Data Persistence

Lecture 9

CIS 1951

Last time, in CIS 1951…

Networking in iOS

- HTTP requests and response handling (async/await)
- URLSession for network tasks
- Parsing JSON data, Encodable & Decodable
- Error handling and network best practices
- **Questions? Comments? Feedback?**

CIS 1951 as a whole

Lectures 1-6: The Basics

Lectures 7-10: Technologies

- Sensors
- Networking
- Data Persistence

Lectures 11-13: Beyond Development

What is Data Persistence?

Data Persistence

Storing and Managing Data in iOS

- **Definition:** The ability to save data to a permanent storage location, so it can later be retrieved and used
- **Importance:**
    - Enhances user experience by saving user settings, preferences, and state
    - Allows for offline access to data
    - Essential for data-intensive applications

Data Persistence

Storing and Managing Data in iOS

TLDR: We want to be able to REMEMBER stuff

Data Persistence

Storing and Managing Data in iOS

- **Options:**
  - UserDefaults
  - Core Data
  - File Management
  - 3rd Party Libraries (Keychain, SwiftData, etc.)

UserDefaults

UserDefaults

Simple, lightweight storage

- Used to store lightweight user preferences and settings
  - Ideal for saving simple configurations (e.g. volume level, display mode)
  - Not intended for sensitive or large quantities of data

UserDefaults

Simple, lightweight storage

- How large is “too large”?
  - Ideally <1 MB
  - Designed for small pieces of data like booleans, integers, strings, or small arrays and dictionaries

1 Using UserDefaults

```swift
@AppStorage("key") var varName: Type = defaultValue
```

2 Saving Preferences with UserDefaults

```
struct FirstView: View {
    @AppStorage("username") var username: String = ""

    var body: some View {
        // Username is automatically saved
        TextField("Enter your username", text: $username)
            .padding()
    }
}
```

3 Retrieving Preferences with UserDefaults

```swift
struct SecondView: View {
    // Retrieve the username from UserDefaults
    // or use a default value
    @AppStorage("username") var username: String = "DefaultUser"

    var body: some View {
        Text("Welcome back, \(username)!")
            .padding()
    }
}
```

UserDefaults

Best Practices and Limitations

- Ensure default values are set for a better user experience
- Designed for simple data types and small datasets
- Use more secure storage methods for sensitive information
- May lead to clutter and misuse if overused for complex data

Core Data

Core Data

Complex, structured data

- Apple’s native framework for object graph and persistence
- Suitable for complex data models with relationships and extensive data.
- Used in apps requiring data persistence beyond simple preferences

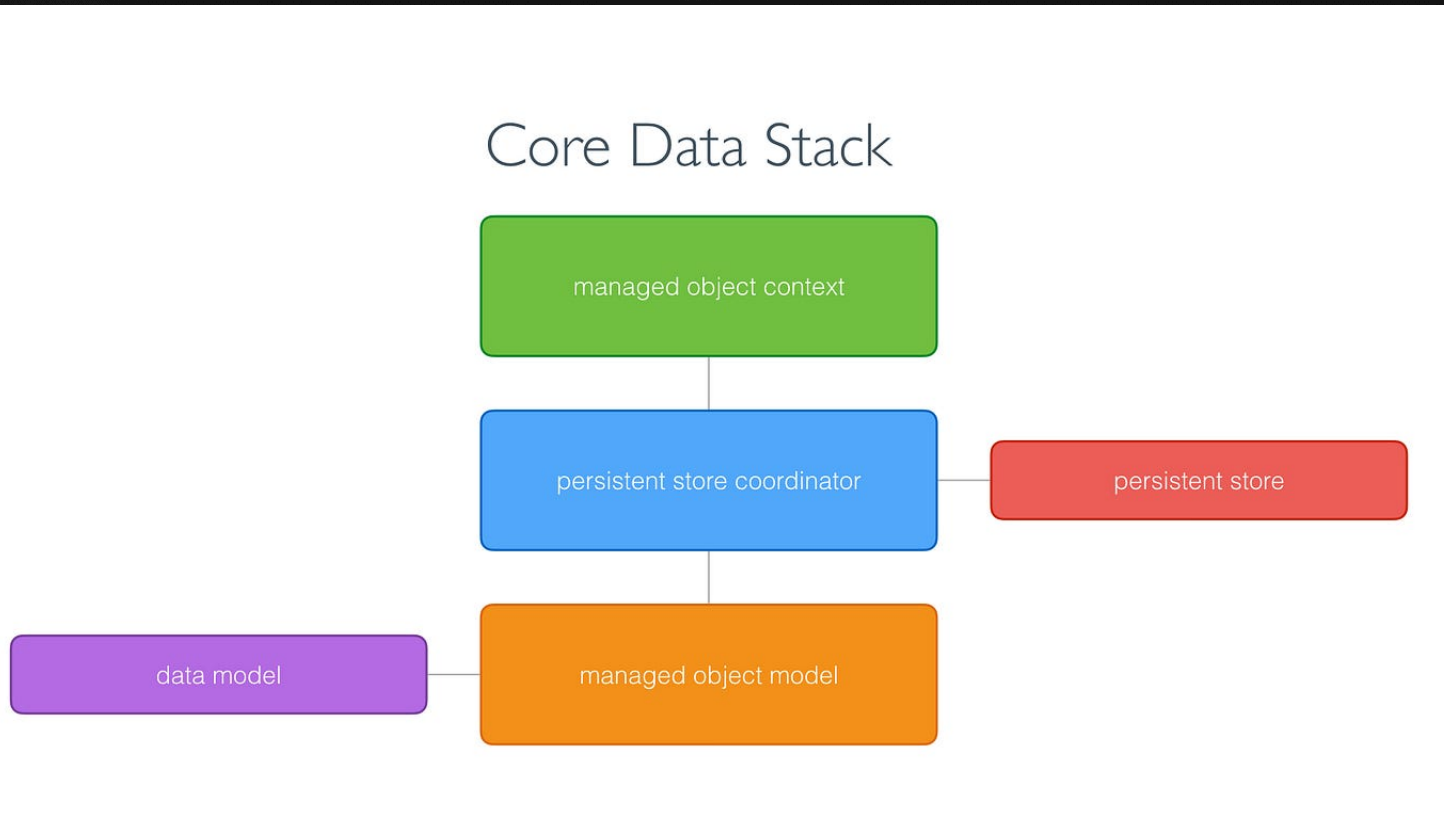
Core Data

Understanding the Pieces

- **Managed Object Context:** The working area for your managed objects, a "scratch pad” in memory
- **Persistent Store Coordinator:** Links the objects in the context to the physical database (e.g., SQLite database)
- **Managed Object Model:** Defines your entities and relationships, typically created from a `.xcdatamodel` file.
- **Persistent Store:** The actual storage location for the data, could be a file on disk or a database

Core Data

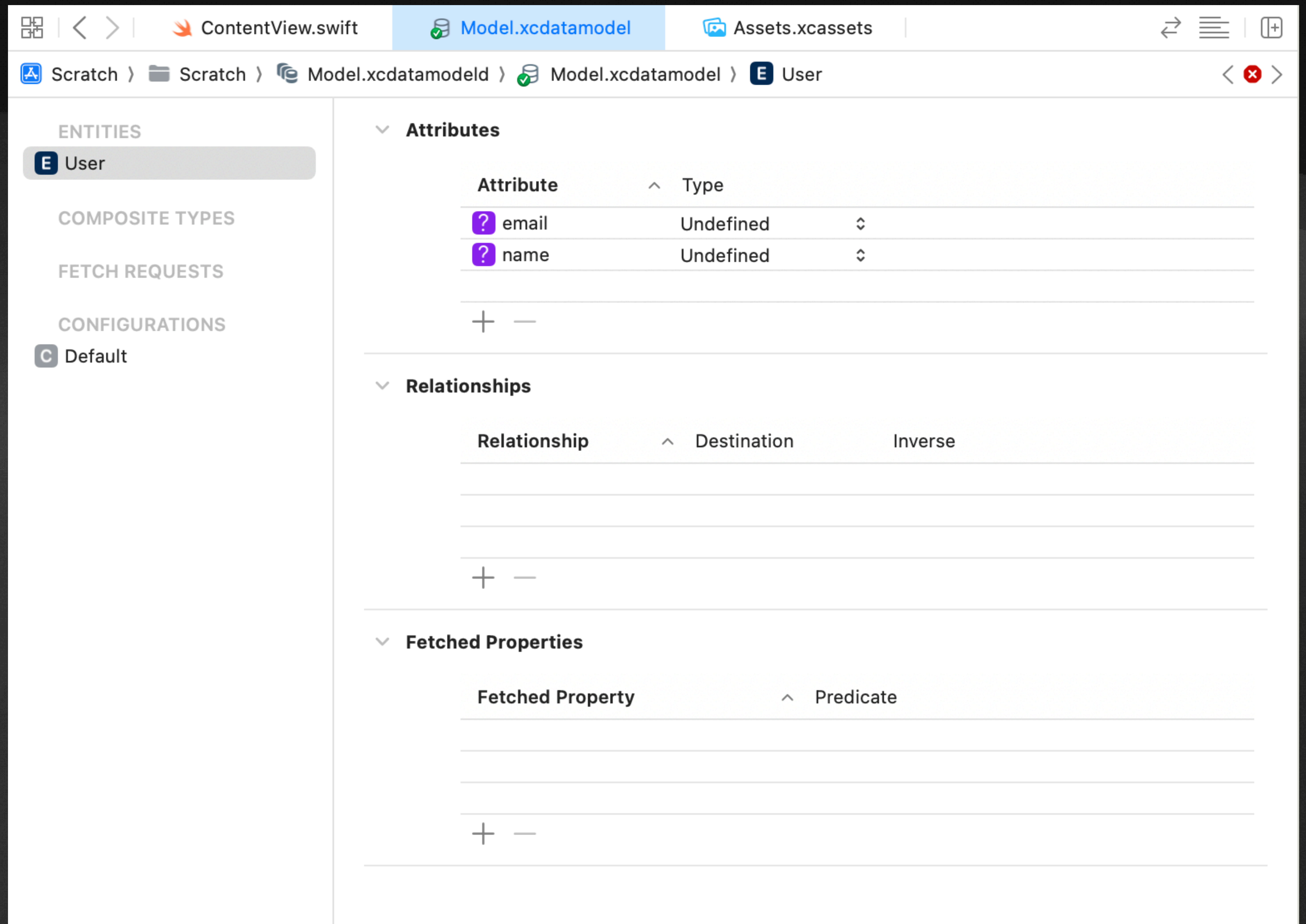
Understanding the Pieces



Core Data: Set Up

Step 1: Create the Data Model

- **Xcode > New File > Data Model**
- Define your entities (i.e. objects) and attributes (i.e. properties)



Core Data: Set Up

Step 2: Initialize the Core Data Stack

```swift
import CoreData

struct PersistenceController {
    static let shared = PersistenceController()

    let container: NSPersistentContainer

    init() {
        container = NSPersistentContainer(name: "User")
        container.loadPersistentStores { (storeDescription, error) in
            if let error = error as NSError? {
                // Error handling...
            }
        }
    }
}
```

Core Data: Set Up

Optional: Configuring Storage Options

```
container = NSPersistentContainer(name: "User")

// Saves data as a binary file instead of SQLite
let description = NSPersistentStoreDescription()
description.type = NSBinaryStoreType
container.persistentStoreDescriptions = [description]

container.loadPersistentStores { …
```

Core Data: Set Up

Step 3: Add Core Data to Your App

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

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

CRUD Operations

What are they?

- **Create:** Adding new records to your database
- **Read:** Fetching existing data
- **Update:** Modifying existing data
- **Delete:** Removing data

CRUD Operations with Core Data

CREATE

```
// Create
let newUser = User(context: managedObjectContext)
newUser.name = "John Doe"
```

CRUD Operations with Core Data

READ

```swift
// Read
// Traditional way - full control, manual management
let fetchRequest = NSFetchRequest<User>(entityName: "User")
let users = try? managedObjectContext.fetch(fetchRequest)
```

CRUD Operations with Core Data

READ

```
// Read
// SwiftUI way - Less control to fetch request details, but seamless integration
struct UserListView: View {
   @FetchRequest(
        entity: User.entity(),
        sortDescriptors: [NSSortDescriptor(keyPath: \User.name, ascending: true)]
  ) var users: FetchedResults<User>

    var body: some View {
        List(users, id: \.self) { user in
            Text(user.name ?? "Unknown")
        }
    }
}
```

CRUD Operations with Core Data

UPDATE

```
// Update
if let firstUser = users.first {
    firstUser.name = "Jane Doe"
}
```

CRUD Operations with Core Data

DELETE

```
// Delete
if let firstUser = users.first {
    managedObjectContext.delete(firstUser)
}
```

CRUD Operations with Core Data

SAVE - Write to DB!

```
// Save Changes
try? managedObjectContext.save()
```

Core Data

Best Practices and Limitations

- Regularly save changes to the Managed Object Context
- Be cautious of memory usage and manage object lifecycles
- Can be complex to set up and manage - not suitable for simple data
- Utilize background contexts for long-running tasks

File Management

File Management

Direct file system access

- Directly reading from and writing to the file system
- Used when storing large documents or binary data that don’t fit into structured databases, non-standard file formats or external files
- Essential for apps that handle media, documents, or require offline content access

1 Writing to a File

```
func saveTextToFile(text: String, fileName: String) {
    let paths = FileManager.default.urls(for: .documentDirectory,
in: .userDomainMask)
    let fileURL = paths[0].appendingPathComponent(fileName)

    do {
        try text.write(to: fileURL, atomically: true, encoding: .utf8)
    } catch {
        // Handle the error
        print("Error saving file: \(error)")
    }
}
```

2 Reading from a File

```
func readTextFromFile(fileName: String) -> String? {
    let paths = FileManager.default.urls(for: .documentDirectory,
in: .userDomainMask)
    let fileURL = paths[0].appendingPathComponent(fileName)

    do {
        let text = try String(contentsOf: fileURL, encoding: .utf8)
        return text
    } catch {
        // Handle the error
        print("Error reading file: \(error)")
        return nil
    }
}
```

File Management

Best Practices and Limitations

- Organize files into appropriate directories
- Handle errors and data integrity during read/write operations
- Regularly back up important data and manage storage usage
- Manual management means higher complexity
- Potential security risks if sensitive data is not properly encrypted

Keychain

Keychain

Secure and sensitive data

- Secure storage for…
    - Sensitive information (e.g. passwords, tokens, and encryption keys)
    - Personal data that must be kept secure
- Protects data even if the device is compromised

1 Saving to Keychain

```swift
import KeychainSwift

func saveToKeychain(key: String, value: String) {
    let keychain = KeychainSwift()
    keychain.set(value, forKey: key)
}
```

2 Reading from Keychain

```swift
import KeychainSwift

func readFromKeychain(key: String) -> String? {
    let keychain = KeychainSwift()
    return keychain.get(key)
}
```

Keychain

Best Practices and Limitations

- Use for small pieces of sensitive data, not large datasets
- Always check for the success or failure of Keychain operations
- Retrieval and storage processes can be slower due to encryption and decryption processes

SwiftData

SwiftData

Flexible for diverse data types

- Similar to Core Data
- Offers a lightweight SQLite database

SwiftData vs. Core Data

Which one do I pick?

Feature	SwiftData	Core Data
Age	Newer	Older
API	More modern and Swift-friendly	More complex and Objective-C-oriented
Efficiency	More efficient	Less efficient
Integration with SwiftUI	Seamless	Not as seamless
Features	Fewer features	More features
Maturity	Less mature	More mature

1 Declaring a Model

```swift
import SwiftData

@Model
class Recipe {
    @Attribute(.unique) var name: String
    var summary: String?
    var ingredients: [Ingredient]
}
```

2 Querying Data in SwiftUI

```
@Query var recipes: [Recipe]

var body: some View {
    List(recipes) { recipe in
        NavigationLink(recipe.name, destination:
RecipeView(recipe))
    }
}
```

SwiftData
Best Practices and Limitations

- Very new framework - watch for updates & potential bugs in edge cases
- Not as feature-rich or complex as Core Data for managing relationships between data

Coding time!

Link Coming