongside filtering, sorting, searching, and migration strategies.

While SwiftData offers tight SwiftUI integration and features like automatic lightweight migrations and custom migration support, the framework is still under development. The current limitations include compatibility only with iOS 17 onwards and a smaller API compared to Core Data. However, SwiftData's potential for the future is promising as it continues to evolve.

This thesis concludes that a detailed evaluation of SwiftData's functionality based on specific application needs is recommended before adopting it for new projects. Further research opportunities include a comparative study between SwiftData and Core Data and exploring SwiftData in networking environment for real-world application implementation. These investigations will solidify SwiftData's position as a powerful tool for building persistent enabled SwiftUI applications.

Keywords: SwiftData, SwiftUI, Swift, iOS, Persistence, Core Data

---

The originality of this thesis has been checked using Turnitin Originality Check service.

Contents

1 Introduction 7

1.1 The Advent of Declarative Frameworks 7

2 Object Graph 8

3 SwiftData 9

3.1.1 Data Models 10
3.1.2 Data Types Supported by SwiftData 12
3.1.3 Model Container 13
3.1.4 Model Context 14

4 Thesis Delimitations 15

5 SwiftData in Practice 16

5.1 One-to-One Relationships 16
5.1.1 Making App persistence ready 17
5.2 One-To-Many relationship 23
5.3 Many-To-Many Relationships 25
5.3.1 Accessing Model Context 28
5.3.2 CRUD (Create) 29
5.3.3 CRUD (Read) 29
5.3.4 CRUD (Update) 31
5.3.5 CRUD (Delete) 32
5.4 Migration 32
5.5 Filter & Search 33

6 Conclusion & Future Research 35

References 37

Appendices
Appendix 1: Source code for Book App
Appendix 2: Source code for Messaging App
Appendix 3: Partial Source code for Learning Diary App

List of Abbreviations

App: Mobile application or App file in Xcode project.

OGM: Object Graph mapping. How objects in a program connects and relates to other objects.

ORM: Object-relational mapping. The set of rules for mapping objects in a programming language to records in a relational database, and vice versa.

Model: Swift file that describes entities (objects of an application) in code.

Schema: Object Graph. Represents relationships and connections between model entities.

1 Introduction

Native iOS development has kept pace and evolved with the advancement of technology. Apple introduced its first iPhone on January 9, 2007 [1], and soon after there was tremendous interest in developing applications for iOS devices.

Earlier iOS application development was done in Objective-C, a C-based object-oriented programming language [2]. More precisely, mobile applications were developed in Objective-C using Cocoa Touch an application development environment that includes Objective-C runtime and frameworks [3].

As the complexity of applications increased Apple introduced the new multi-paradigm programming language Swift in 2014 [4] which incorporates many modern language features such as automatic memory management, type inference, generics, first-class functions, etc [5].

Swift was designed for safety and includes numerous features such as mandatory initialization of variables before usage, overflow checks on integers and arrays, and so on. National Security Agency (NSA) of America in its latest brief recommends developers adopt memory-safe language such as Swift [6].

1.1 The Advent of Declarative Frameworks

Complex application workflow and responsive user interfaces are symbol of modern mobile applications. Industry has reacted to the requirement to simplify application development and decrease time to market by introducing frameworks that adopt declarative paradigms in application development such as React and React Native.

Declarative programming focuses on the desired result “What”, without explicitly specifying “How”, to achieve the result. Although Swift programming made iOS

programming safer and more manageable, it still relied on the imperative paradigm of coding using the UIKIT framework.

SwiftUI framework introduced in 2019 is a framework that utilizes declarative UI paradigms to create dynamic user interfaces for iOS and Mac devices. The framework is interoperable with UIKit and other Apple APIs. Moving forward apple's roadmap is to transition application development toward SwiftUI and gradually phase out UIKit.

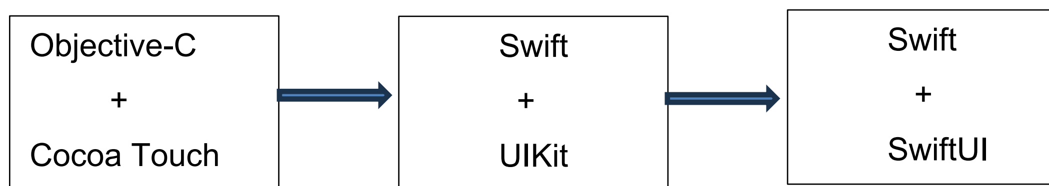


Figure 1. Evolution of iOS frameworks.

Core Data is a mature persistence framework that has served as a main persistence solution since the days of Objective-C. That brings it the advantage of stability and time-tested functionality [7]. However, maturity also means the importance of keeping backward compatibility, following old design patterns, and thus introducing rigidity in future development. Therefore, its integration with modern frameworks like SwiftUI is not seamless.

SwiftData is built on Core Data and takes advantage of recent developments in Swift language functionality such as Macros. It is built from the ground up to integrate with SwiftUI development practices.

2 Object Graph

Objects in an object-oriented programming are related to each other in simple or complex networks. For instance, a collection of string is a simple graph,

whereas collection of big objects, that reference other big objects, with their own views and navigations are called complex object graph. This grouping /network of objects is known as object graphs [8].

Object graphs tend to become more complex as they evolve with the addition of new objects as application grows due to usage or introduction of new functionality [9]. For instance, a learning diary application could have tag object and diary object. Each diary entry could be associated with multiple tags and vice versa. Tag could keep track of all entries it is attached to. This would create complex graph as the number of diary entries increases.

Apple has history of utilizing object graphs in their persistence framework. In fact, core data is defined as an advance graph management framework that is adept in storing and managing large volumes of data [10]. Since, SwiftData is built on Core Data it leverages schema (object graph) to manage persistence. OGM (object graph mapper) insulates developer from underlying mechanism involved in managing relationships and persistence layer. This abstraction enables developer to interact with persistence mechanism using Swift programming language. Therefore, complexity of development decreases as developers are exempted from writing custom persistence code for various storage medium such as SQLite or XML.

The concept of OGM (Object Graph Mapping) shares similarities with ORM (Object Relational Mapping). They both provide abstraction layer between objects and underlying persistence mechanism, but they differ in their implementation. Due to this similarity Core Data / SwiftData is also mistakenly called ORM. The following section will elaborate on SwiftData.

3 SwiftData

SwiftData is a persistence framework that facilitates on-device data storage using declarative code. SwiftData enables iOS developers to use regular Swift language constructs to create and manage data models. As such, it offers familiar

workflow and syntax to SwiftUI developers thus creating a seamless application development and integration experience [11].

SwiftData achieve this fluency by leveraging Swift language features such as macros; a central concept in SwiftData. Macros are keywords that at compile time modifies the code they are annotated to [12]. @Model being the most prominent macros in SwiftData simplifies and reduces the amount of code that needs to be written by a coder to make class properties persistent. Following sections will briefly discuss key components that constitutes SwiftData.

3.1.1 Data Models

Data models are the visual representation of entities in an application. It outlines the relationships between entities and the type of attributes that are stored in the database [13].

Data models in essence represents the business logic of an application and describe the data attributes and its relationships. It separates the views from the data that defines business logic [14].

Defining model is the first step in developing SwiftData application. Unlike in Core Data, in SwiftData applications, data models are normal classes and not created in special editor. Figure 2 depicts data model class in swift and Figure 3 provides snippet of SwiftData model class. Model class describes the type of data, entities hold. In this simple model, Item entity has single property named timestamp of type Date.

```
import Foundation
import SwiftData

final class Item {
    var timestamp: Date

    init(timestamp: Date) {
        self.timestamp = timestamp
    }
}
```

Figure 2. Example of Data Model

However, to persist data and turn data model into SwiftData model, SwiftData needs additional instructions. This functionality is achieved by annotating data models with the @Model macro [15][16]. As previously discussed, macros are a cornerstone within the SwiftData framework. Macros inject additional code in the data model class, describing for instance, what properties will persist(@_persisitedProperty) and what won't (@Transient). Figure 3 Depicts SwiftData models with code auto injected by @Model macro. Furthermore, SwiftData represents reference types as relationships and the type of schema (object graph of the application) that gets generated is dependent on the annotations applied to the properties. These annotations and their effect on schema will be explored further in the thesis.

```
import Foundation
import SwiftData

@Model
final class Item {
    @_PersistedProperty
    var timestamp: Date

    init(timestamp: Date) {
        self.timestamp = timestamp
    }
    @Transient
    private var _$backingData: any SwiftData.BackingData<Item> = Item.createBackingData()

    public var persistentBackingData: any SwiftData.BackingData<Item> {
        get {
            _$backingData
        }
        set {
            _$backingData = newValue
        }
    }

```

Figure 3. Example of SwiftData Model

3.1.2 Data Types Supported by SwiftData

In SwiftData computed properties are not stored as by definition computed properties calculates (computes) rather than store data [17][18]. Except for computed properties, SwiftData can persist properties of various types (primitive and complex) as long as they conform to Codable protocol. Codable protocols facilitates saving and transfer of data [19].

Table 1: Data types supported by SwiftData model.

Primitive Types	Complex Types
Integers, Decimal, Double, Float	Enums
Boolean	Structs, Classes
String	Arrays
Data (images/blobs)	
Date	
URL	
UUID	

3.1.3 Model Container

ModelContainer object acts as an intermediary between the in-memory transient workspace of model objects (model contexts) and the underlying persistent storage [20].

Model containers assume the responsibility of creation and management of physical data persistence layer. It defines how the data is stored on device (in-memory or on disk) and is responsible for maintaining the versioning, migration, and graph separation of corresponding data model and storage.

As discussed earlier in Data model section, Schema (object graph representation) is automatically created by SwiftData. Schema represents the data model and encapsulates all changes made to the model. ModelContainer is instantiated with Schema reference and act as conduit between the Schema and its persistence. ModelContainer facilitates automated schema migrations to ensure the persisted data remains compatible with evolving schema definitions.

Creating a model container

```
init(for: Schema, migrationPlan: (any SchemaMigrationPlan.Type)?,
configurations: [ModelConfiguration]) throws
```

Creates a model container using the specified schema, migration plan, and configurations.

Figure 3. API reference for initializing model container [20].

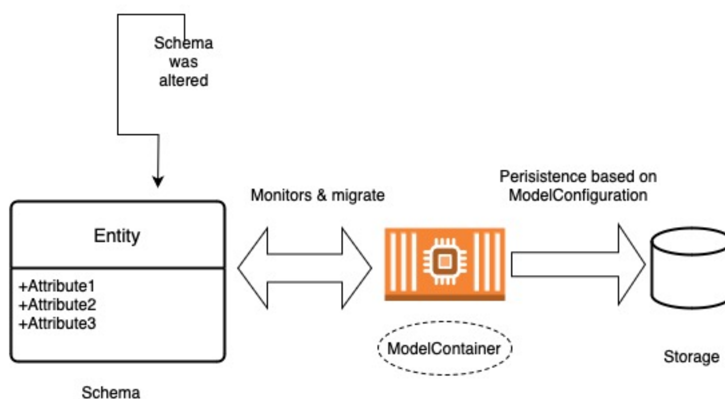


Figure 4. Conceptual overview of model container

ModelContainer persists data based on the Schema persistence behaviour described in ModelConfiguration object. ModelConfiguration controls where the data is stored, whether in-memory for transient data or on disk for persistent data. ModelConfiguration can be further configured to enable read-only mode for persisted data to prevent writes to sensitive data [21].

```
init(String?, schema: Schema?, isStoredInMemoryOnly: Bool, allowsSave:
Bool, groupContainer: ModelConfiguration.GroupContainer, cloudKit
Database: ModelConfiguration.CloudKitDatabase)
```

Creates a named model configuration for the specified schema.

Figure 5. API reference for Model configuration initializer [21].

3.1.4 Model Context

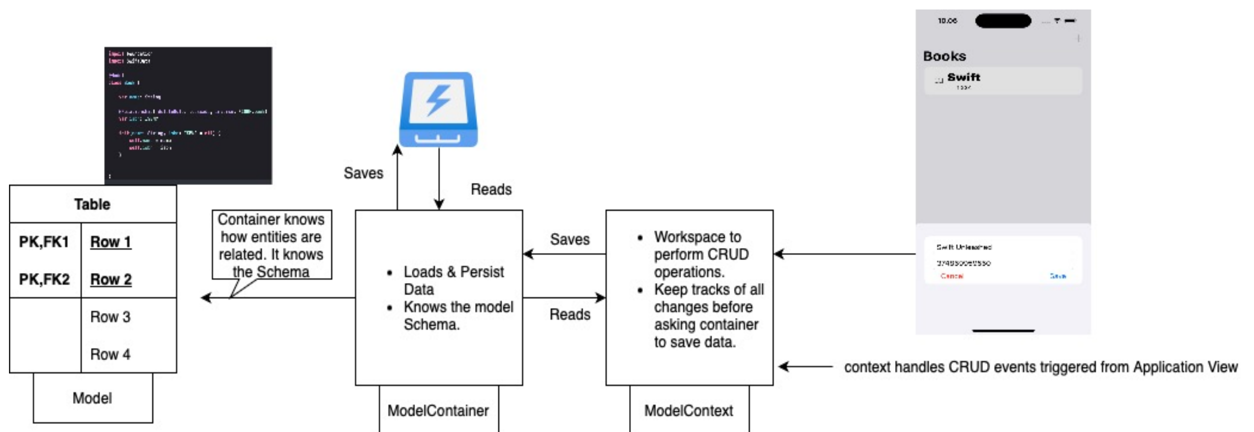
ModelContext is an in-memory workspace in SwiftData. It serves as the central location for all CRUD (Create, Read, Update, Delete) operations before they are saved by ModelContainer. ModelContext is also responsible for keeping track of changes to in-memory data and persists edits through ModelContainer [22].

Furthermore, model context is equipped to recognize the schema graph via root model and perform necessary operations on related models without explicit insertion of individual model in context. This is achieved due to integration between model container and model context. As it is model container that provides knowledge about schema and storage to model context. When Model container is attached to a view it establishes a binding between key in environment and the container's mainContext. This binding enables Swift data applications to perform queries using context on saved data. Figure 6 depicts modelContext being accessed via @Enviornment property wrapper and figure 7 provides conceptual overview of SwiftData framework.

```
struct LastModifiedView: View {
    @Environment(\.modelContext) private var modelContext

}
```

Figure 6. Accessing model context via @Environment property wrapper [22].

Figure 7. Conceptual view of SwiftData Framework.



4 Thesis Delimitations

The purpose of thesis is to explore SwiftData framework and how to adopt it in native iOS application built using SwiftUI framework. As such, this thesis is not a comparative analysis between Core Data and SwiftData.

SwiftData and SwiftUI are still evolving with inclusion of new API and deprecation of older. Therefore, this thesis is not an exhaustive compilation of every functionality provided by SwiftData. Instead, this thesis focuses on the core concepts of CRUD (Create, Read, Update, Delete) and how it can be implemented in SwiftData.

The thesis explores these topics by building three demo applications that each represents most adopted entity relationships in an application, namely: one-to-one relationship, one-to-many relationship and many-to-many relationship.

Furthermore, to avoid duplication of efforts, API functionality is demonstrated as it is deemed relevant and as such not all functionalities will be demonstrated in each application.

Finally, as this thesis is about SwiftData. SwiftUI functionality has been kept to minimal. The interface is designed to take input, remove, and display list items with basic navigation capabilities. Optimization of views and refactoring of SwiftUI code has not been implemented in this thesis.

Subsequent sections will demonstrate practical aspects of SwiftData.

5 SwiftData in Practice

Following section will explore SwiftData via demo applications. First demo application, a book application, explores One-To-One relationships and crucial four out of five steps required in adopting SwiftData in an application. Second application, a messaging application, explores One-To-Many relationships and finally a learning diary application that explores remaining fifth step and Many-to-Many relationships. Additionally, CRUD functionality along with Filter, Sort, Search, and migration strategies are discussed in following sections.

5.1 One-to-One Relationships

Two entities are said to be in a one-to-one relationship when single record in the first table is related to only one record in the second table and vice-versa [23]. In the demo application, Book entity relates to ISBN entity via the isbn property in the book. Each book record will be associated with only single isbn record and each isbn will only be associated with single book record.

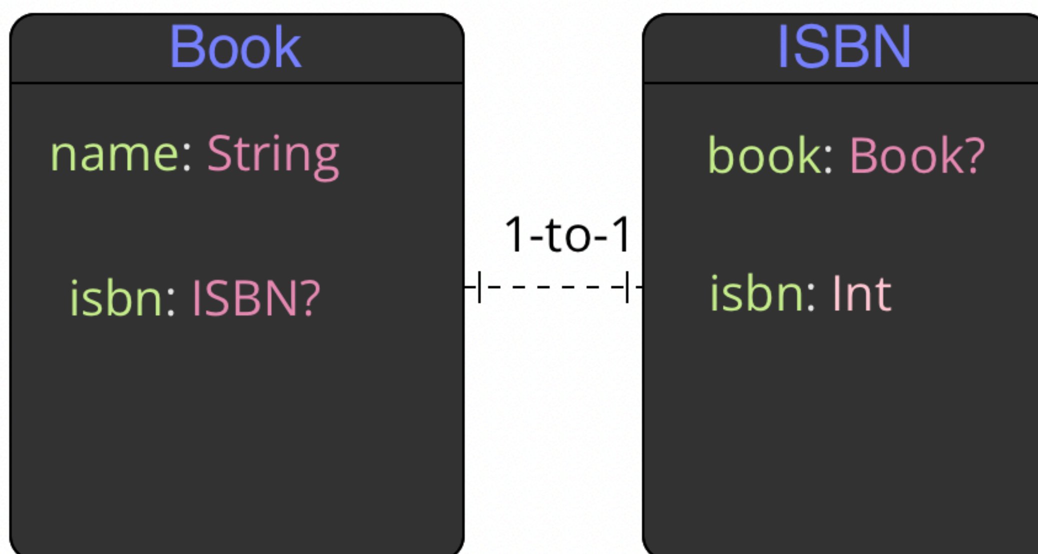


Figure 8. One-to-One Relationship Entity diagram

In the book application model, it is optional to include reference to isbn property and vice versa. Optionals are donated by "?" in swift and it denotes whether a property has value or not (nil). Therefore, a book item can be created without providing corresponding isbn record and isbn record can be created without associating it with the book record.

5.1.1 Making App persistence ready

First step is to build a data model for an application. Data models represents application logic in code. This demo app consists of two models namely: Book.swift and ISBN.swift. Models are kept simple to understand how SwiftData handles one-to-one relationships.

```swift
import Foundation
import SwiftData

@Model
class Book {

    var name: String

    @Relationship( deleteRule: .cascade, inverse: \ISBN.book)
    var isbn: ISBN?

    init(name: String, isbn: ISBN? = nil) {
        self.name = name
        self.isbn = isbn
    }


}
```

```swift
import Foundation
import SwiftData

@Model
class ISBN {
    @Attribute(.unique)
    var isbn: Int
    @Relationship(deleteRule: .cascade)
    var book: Book?

    init(isbn: Int, book: Book? = nil) {
        self.isbn = isbn
        self.book = book
    }


}
```

Figure 9: Setting up SwiftData models.

Secondly, SwiftData library must be imported, and model class must be annotated with @Model macro as depicted in figure 9. Although, structs are widely used in iOS applications. Models are constructed using classes since classes have inbuild identification features thereby making them suitable for reuse across various views. Annotation of class with @Model enables SwiftData to understand the schema (object graph) of the application and it monitors the class for any changes to adjust schema accordingly. Furthermore, this class is the interface against which the code is written to interact with data as ModelContainer consumes this class Schema to setup database for persistence.

Thirdly, attributes that are linked to other models are annotated with @Relationship macro.

Relationship(_:deleteRule:minimumModelCount:maximumModelCount:originalName:inverse:hashModifier:)

Figure 10. Relationship macro parameters [24].

@Relationship macro informs SwiftData about how to manage annotated property for relationship between models. It outlines the delete rules that are enforced when records are deleted [25]. For instance, in Book application presented above in Figure 9. cascade rule deletes any associated record in another model along with the book record, ISBN record in this case. Other rules are as follow:

Cascade	Deletes related record in associated model alongside
Deny	Record cannot be deleted if it contains reference to another model. For instance, ice-cream factory record cannot be deleted if there are associated records of ice-creams it produces in product model.
noAction	No changes are made to related models.
nullify	It nullifies the related model's reference and swift garbage collector (Automatic reference counting) handles the memory reclaim. This is default option if not explicitly selected.

Table 2. Delete rules of Relationship [25]

Inverse parameter is the keypath of the inverse property in the related model. It is sufficient to define inverse parameter in one of the models. In fact, it is not permitted to declare inverse relationship on both sides of the relationship. In Book model isbn property is related to the book property in ISBN model. This is SwiftData process of making bi-directional connection between models.

However, relationship can also be inferred implicitly if at least one of the referencing properties is optional. For instance, a Book model can be defined using non-nil isbn property and isbn model with optional book. In this instance, SwiftData will infer relationships implicitly as depicted in figure 11.

```
import Foundation
import SwiftData

@Model
class Book {

    var name: String

    var isbn: ISBN

    init(name: String, isbn: ISBN) {
        self.name = name
        self.isbn = isbn
    }


}
```

```
import Foundation
import SwiftData

@Model
class ISBN {
    @Attribute(.unique)
    var isbn: Int

    var book: Book?

    init(isbn: Int) {
        self.isbn = isbn
        //self.book = book
    }


}
```

Figure 11. Implicit Relationship in SwiftData

Additionally, properties can also be annotated using @Attribute macro to mark property as unique or to further customize property behaviour. For instance, @Attribute(.unique) in case of ISBN model, ensures that isbn record is unique and attempt to re-use value results in UPSERT operation instead of INSERT as depicted in figure 14. That is, record is updated with the new book object instead of another object being created as depicted in app workflow diagram. This arises from the fact that, If the insert collides with existing data, it automatically becomes an update due to unique attribute and SwiftData updates the properties of the existing data instead of creating new record [26].

Another commonly used @Attribute option is externalStorage used in case of storage of data blobs such as images [26].

Finally, Model container is created in App file to make it available for the entire lifecycle of an application as depicted in figure 12.

```
import SwiftUI
import SwiftData

@main
struct ProjectX_1_1App: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .modelContainer(for: [Book.self,ISBN.self])
    }
}
```

4

Figure 12. Model container modifier attached to Scene.

The model container provides the persistent storage for application model types. It can be initialized to use the default settings as in this case by specifying schemas (Book, ISBN), or it can be customized with configurations and migration options. By attaching modelContainer modifier within scene or view code, the application environment is configured to bind the new modelContext key to the container's mainContext property [22]. It is via this automatic binding modelContext can track changes and persist edit through the ModelContainer.

At this stage, application can store data. However, data is not directly communicated to container, model context works as a conduit to get input data from views or to fetch saved data via container to display in view. Subsequent sections will elaborate on model context and its usage. Figure 13 depicts four steps required in adopting SwiftData in an application. Fifth step, accessing model context is discussed under Many-to-Many relationship sub-chapter.

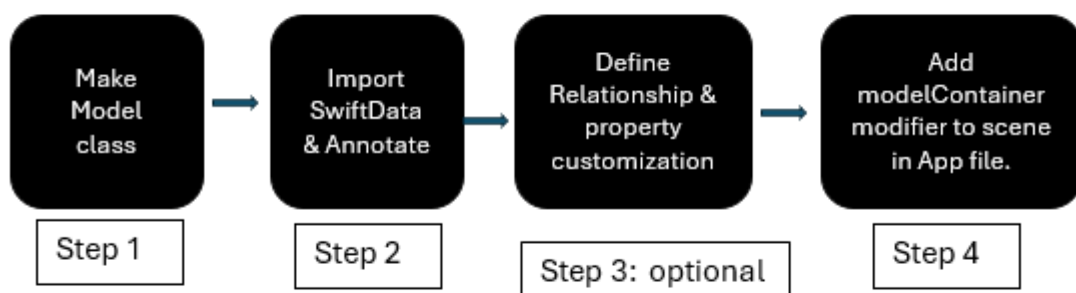
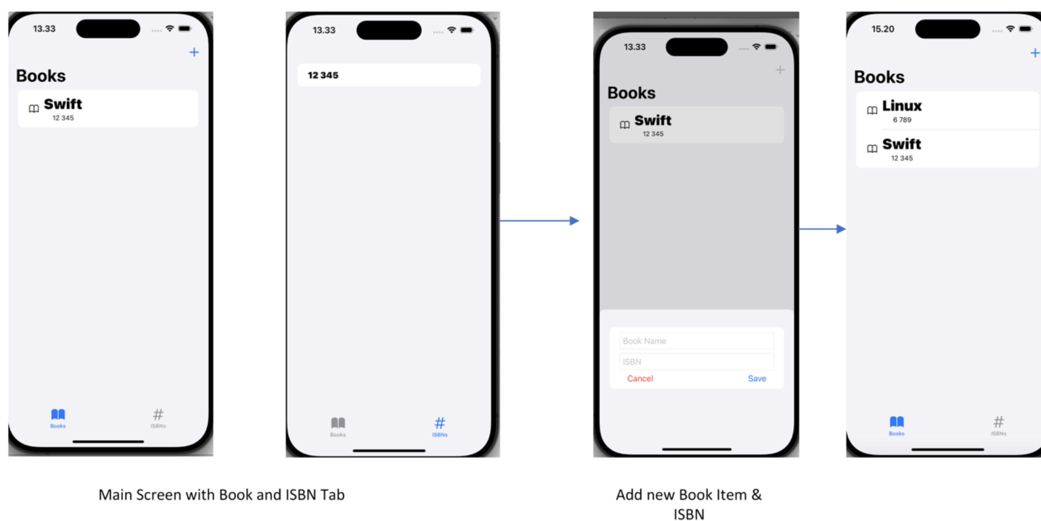


Figure 13. Four steps process to adopt SwiftData.



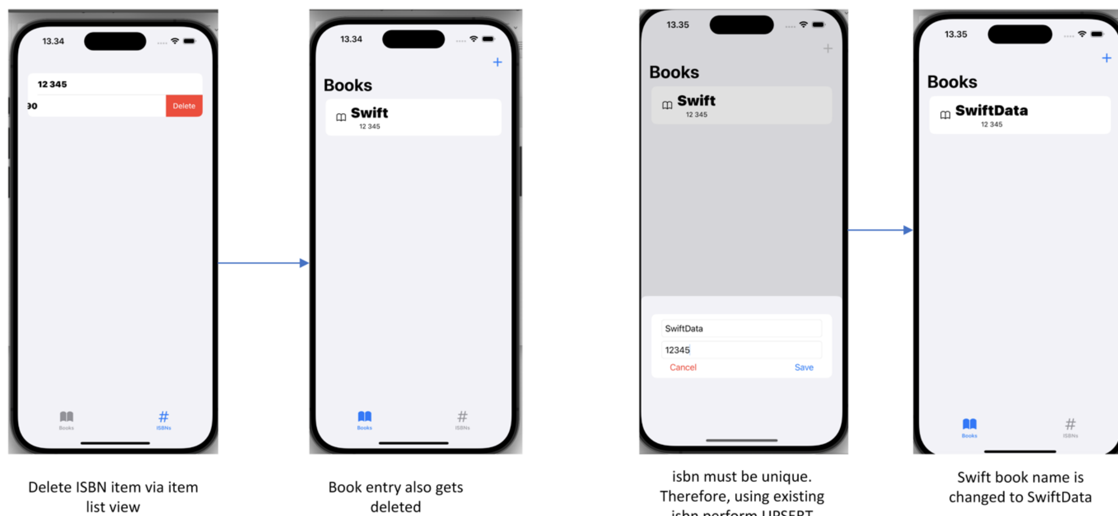


Figure 14. One-to-One relationship Book app workflow.

5.2 One-To-Many relationship

A one-to-many relationship is established between two entities when a single record in the primary table can be linked to multiple records in the secondary table. Conversely, each record within the secondary table can only be associated with a single record in the primary table [23].

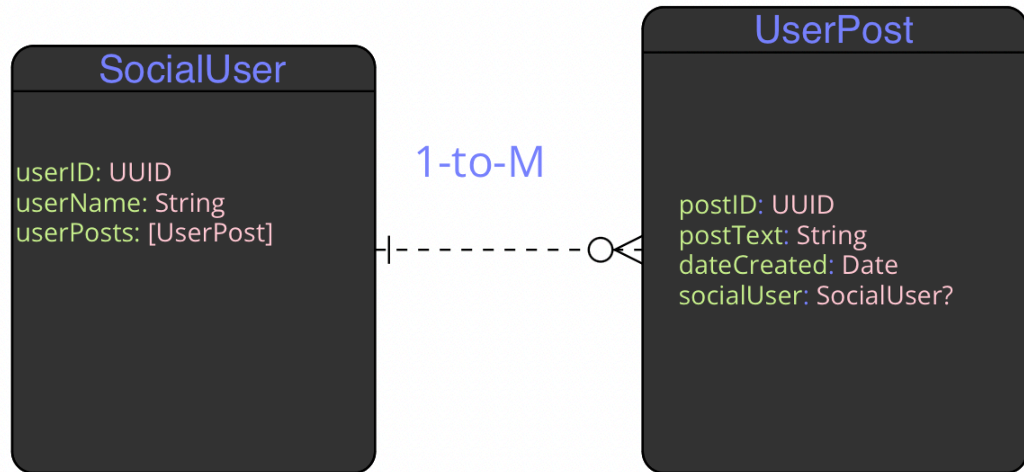


Figure 15. One-To-Many relationship Entity

In SwfitData this relationship is represented using Arrays for child properties (figure 16). In above model (figure 15) a social user can have one or more associated posts, however, a single post object is only associated with one user. SwiftData can implicitly determine this relationship as long as one property is an array and its corresponding property in other model is an optional type [27].

```swift
import Foundation
import SwiftData

@Model
class SocialUser {
    var userId: UUID = UUID()
    var userName: String

    @Relationship( inverse: \UserPost.socialUser)
    var userPosts: [UserPost] = []

    init(userName: String) {

        self.userName = userName

    }

}
```

```swift
import SwiftUI
import SwiftData

@Model
class UserPost {

    var postID: UUID = UUID()
    var postText: String
    var dateCreated: Date = Date.now

    var socialUser: SocialUser?

    init( postText: String, dateCreated: Date = .now, socialUser: SocialUser? = nil) {

        self.postText = postText
        self.dateCreated = dateCreated
        self.socialUser = socialUser
    }

}
```

Figure 16. SwiftData models depicting One-To-Many relationship.

In the above model (figure 16), delete rule is nullify (default) as no rule is explicitly provided. Therefore, SwiftData will delete user object without deleting associated posts in corresponding model as depicted in figure 17.

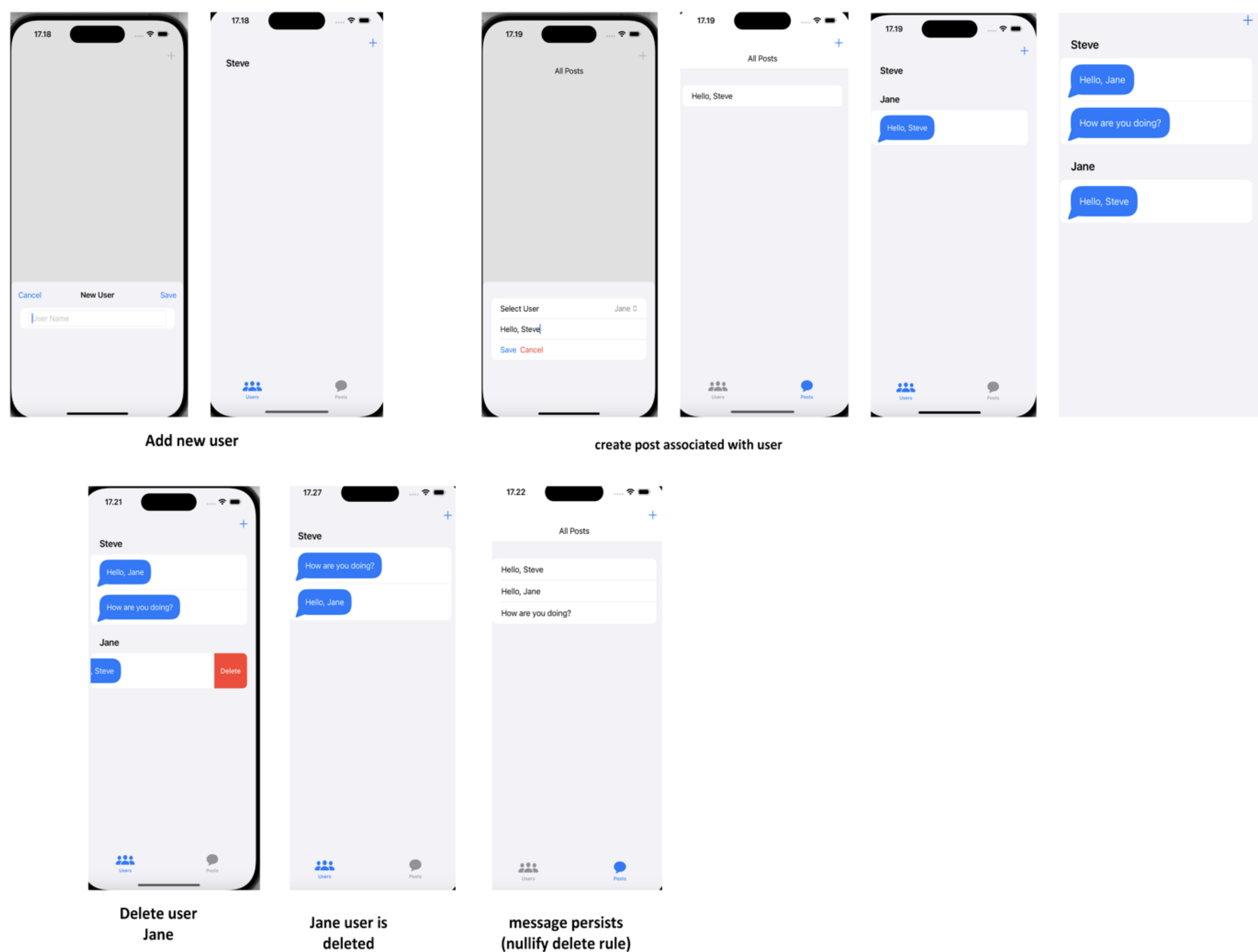


Figure 17. Messaging App, workflow representing 1-to-Many relationship.

Furthermore, as the schema has relationship information, relational query can be made using the parent model. Parent model (SocialUser) in this case has all the necessary information to display post content described in child (UserPost) model. Below (figure 18) is the code snippet demonstrating this relationship query.

```
List{
    ForEach(users){ user in

        Section("\(user.userName)"){
            ForEach(user.userPosts){post  in
                Text(post.postText)
```

Figure 18. Accessing associated records in child model

5.3 Many-To-Many Relationships

In Many-to-Many relationship, a record in one model can be associated with one or more record in corresponding model [23]. In this demo application, a diary entry can be associated with multiple tag entries and a single tag can be attached to many diary entries as depicted in figure 19.

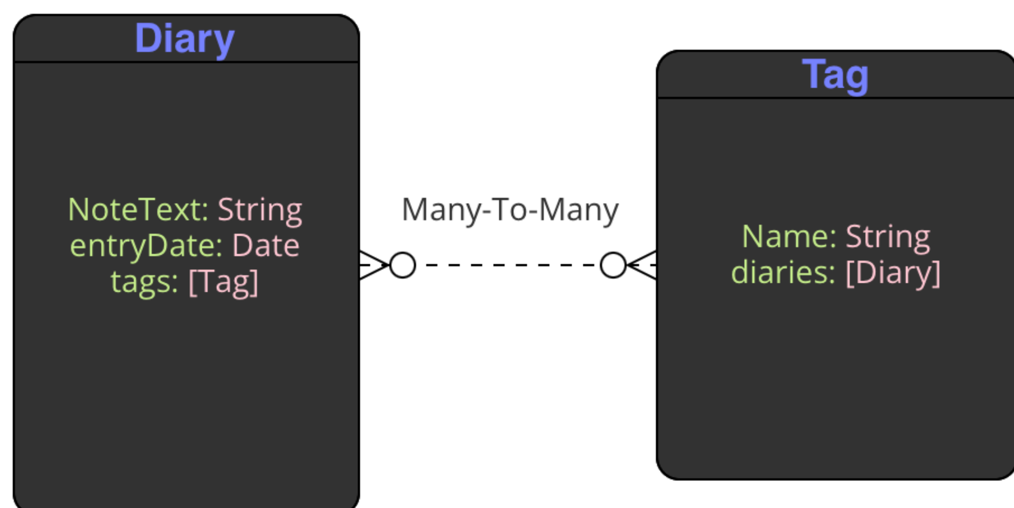


Figure 19. Many-To-Many relationship entity

Many-to-Many relationship can be expressed in Swift in terms of Arrays on both sides of the relationship. However, SwiftData doesn't infer many-to-many relationships implicitly. Therefore, inverse relationship must be explicitly established on one side of the model as depicted in Figure 20.

```
import Foundation
import SwiftData

@Model
final class Diary {

    var noteText: String
    var entryDate: Date

    @Relationship(deleteRule: .nullify, inverse: \Tag.diaries)
    var tags = [Tag]()


    init(noteText: String, entryDate: Date = .now) {
        self.noteText = noteText
        self.entryDate = entryDate
    }
}
```

```
8  import Foundation
9  import SwiftData
10
11 @Model
12 class Tag {
13
14     var name: String
15     var diaries = [Diary]()
16
17     init(name: String) {
18         self.name = name
19     }
20 }
```

Figure 20. SwiftData model with explicit inverse relationship.

Figure 21. Learning Diary app workflow representing Many-To-Many relationships.

Figure 21 depicts app workflow of a demo application named learning diary. This app is built to explore CRUD functionalities, basic sorting, filtering, search, and light migrations of a data model. However, before data creation and storage, a SwiftData app needs access to model context. As discussed in theoretical section, context act as a workspace to perform CRUD operations and it communicates with container to save data in persistence storage. This is the fifth step in enabling SwiftData ready application as depicted in figure 22. Following sections will elaborate on each of these topics.

5.3.1 Accessing Model Context

As discussed earlier, context is the workspace where data is accessed and manipulated before it is saved in persistence storage via container. Attaching of modelContainer modifier to view/Scene automatically provide us access to container's context via @Enviornment property wrapper. It can be accessed by importing SwiftData library in respective app view and accessing environment property as depicted in the figure 22 below.

Figure 22. Accessing Model Context

Besides the context, environment property also provide access to other useful system wide environment keys such as dismiss, it dismisses the current presentation and is widely used to return control in save and cancel action buttons.

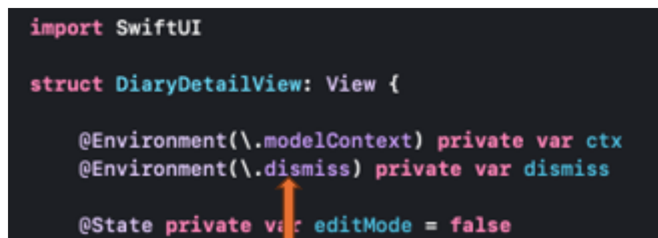


Figure 23. Accessing dismiss Environment Key path.

5.3.2 CRUD (Create)

Adding new entries to a database requires creating respective objects and calling context's insert method to save the object in persistent storage as depicted in figure 24. SwiftData context's has default autosave enabled, that is data is saved on insert call without explicitly calling context's save method. This example makes explicit save call to catch errors in development.

In context of many-to-many relationship, at least one side of the relationship should be initialized and inserted first. Although, each side can exist independently in certain cases it could cause problems where there is tight coupling [28]. Moreover, it is sufficient to insert one model object. SwiftData can insert corresponding entries based on the attribute's relationship. In this case, tag object is populated via diary model entry.

```
Button("Save"){
    let newDiary = Diary(noteText:noteText)
    newDiary.tags = Array(attachedTags)
    ctx.insert(newDiary)
    do {
        try ctx.save()
    }catch {
        print(error.localizedDescription)
    }
    dismiss()
}
```

Figure 24. Creating new record

5.3.3 CRUD (Read)

Reading items from database is straight forward process of using query macro to fetch objects from the database as depicted in figure 25.

```
struct TagListView: View {
    @Environment(\.modelContext) private var ctx
    @Query(sort: \Tag.name) private var tags:[Tag]
```

```
import SwiftUI
import SwiftData

struct DiaryListView: View {
    @Environment(\.modelContext) private var modelContext
    @Query private var diaries: [Diary]
```

Figure 25. Query macro to fetch data.

Query macro also provide options for sorting and filtering. In above code snippet (figure 25), Tag names are queried and presented in ascending order. Sorting order can be defined in order parameter, reverse option organizes items in descending order as depicted in figure 26.

```
@Query(sort: \Tag.name, order: .reverse) private var tags:[Tag]
```

Figure 26. Query macro with sorting in descending order.

Subsequently, “tags” variable can be iterated in a list view to display individual record as shown in figure 27.

```
NavigationStack{
    List {
        ForEach(tags){ tag in
            NavigationLink(value: tag){
                VStack(alignment: .leading){
                    Text(tag.name)
                    Text(" Attached to \(tag.diaries.count) entries")
                        .font(.caption)
                }

            }
```

Figure 27. Iterating through Query variable in list view

Figure 28. provides conceptual overview of Query macro. Query loads the records from database in a collection object that can be consumed in a view to access and manipulate data in a database.

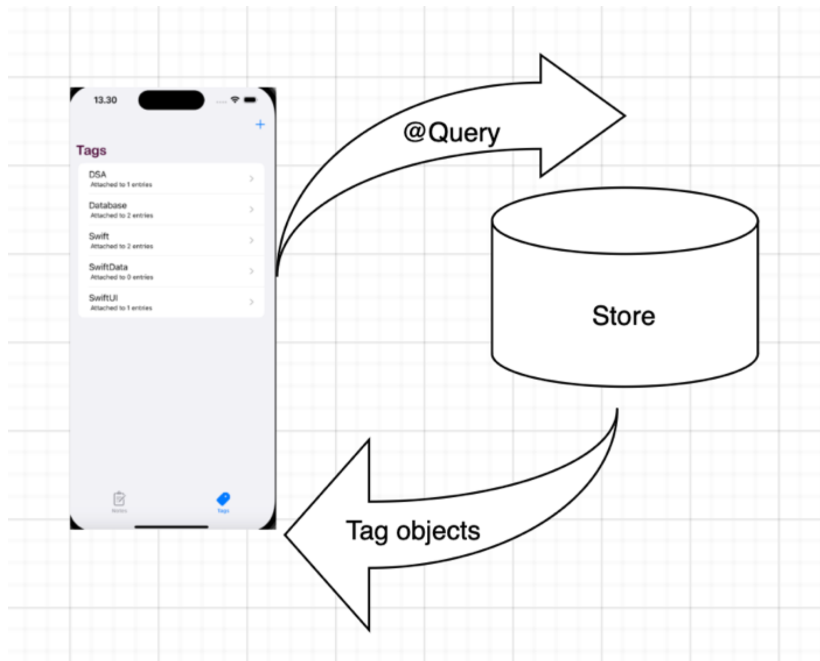


Figure 28. Query loads records from database.

5.3.4 CRUD (Update)

Updating a record is a multistep process of accessing an existing record, making edits, and saving the result. This can be achieved by accessing internal backing properties (_propertyName) of each of these state properties and initializing it with passed data as depicted in figure 29.

```
init(diary:Diary){
    self.diary = diary
    self._note = State.init(initialValue: diary.noteText)
    self._attachedTags = State.init(initialValue: Set(diary.tags))

}
```

Figure 29. Accessing backed properties in initializer

Thereafter, edits can be made in a view and context's save method can be invoked to save the data.

5.3.5 CRUD (Delete)

Deleting a record is straightforward. Delete action can be triggered from a view for instance, by attaching an onDelete modifier to a list cell as depicted in figure 30.

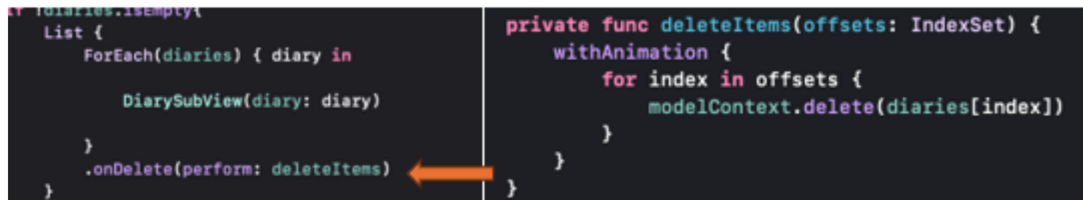


Figure 30. onDelete modifier and delete function.

deleteItems function takes an IndexSet; a collection of integers representing indexes of elements in another collection. It iterates through this IndexSet and delete the item at a given index. It is not required to call save method of context at the end of delete operation.

5.4 Migration

As the app development progresses, schema evolves as per the app requirements. SwiftData adopts lightweight migration and custom migration strategies for complex scenarios when changes are made to model schema.

In lightweight migration SwiftData automatically migrate persisted data to newer schema without any intervention. Renaming an attribute, adding computed properties or Transient properties doesn’t require formal versioning and Schema migration plan as depicted in figure 31 [29].

However, in a production app it is recommended to create custom migration plan as users may not perform timely updates and even for lightweight migrations, SwiftData doesn’t offer automatic migrations with gaps in schema versions. For instance, user data will not automatically migrate from version 1 to version 6 while skipping all the versions in between.

Custom migration will entail providing schema versions and migration plan. Developer must also provide logic to tackle scenarios in case of conflicts and omissions. This thesis doesn't cover custom migration plans.

```
import Foundation
import SwiftData

@Model
final class Diary {

    @Attribute(originalName:"noteText")
    var noteT: String

    var entryDate: Date

    @Relationship(deleteRule: .nullify, inverse: \Tag.diaries)
    var tags = [Tag]()


    init(noteText: String, entryDate: Date = .now) {
        self.noteT = noteText
        self.entryDate = entryDate
    }

}
```

```
import Foundation
import SwiftData

@Model
class Tag {

    var name: String
    var diaries = [Diary]()

    @Transient
    var tagCounts: Bool{
        diaries.count > 0
    }

    init(name: String) {
        self.name = name
    }
}
```

Figure 31. Lightweight migration: Rename property, add computed property. Transient macro doesn't persist data.

5.5 Filter & Search

Besides Sorting, Query is capable of Filtering data as well. As can be observed in figure 32, filter parameter expects a Predicate macro. Predicate is a logical condition evaluating to true or false, used to test values in searching and filtering operations [30].

The predicate in figure 32, returns only tags that are associated with at least one diary entry. Therefore, when a new tag is created it will not display in filtered tag list until it has been assigned to a diary entry.

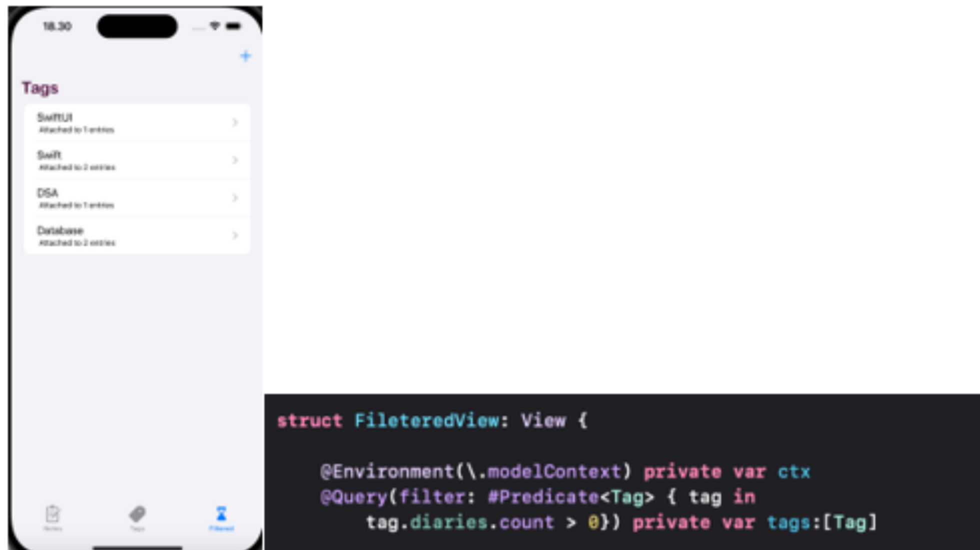


Figure 32 Display tags with at least one diary entry.

In SwiftData search function is implemented by adding searchable modifier to a view. It displays search field to make queries. Context's fetch method is invoked based on the input (searchKey), and it returns an array of models that match the specified criteria as indicated in figure 33.

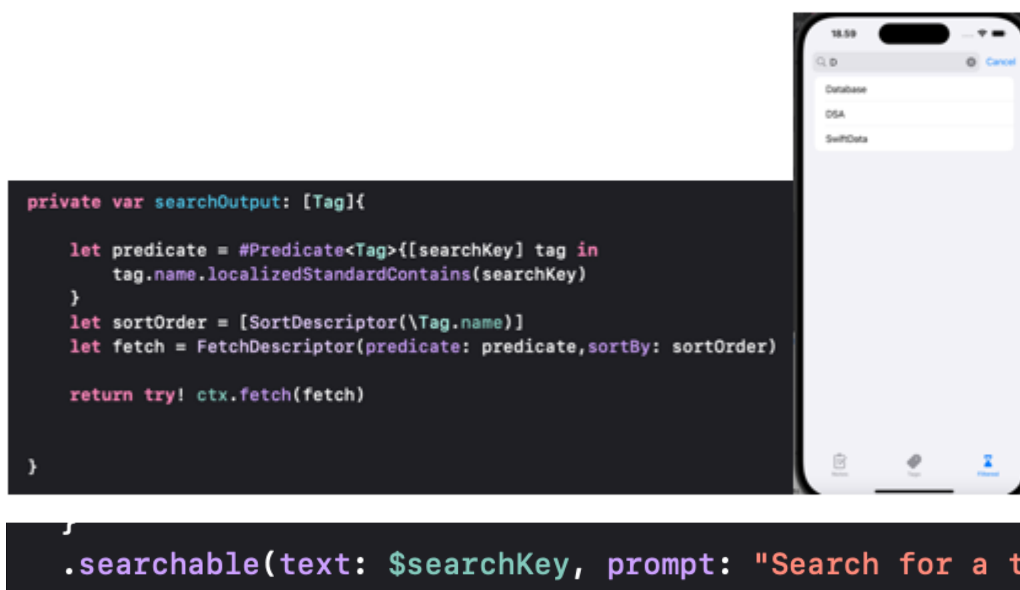


Figure 33 Searchable modifier and fetch method

6 Conclusion & Future Research

SwiftData integrates seamlessly with SwiftUI Application. The fact that models are regular swift classes decreases the distance between code and data model definition. As explored in the thesis, adopting SwiftData in an application is a 5-step process. Four steps to enable SwiftData and fifth step is to manipulate and interact with data. These five steps are illustrated below (figure 34):

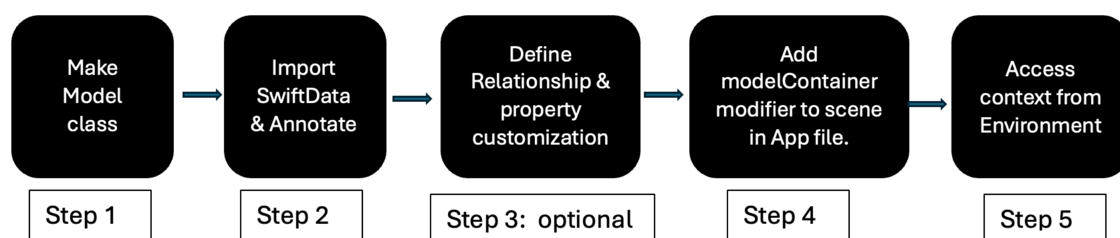


Figure 34: Five step process to adopt SwiftData

However, due diligence is required while designing models. Although, third step of defining relationships is not mandatory for some relationships for example, in one-to-one and one-to-many relationship, it is recommended to explicitly declare relationships to enhance readability and validate intent. Explicit declaration will also assist in spotting logical errors and is mandatory for defining delete rules.

In addition, SwiftData provides easy mechanisms for searching, sorting, and filtering data within an application. Query macro has in-built sort and filter functions that returns collection of objects based on specified criteria. Likewise, context's fetch method is well equipped to integrate search functionality in an app. Moreover, SwiftData provides unattended automatic lightweight migrations and custom migrations for complex schema evolution.

Nevertheless, as the framework is evolving and since each app have its distinct requirements, it is bound to encounter limitations of the framework. Although, during the research and development of demo applications, no limitations were encountered except for few related to SwiftUI.

One of the main limitations of SwiftData is that it only supports iOS 17 onwards and most commercial application tends to maintain backward compatibility for at least three previous versions. Hudson has listed few of the limitations in comparison to Core Data [31].

Therefore, detailed evaluation of SwiftData's functionality based on specific application needs is recommended before adopting it for new projects. Furthermore, a comparative study of SwiftData and Core Data framework might assist in decision making for companies considering adopting SwiftData for newer projects, migrating from Core Data to SwiftData would assist companies planning migration to SwiftData or custom migrations of complex data models in SwiftData to further understand migrations in SwiftData. In Addition, a study exploring SwiftData in networking environment is crucial for real-world production application.

References

1 https://www.apple.com/newsroom/2007/01/09Apple-Reinvents-the-Phone-with-iPhone/. Accessed on: 14/04/2024.

2 https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html. Accessed on: 14/04/2024.

3 https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Cocoa.html. Accessed on: 14/04/2024.

4 https://en.wikipedia.org/wiki/Swift_(programming_language)#:~:text=Swift%20was%20first%20released%20in,version%206%2C%20released%20in%202014.&text=Chris%20Lattner%2C%20Doug%20Gregor%2C%20John,Joe%20Groff%2C%20and%20Apple%20Inc. Accessed on: 14/04/2024.

5 https://developer.apple.com/swift/. Accessed on: 14/04/2024.

6 https://media.defense.gov/2023/Apr/27/2003210083/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY_V1.1.PDF. Accessed on: 14/04/2024.

7 https://www.hackingwithswift.com/quick-start/swiftdata/swiftdata-vs-core-data. Accessed on: 14/04/2024.

8 https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/ObjectGraph.html. Accessed on: 14/04/2024.

9 https://learning.oreilly.com/library/view/objective-c-recipes-a/9781430243717/Chapter09.html#s497-497. Accessed on: 3/05/2024.

10 https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/Performance.html. Accessed on: 16/04/2024.

11 https://developer.apple.com/videos/play/wwdc2023/10187/. Accessed on: 16/04/2024.

12 https://docs.swift.org/swift-book/documentation/the-swift-programming-language/macros/. Accessed on: 17/04/2024.

13 https://www.ibm.com/topics/data-modeling#:~:text=Data%20modeling%20is%20the%20process,between%20data%20points%20and%20structures. Accessed on: 24/04/2024.

14 https://developer.apple.com/documentation/swiftui/managing-model-data-in-your-app#. Accessed on: 24/04/2024.

15 https://developer.apple.com/tutorials/develop-in-swift/save-data. Accessed on: 12/03/2024.

16 https://developer.apple.com/documentation/swiftdata/preserving-your-apps-model-data-across-launches# . Accessed on: 17/03/2024.

17 https://docs.swift.org/swift-book/documentation/the-swift-programming-language/properties/ .Accessed on: 20/04/2024.

18 https://stackoverflow.com/questions/77218060/swiftdata-filtering-query-with-computed-property. Accessed on: 20/04/2024.

19 https://www.kodeco.com/books/swift-cookbook/v1.0/chapters/4-use-codable-protocol-in-swift. Accessed on: 20/04/2024.

20 https://developer.apple.com/documentation/swiftdata/modelcontainer. Accessed on: 20/03/2024.

21 https://developer.apple.com/documentation/swiftdata/modelconfiguration . Accessed on: 20/03/2024.

22 https://developer.apple.com/documentation/swiftdata/modelcontext. Accessed on: 20/03/2024.

23 https://learning.oreilly.com/library/view/database-design-for/9780133122282/ch10.html#ch10lev1sec3. Accessed on: 27/04/2024.

24 https://developer.apple.com/documentation/swiftdata/relationship(_:deleterule:minimummodelcount:maximummodelcount:originalname:inverse:hashmodifier:). Accessed on: 26/03/2024.

25 https://developer.apple.com/documentation/swiftdata/schema/relationship/deleterule-swift.enum. Accessed on: 26/03/2024.

26 https://developer.apple.com/documentation/swiftdata/schema/attribute/option. Accessed on: 26/03/2024.

27 https://www.hackingwithswift.com/quick-start/swiftdata/how-to-create-one-to-many-relationships. Accessed on: 28/03/2024.

28 https://www.hackingwithswift.com/quick-start/swiftdata/how-to-create-many-to-many-relationships. Accessed on: 28/03/2024.

29 https://www.hackingwithswift.com/quick-start/swiftdata/lightweight-vs-complex-migrations. Accessed on: 23/04/2024.

30 https://developer.apple.com/documentation/foundation/predicate. Accessed on: 14/04/2024.

31 https://www.hackingwithswift.com/quick-start/swiftdata/what-is-swiftdata#:~:text=One%20downside%20is%20that%20SwiftData,watchOS%2010%2C%20and%20visionOS%201.0. Accessed on: 03/05/2024.

Source Code for Book App

Below is project Structure of Book App.

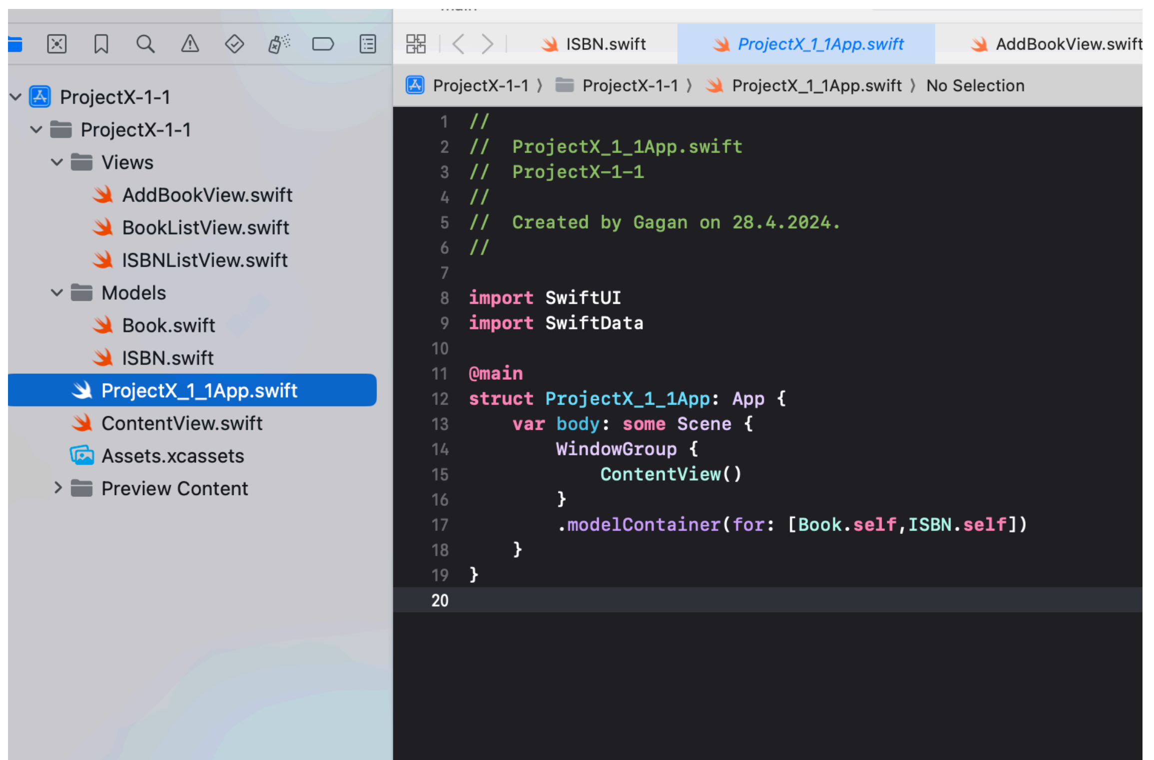


Figure 1. Book App project structure

ContentView.Swift is the entry point of application. Models Group and view group contains their respective models and views files.

```
//  AddBookView.swift
//  ProjectX-1-1
//
//  Created by Gagan on 28.4.2024.
//

import SwiftUI
import SwiftData

struct AddBookView: View {

    @Environment(\.modelContext) private var ctx
    @Environment(\.dismiss) private var dismiss


    @State private var name: String = ""
    @State private var isbn: Int?

    var body: some View {
        NavigationStack{
            Form{
                VStack {
                    TextField("Book Name", text: $name)
                        .textFieldStyle(.roundedBorder)
                    TextField("ISBN", value: $isbn, format: .number)
                        .keyboardType(.numberPad)
                        .textFieldStyle(.roundedBorder)
                    HStack{
                        Button(role: .destructive) {
                            dismiss()
                        } label: {
                            Text("Cancel")
                        }
                        Spacer()
                        Button("Save"){
                            if let isbn = isbn {
                                let newISBN = ISBN(isbn: isbn)
                                let newbook = Book(name: name,isbn: newISBN)
                                ctx.insert(newbook)
                            } else {
                                let newbook = Book(name: name)
                                ctx.insert(newbook)
                            }

                            do {
                                try ctx.save()
                                print("Data Saved")
                            }catch{
                                print(error.localizedDescription)
                            }
                            dismiss()

                        }

                    }
                    .padding(.horizontal)

                }
            }

        }
    }
}
```

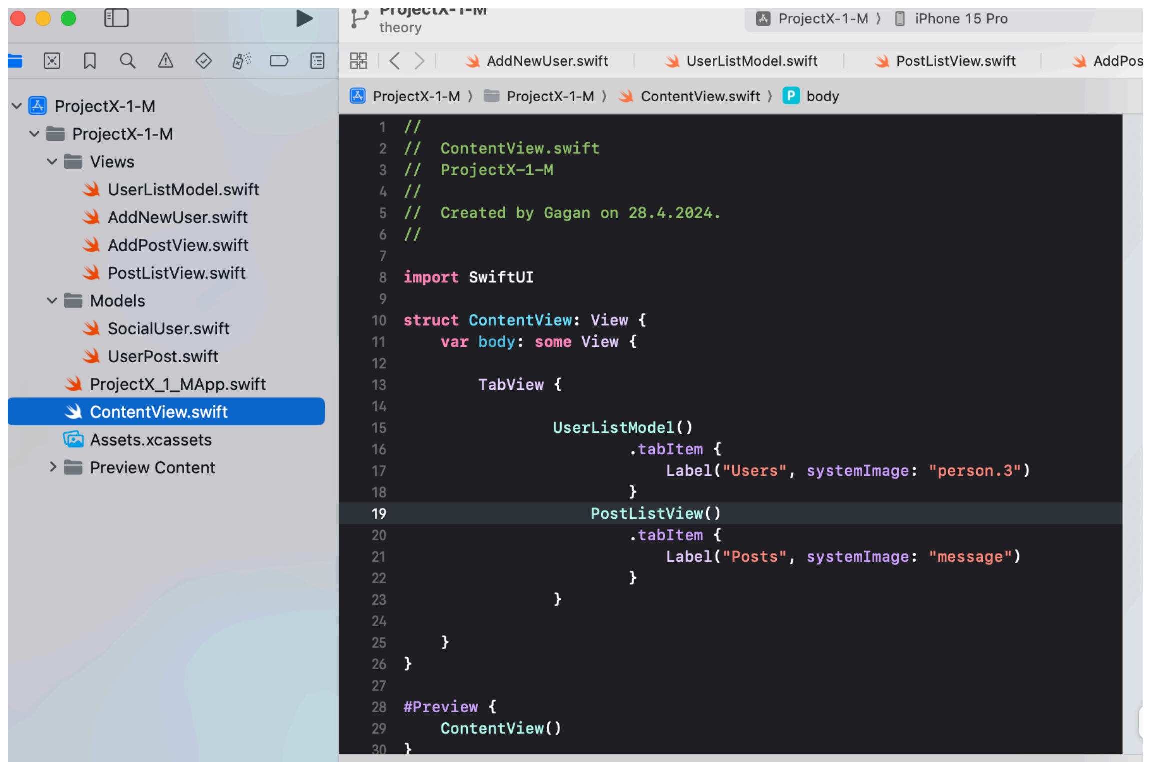
```
//
//  BookListView.swift
//  ProjectX-1-1
//
//  Created by Gagan on 28.4.2024.
//

import SwiftUI
import SwiftData

struct BookListView: View {

    @Environment(\.modelContext) private var ctx

    @Query(sort: \Book.name) private var books: [Book]

    @State private var  showAddNew = false

    var body: some View {
        NavigationStack {
            List{
                ForEach(books){ book in
                    BookSubView(book: book)
                }
                .onDelete(perform: deleteItems)

            }

            .navigationTitle("Books")
            .toolbar{
                ToolbarItem(placement: .confirmationAction){
                    Button{
                        showAddNew.toggle()

                    }label: {
                       Image(systemName: "plus")
                    }
                }
            }
            .sheet(isPresented: $showAddNew, content: {
                AddBookView()
                    .presentationDetents([.fraction(0.3)])
            })
        }
    }
    private func deleteItems(offsets: IndexSet) {
        withAnimation {
            for index in offsets {
                ctx.delete(books[index])
            }
        }
    }

}

#Preview {
    BookListView()
}

struct BookSubView: View {
    let book: Book
    var body: some View {
        HStack {
            Image(systemName: "book")
            VStack{
                Text(book.name)
                    .font(.title)
```

```
                        .fontWeight(.black)
                    if let isbn = book.isbn {
                        Text("\(isbn.isbn)")
                            .font(.footnote)
                    } else {
                        Image(systemName: "number")
                    }
                }
            }
        }

}
//
//  ISBNListView.swift
//  ProjectX-1-1
//
//  Created by Gagan on 28.4.2024.
//

import SwiftUI
import SwiftData

struct ISBNListView: View {
    @Query private var isbns:[ISBN]
    @Environment(\.modelContext) private var ctx


    var body: some View {
        NavigationStack{
            List{
                ForEach(isbns){ isbn in
                    Text("\(isbn.isbn)")
                        .fontWeight(.black)
                }
                .onDelete(perform: deleteItems)
            }
        }
    }
    private func deleteItems(offsets: IndexSet) {
        withAnimation {
            for index in offsets {
                ctx.delete(isbns[index])
            }
        }
    }
}

#Preview {
    ISBNListView()
}

//  Book.swift
//  ProjectX-1-1
//
//  Created by Gagan on 28.4.2024.
//

import Foundation
import SwiftData

@Model
class Book {

    var name: String

    @Relationship( deleteRule: .cascade, inverse: \ISBN.book)
    var isbn: ISBN?
```

```
    init(name: String, isbn: ISBN? = nil) {
        self.name = name
        self.isbn = isbn
    }


}
//
//  ISBN.swift
//  ProjectX-1-1
//
//  Created by Gagan on 28.4.2024.
//

import Foundation
import SwiftData

@Model
class ISBN {
    @Attribute(.unique)
    var isbn: Int
    @Relationship(deleteRule: .cascade)
    var book: Book?

    init(isbn: Int, book: Book? = nil) {
        self.isbn = isbn
        self.book = book
    }


}
//
//  ContentView.swift
//  ProjectX-1-1
//
//  Created by Gagan on 28.4.2024.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        TabView {
            BookListView()
                .tabItem {
                    Label("Books", systemImage: "book")
                }
            ISBNListView()
                .tabItem {
                    Label("ISBNs", systemImage: "number")
                }
        }
    }
}

#Preview {
    ContentView()
}
```

Source Code for messaging App.

Below is project Structure of messaging App.



ContentView.swift is the entry point of application. Views and models groups contains their respective files.

```
//
//  SocialUSer.swift
//  ProjectX-1-M
//
//  Created by Gagan on 29.4.2024.
//

import Foundation
import SwiftData

@Model
class SocialUser {
    var userId: UUID = UUID()
    var userName: String

    @Relationship( inverse: \UserPost.socialUser)
    var userPosts: [UserPost] = []

    init(userName: String) {

        self.userName = userName

    }


}
//
//  UserPost.swift
//  ProjectX-1-M
//
//  Created by Gagan on 29.4.2024.
//

import SwiftUI
import SwiftData

@Model
class UserPost {

    var postID: UUID = UUID()
    var postText: String
    var dateCreated: Date = Date.now

    var socialUser: SocialUser?


    init( postText: String, dateCreated: Date = .now, socialUser: SocialUser?
= nil) {

        self.postText = postText
        self.dateCreated = dateCreated
        self.socialUser = socialUser
    }


}
//
//  UserListModel.swift
//  ProjectX-1-M
//
//  Created by Gagan on 29.4.2024.
//

import SwiftUI
import SwiftData

struct UserListModel: View {
```

```
    @Environment(\.modelContext)private var ctx
    @Query private var users: [SocialUser]
    @Query private var posts: [UserPost]
    @State private var showAdd = false

    var body: some View {
        NavigationStack {
            Group{

                List{
                    ForEach(users){ user in

                        Section("\(user.userName)"){
                            ForEach(user.userPosts){post  in
                                Text(post.postText)
                                    .padding()
                                    .background(.blue)
                                    .foregroundStyle(.white)
                                    .clipShape(RoundedRectangle(cornerRadius:
19))

                                    .overlay(alignment: .bottomLeading){
                                        Image(systemName:
"arrowtriangle.down.fill")
                                            .font(.title)
                                            .rotationEffect(.degrees(45))
                                            .offset(x:-10,y:10)
                                            .foregroundStyle(.blue)
                                    }

                            }


                        }


                    }
                    .onDelete(perform: deleteItems)
                }
                .headerProminence(.increased)

            }
            .toolbar{
                ToolbarItem(placement: .topBarTrailing){
                    Button{
                        showAdd.toggle()
                    }label: {
                        Image(systemName: "plus")
                    }
                }
            }
            .sheet(isPresented: $showAdd, content: {
                AddNewUser()
                    .presentationDetents([.fraction(0.30)])
        })
        }

    }
    private func deleteItems(offsets: IndexSet) {
           withAnimation {
               for index in offsets {
                   ctx.delete(users[index])
               }
           }
       }
    }
```

```
#Preview {
    UserListModel()
}
//
//  AddNewUser.swift
//  ProjectX-1-M
//
//  Created by Gagan on 29.4.2024.
//

import SwiftUI
import SwiftData

struct AddNewUser: View {

    @Environment(\.modelContext) private var ctx
    @Environment(\.dismiss) private var dismiss

    @State  var userName: String = ""
    @State  var postText: String = ""


    var body: some View {
        NavigationStack{
            Form{
                VStack{
                    TextField("User Name", text: $userName)
                        .textFieldStyle(.roundedBorder)

//                    TextField("Post", text:$postText)
//                        .textFieldStyle(.roundedBorder)
                }


            }
            .navigationTitle("New User")
            .toolbar{
                ToolbarItem(placement: .topBarTrailing){
                    Button("Save"){
                        let newUser = SocialUser(userName: userName)
//                        newUser.userPosts.append(UserPost(postText:
postText))
                        ctx.insert(newUser)
                        do {
                            try ctx.save()
                        }catch {
                            print(error.localizedDescription)
                        }
                        dismiss()
                    }
                }
                ToolbarItem(placement: .topBarLeading){
                    Button("Cancel"){
                    dismiss()
                    }
                }


            }

        }
    }
}

#Preview {
    AddNewUser(userName: "User One")
}
```

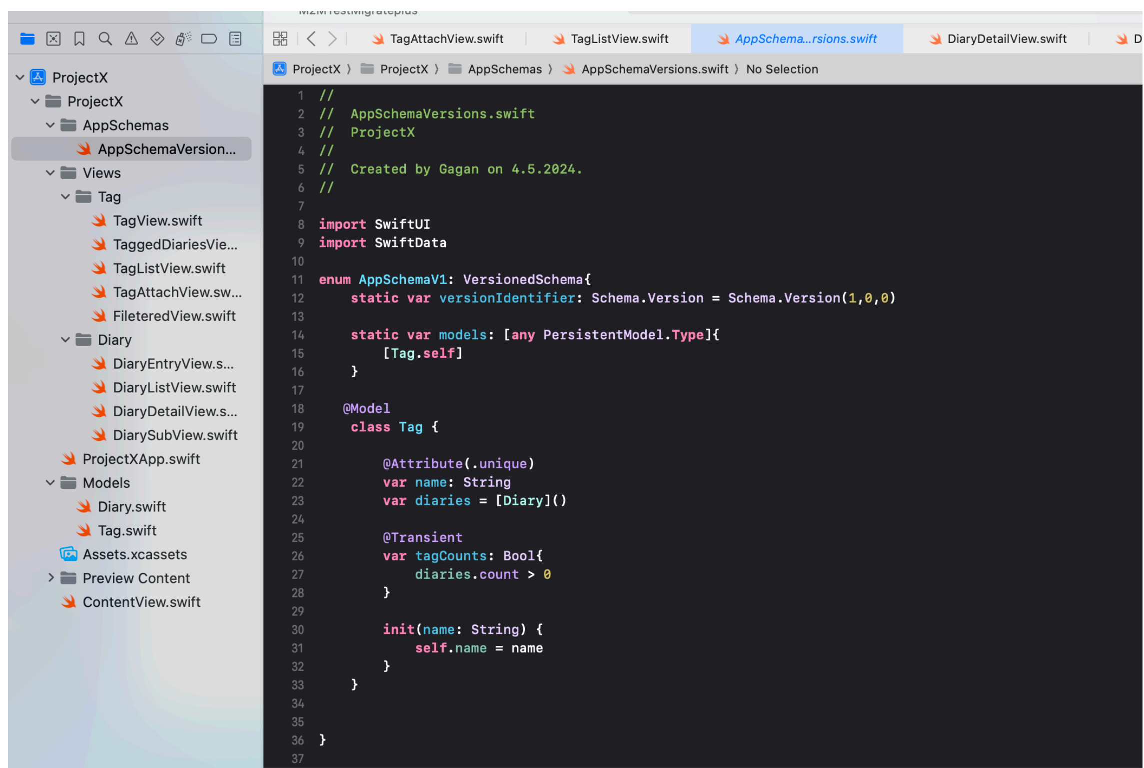
```
//
//  AddPostView.swift
//  ProjectX-1-M
//
//  Created by Gagan on 29.4.2024.
//

import SwiftUI
import SwiftData

struct AddPostView: View {

    @Environment(\.modelContext) private var ctx
    @Environment(\.dismiss) private var dismiss

    @Query(sort: \SocialUser.userName) private var socialUsers: [SocialUser]
    @Query private var allPosts: [UserPost]
    @State var selectedUser: SocialUser?
    @State var message: String = ""

    var body: some View {
        NavigationStack{
            Form{
                Picker("Select User", selection: $selectedUser) {
                    ForEach(socialUsers){ user in
                        Text(user.userName)
                            .tag(Optional(user))
                    }
                }
                TextField("Message:", text: $message)
                HStack{
                    Button("Save"){
                        let msg = UserPost(postText: message)
                        msg.socialUser = selectedUser
                        ctx.insert(msg)
                        dismiss()
                        do {
                            try ctx.save()
                        }catch {
                            print(error.localizedDescription)
                        }
                    }
                    Button("Cancel", role: .destructive){
                        dismiss()
                    }
                }

            }
        }
    }
}

//#Preview {
//    AddPostView()
//}
//
//  PostListView.swift
//  ProjectX-1-M
//
//  Created by Gagan on 29.4.2024.
//

import SwiftUI

import SwiftData

struct PostListView: View {
```

```
    @Environment(\.modelContext)private var ctx
    @Query private var posts: [UserPost]
    @State private var showAdd = false

    var body: some View {
        NavigationStack {
            Section("All Posts"){

                List{
                    ForEach(posts){ post in

                        Text(post.postText)

                    }
                    .onDelete(perform: deleteItems)
                }
                .toolbar{
                    ToolbarItem(placement: .topBarTrailing){
                        Button{
                            showAdd.toggle()
                        }label: {
                            Image(systemName: "plus")
                        }
                    }
                }
                .sheet(isPresented: $showAdd, content: {
                    AddPostView()
                        .presentationDetents([.fraction(0.30)])
            })

            }
            .headerProminence(.increased)

        }


    }
    private func deleteItems(offsets: IndexSet) {
           withAnimation {
               for index in offsets {
                   ctx.delete(posts[index])
               }
           }
       }
}



#Preview {
    PostListView()
}
```

Source Code for Learning Diary App.

Below is project Structure of Learning Diary App.

Full source code is not shared here as this application will be developed further into production ready app with inclusion of Artificial intelligence to assist in learning. Models are shared as they are relevant for SwiftData and Schema will change in final app. Furthermore, snippets of other relevant code is already shared in the thesis.



```swift
//
//  Diary.swift
//  ProjectX
//
//  Created by Gagan on 27.4.2024.
//

import Foundation
import SwiftData

@Model
final class Diary {

    @Attribute(originalName:"noteText")
    var noteT: String

    var entryDate: Date

    @Relationship(deleteRule: .nullify, inverse: \Tag.diaries)
    var tags = [Tag]()



    init(noteText: String, entryDate: Date = .now) {
        self.noteT = noteText
        self.entryDate = entryDate
    }



}

//
//  Tag.swift
//  ProjectX
//
//  Created by Gagan on 27.4.2024.
//

import Foundation
import SwiftData

@Model
class Tag {

    var name: String
    var diaries = [Diary]()

    @Transient
    var tagCounts: Bool{
        diaries.count > 0
    }

    init(name: String) {
        self.name = name
    }
}

//
//  ProjectXApp.swift
//  ProjectX
//
//  Created by Gagan on 27.4.2024.
//

import SwiftUI
import SwiftData
```

```
@main
struct ProjectXApp: App {
    var sharedModelContainer: ModelContainer = {
        let schema = Schema([
            Diary.self,Tag.self
        ])
        let modelConfiguration = ModelConfiguration(schema: schema, 
isStoredInMemoryOnly: false)

        do {
            return try ModelContainer(for: schema, configurations: 
[modelConfiguration])
        } catch {
            fatalError("Could not create ModelContainer: \(error)")
        }
    }()

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .modelContainer(sharedModelContainer)
    }

    /*
     NavBar title color customization Adpated from :
     https://stackoverflow.com/questions/77664511/how-to-change-navigation-
title-color-in-swiftui
     Book: Mastering SwiftUI 5. Chapter 11, Page:268. Author: Simon Ng. 2023.
     Sourced on:  23/4/2024

     */
    init() {
        let navBarAppearance = UINavigationBarAppearance()
        navBarAppearance.largeTitleTextAttributes = [.foregroundColor: 
UIColor(named: "NavigationBarTitle") ?? UIColor.magentoBar, .font: 
UIFont(name: "ArialRoundedMTBold", size: 25)!]
        navBarAppearance.titleTextAttributes = [.foregroundColor: 
UIColor(named: "NavigationBarTitle") ?? UIColor.magentoBar, .font: 
UIFont(name: "ArialRoundedMTBold", size: 15)!]
        navBarAppearance.backgroundColor = .clear
        navBarAppearance.backgroundEffect = .none
        navBarAppearance.shadowColor = .clear

        UINavigationBar.appearance().standardAppearance = navBarAppearance
        UINavigationBar.appearance().scrollEdgeAppearance = navBarAppearance
        UINavigationBar.appearance().compactAppearance = navBarAppearance
    }

}
//
//  ContentView.swift
//  ProjectX
//
//  Created by Gagan on 27.4.2024.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        TabView {
            DiaryListView()
                .tabItem{
                    Label("Notes", systemImage: "pencil.and.list.clipboard")
                }
```

```
            TagListView()
                .tabItem {
                    Label("Tags", systemImage: "tag")
                }
            FileteredView()
                .tabItem{
                    Label("Filtered", systemImage: "magnifyingglass")
                }
        }
    }
}

#Preview {
    ContentView()
}
```