**CORE DATA WITH SWIFTUI**

**1. Offline Notes App**

**Scenario:**

A notes app needs to store user-created notes locally and sync them when online. The app supports **basic CRUD operations** (Create, Read, Update, Delete) but does not use Combine.

**Implementation:**

* **Core Data Entity:** Note
  + title: String
  + content: String
  + dateCreated: Date
* **Solution:**
  + Use @FetchRequest in SwiftUI to dynamically update UI when data changes.
  + Save notes locally and allow editing.

**2. Task Management App**

**Scenario:**

A task management app allows users to create tasks with due dates and categories. It must persist tasks even if the app is closed.

**Implementation:**

* **Core Data Entity:** Task
  + title: String
  + isCompleted: Bool
  + dueDate: Date
  + category: String(optional)
* **Solution:**
  + Use @FetchRequest to retrieve tasks dynamically.
  + Add filtering options (e.g., "Show Only Completed").
  + Persist user data without needing an internet connection.

**3. Recipe Storage App**

**Scenario:**

A recipe app needs to store **ingredients and cooking steps** for offline access.

**Implementation:**

* **Core Data Entities:**
  1. Recipe
     + title: String
     + category: String
  2. Ingredient
     + name: String
     + quantity: String
     + **Relationship:** Many-to-One with Recipe (optional)
  3. Instruction
     + stepNumber: Int
     + description: String
     + **Relationship:** Many-to-One with Recipe
* **Solution:**
  1. Display recipes in a List.
  2. Store multiple ingredients per recipe.
  3. Store multiple instructions per recipe.

**4. Workout Tracker App**

**Scenario:**

A fitness app tracks workouts with sets and repetitions.

**Implementation:**

* **Core Data Entities:**
  1. Workout
     + name: String
     + date: Date
     + Relationships -> Set of exercises
  2. Exercise
     + name: String
     + sets: Int
     + reps: Int
     + **Relationship:** Many-to-One with Workout

### **One-to-Many Relationship (Workout → Exercises)**

In a fitness-related database, a **one-to-many (1:M) relationship** exists between **workouts** and **exercises** because:

* One **workout** consists of **many** exercises.
* Each **exercise** belongs to **only one** workout (in a simple model).

For example:

| **Workout (Table 1)** | **Exercise (Table 2)** |
| --- | --- |
| Workout ID: 1 (Full Body) | Exercise ID: 1 (Squat, Workout ID: 1) |
| Workout ID: 1 (Full Body) | Exercise ID: 2 (Bench Press, Workout ID: 1) |
| Workout ID: 1 (Full Body) | Exercise ID: 3 (Deadlift, Workout ID: 1) |

This means:

* A **workout** can have **many exercises**.
* An **exercise** belongs to **only one workout** (if structured this way).

**Solution:**

* 1. Allow users to create workouts and add exercises.
  2. Retrieve workouts using @FetchRequest.
  3. Delete exercises or workouts when needed.

**5. Book Collection App**

**Scenario:**

A personal library app tracks books read by a user, including details like the author, genre, and reading status.

**Implementation:**

* **Core Data Entity:** Book
  + title: String
  + author: String
  + genre: String
  + isRead: Bool
  + dateAdded: Date
* **Solution:**
  + List books with sorting by **dateAdded**.
  + Filter books by **reading status**.
  + Allow users to update their reading progress.

**Summary of Use Cases**

| **App** | **Entities** | **Key Features** |
| --- | --- | --- |
| Notes App | Note | Offline notes, CRUD |
| Task Manager | Task | Task tracking, completion status |
| Recipe App | Recipe, Ingredient | Recipe with Ingredient CRUD |
| Workout Tracker | Workout, Exercise | Workout, Exercise with CRUD |
| Book Collection | Book | Sorting, filtering |

These are **real-world Core Data use cases** .