# Seminář 7: Persistence dat

## Tvorba mobilních aplikací

Roman Vyjídáček

# Obsah semináře

- Asynchronní operace
- CoreData
- SwiftData
- GRDB
- UserDefaults & Keychain

# Asynchronní operace

# Asynchronní operace

**Asynchronní vs. synchronní:**

- Synchronní operace blokuje aktuální vlákno, dokud neskončí.
- Asynchronní operace umožňuje pokračovat v provádění ostatního kódu, zatímco operace běží na pozadí.

**Motivace pro asynchronní model:**

- Zabránit blokování hlavního vlákna uživatelského rozhraní (UI).
- Lépe využít výkon více jádrových procesorů.
- Zlepšit uživatelský zážitek (plynulejší ovládání aplikace).

# Hlavní přínosy asynchronního zpracování

- UI se “nezasekává” při dlouhotrvajících procesech (např. síťová komunikace, čtení/ukládání dat).
- Více úloh může běžet současně (v rámci různých vláken nebo procesů).
- Neplýtvají se prostředky aktivním čekáním.

# Potenciální problémy a výzvy

- Zatímco synchronní kód je lineární, asynchronní kód obvykle vyžaduje callbacky, completion handlery nebo jiné mechanismy.
- Může docházet k souběhu (race conditions). Nutnost používat zámky, semafory nebo jiné synchronizační mechanismy.
- Identifikace a ošetření chyb v asynchronním řetězci volání není vždy tak přímočará jako v synchronním kódu.
- Vyhledávání chyb a ladění je složitější, protože běh programu není striktně sekvenční.

# Vzory a přístupy k asynchronnímu programování

**Callbacky (Completion Handlers)**

- Tradiční způsob, kdy funkci předáte „callback“ – kód, který se spustí po dokončení asynchronní akce.
- Výhody: Jednoduchá implementace, široká podpora.
- Nevýhody: Callback hell (vnořování a nečitelný kód), potenciálně obtížnější error handling.

**Promises/Futures**

- Stavový objekt reprezentující výsledek budoucí (asynchronní) operace.
- Výhody: Lepší čitelnost než řetězení callbacků, oddělení logiky řízení toku od samotné funkce.
- Nevýhody: Stále se mohou řetězit, nutnost naučit se další abstrakci.

# Vzory a přístupy k asynchronnímu programování

**Asynchronní funkce (async/await)**

- Syntaktický cukr nad Promise/Future konceptem.
- Výhody: Kód vypadá „téměř“ jako synchronní, snadné čtení a psaní, přirozený error handling pomocí try/catch.
- Nevýhody: Vyžaduje podporu v jazyce (C#, JavaScript, Swift 5.5+ atd.).

**Reaktivní programování**

- Kód se chová jako reakce na události („streamy dat“).
- Výhody: Výborné pro zpracování kontinuálních datových proudů, snadné škálování a znovupoužitelnost.
- Nevýhody: Vyšší křivka učení, potřebné pochopení reaktivních konceptů (např. Observables, Publishers, Subscribers).

# Vlákna

- Vlákno (Thread) je základní prováděcí jednotka, ve které kód běží.

**Hlavní vlákno (Main/UI Thread):**

- Zajišťuje zpracování uživatelského rozhraní.
- Blokování tohoto vlákna = aplikace působí zamrzle.

**Přepínání vláken:**

- Pro asynchronní operace se často používá pozadí (background) vlákno.
- Po dokončení úlohy je potřeba vrátit se na hlavní vlákno pro aktualizaci UI.

# Grand Central Dispatch (GCD)

- GCD je nízkoúrovňový API poskytující fronty pro asynchronní (a paralelní) zpracování úloh.

- Při práci s GCD často narazíte na typy DispatchQueue, DispatchWorkItem, DispatchGroup atd.

# Grand Central Dispatch (GCD)

## Výhody

- Přímá kontrola nad tím, na jakou queue se úloha zařadí.
- Nízká režie a vysoký výkon.
- Vhodné pro jednoduché případy, kdy nepotřebujete robustní abstrakci.

## Nevýhody

- Kód se může snadno znepřehlednit, pokud používáte hodně callbacků a bloků.
- Neřeší vztahy mezi úkoly tak přehledně jako objekty typu Operation.

```
// Provést asynchronní úlohu na globální
   frontě
DispatchQueue.global(qos: .background).async {
    // Operace na pozadí
     let result = longRunningFunction()
    // Po dokončení aktualizovat UI na hlavní
       frontě
    DispatchQueue.main.async {
        updateUI(with: result)
    }
}
```

# Operation & OperationQueue

- Operation je třída, která zapouzdřuje úlohu (task) do objektu.
- OperationQueue je fronta, která spravuje provádění instancí Operation.
- Umožňuje definovat závislosti mezi Operation objekty, prioritizaci a také pokročilé funkce pro zrušení (cancel()), pauzu atd.

# Operation & OperationQueue

**Výhody**

- Větší modularita a opakovaná použitelnost – Operation je opakovatelný objekt.
- Snadná správa závislostí mezi úlohami (operationB.addDependency(operationA)).
- Možnost lépe pracovat s cancel a kontrolovat stav.

**Nevýhody**

- O něco složitější než GCD (musíte dělat subclass nebo používat blokové operace).
- V některých případech může být zbytečně robustní, pokud potřebujete vyřešit jen drobné asynchronní volání.

```
class MyOperation: Operation {
    override func main() {
        // Dlouhotrvající úloha
        if isCancelled { return }
        let result = longRunningFunction()

        if isCancelled { return }
        // Další zpracování...
    }
}

let operation1 = MyOperation()
let operation2 = MyOperation()

// operation2 se spustí až po operation1
operation2.addDependency(operation1)

let queue = OperationQueue()
queue.addOperations([operation1, operation2],
                    waitUntilFinished: false)
```

# Combine

- Combine je reaktivní framework od Apple.
- Pracuje s datovými toky (streamy) a používá typy jako Publisher, Subscriber a Operator.
- Vhodné pro deklarativní přístup, kdy definujete, jak se mají data propagovat a transformovat.

# Combine - Publisher

- „Zdroj“ dat, jenž asynchronně vydává (emise) hodnoty nebo chyby (errors).
- Příkladem publisheru může být například proud hodnot přicházející ze síťového volání, notifikace systémových změn, uživatelské akce a podobně.

**Základní role:**

  - Publisher definuje, jaký typ hodnot vydává (např. Int, Data, String…).
  - Může vydat různý počet hodnot (0, 1, mnoho) po určitou dobu.
  - Může také vydat chybu (Error) nebo dokončit bez chyby (.finished).

**Příklady v Combine:**

  - Just(42) – publisher, který vydá hodnotu 42 a pak se dokončí.
  - URLSession.shared.dataTaskPublisher(for:) – vydává data z HTTP požadavku (nebo chybu).
  - Timer.publish(every:tolerance:on:in:options:) – vydává časová upozornění v pravidelných intervalech.

- Publisher sám o sobě neprovádí práci, dokud nemá Subscriber. Jedná se o tzv. „lazy evaluation“ – až v okamžiku přihlášení se subscriberu (tzv. subscription) se publisher skutečně začne spouštět a vydávat hodnoty.

# Combine - Subscriber

- „Příjemce“ nebo „odběratel“ dat od publisheru. Subscriber rozhoduje, jak chce data zpracovávat (např. zobrazit je, uložit je, transformovat je nebo je předat dál).

**Základní role:**

- Přihlásí se k publisheru a tím zahájí proces publikování (odesílání) dat.
- V okamžiku, kdy publisher vydá hodnotu, subscriber o tom dostane notifikaci.
- Subscriber může dále rozhodnout, kolik dalších dat přijme (zpětna vazba tzv. demand).
- Může obdržet informaci o chybě nebo o dokončení publisheru.

**Běžné implementace Subscriber:**

- sink(receiveCompletion:receiveValue:) – jednoduchý subscriber, který předává hodnotu do closure a nakonec se zruší.
- assign(to: on:) – předá hodnoty do nějaké proměnné nebo property.

**Cancellable:**

- Když se subscriber přihlásí k publisheru, dostane tzv. subscription. V Combine se obvykle pracuje s protokolem Cancellable (např. AnyCancellable), který umožňuje subscription kdykoliv zrušit (cancel()).

# Combine - Operator

- Operator je prostředník, který transformuje, filtruje nebo jinak zpracovává proud dat mezi publisherem a subscriberem.
- Operátor vždy vstup: (Publisher) → výstup: (Publisher), takže je řetězitelný.

**Základní role:**

- Vezme data (nebo chyby) z předchozího publisheru.
- Provede transformaci (map, filter, reduce, debounce, combineLatest, merge, flatMap atd.).
- Výsledkem je nový publisher, který předává data dál.

**Příklady operátorů:**

- map: Transformuje každou vydanou hodnotu na jiný typ nebo jinou hodnotu.
- filter: Propouští pouze hodnoty, které splňují danou podmínku.

# Combine - Operator

**Příklady operátorů:**

- map: Transformuje každou vydanou hodnotu na jiný typ nebo jinou hodnotu.
- filter: Propouští pouze hodnoty, které splňují danou podmínku.
- flatMap: Rozbaluje vnořené publishery, často se používá pro síťová volání typu Publisher<Publisher<Value, Error>, Error>.
- debounce: Užitečné při zpracování častých událostí (např. text input), aby se odesílaly dál jen změny s určitým odstupem času.
- combineLatest, merge, zip: Kombinují víc publisherů dohromady, např. když sledovat stav dvou různých streamů a reagovat, když oba vydají novou hodnotu.

```
let cancellable = somePublisher
    .map { ... }
    .filter { ... }
    .flatMap { ... }
    .sink { completion in
        // dokončení nebo chyba
    } receiveValue: { value in
        // zpracování hodnot
    }
```

# Combine

## Výhody

- Snadné zpracování asynchronních proudů (např. síťová volání, uživatelské události, notifikace).
- Možnost skládat více zdrojů dat (publishes) pomocí operátorů (map, flatMap, filter, merge…).
- Dobře zapadá do SwiftUI (např. @StateObject, @Published).

## Nevýhody

- Vyšší křivka učení (koncept reaktivního programování je pro některé vývojáře nový).
- Pro menší, jednorázová asynchronní volání může být zbytečně komplexní.

```swift
import Combine

// Publisher, který vydá jedno číslo po 1 vteřině
let publisher = Future<Int, Never> { promise in
    DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
        promise(.success(42))
    }
}

let cancellable = publisher
    .map { $0 * 2 }
    .sink { value in
        // "Doručená hodnota: 84"
        print("Doručená hodnota:", value)
    }
```

# Swift Concurrency (async/await)

**Hlavní myšlenka**

- Od Swift 5.5 lze používat async/await a Task (Structured Concurrency).
- Cílem je zlepšit čitelnost a psaní asynchronního kódu, aby vypadal téměř synchronně.

**Výhody**

- Lineární styl zápisu: asynchronní volání působí přirozeně.
- Error handling pomocí try a catch.
- Lepší práce s cancel (Task podporuje zrušení).
- Navíc existují koncepce actorů pro bezpečné sdílení stavu.

**Nevýhody**

- Vyžaduje Swift 5.5+ a iOS 15+, macOS 12+ (nebo novější) pro plnou funkčnost (lze použít back-deployment, ale s limity).
- Nutnost naučit se nové koncepty: Task, AsyncSequence, AsyncStream, Actor atd.

# Swift Concurrency (async/await)

**Nevýhody**

- Vyžaduje Swift 5.5+ a iOS 15+, macOS 12+ (nebo novější) pro plnou funkčnost (lze použít back-deployment, ale s limity).
- Nutnost naučit se nové koncepty: Task, AsyncSequence, AsyncStream, Actor atd.

- fetchData() je asynchronní funkce, takže ji lze volat pouze s await.
- Kód Task { ... } se spustí asynchronně, ale čte se lineárně.
- await MainActor.run { … } spustí kód v hlavním vlákně

```swift
func fetchData() async throws -> Data {
    // Simulace asynchronního síťového volání
    return Data("Hello World".utf8)
}

func process() {
    Task {
        do {
            let data = try await fetchData()
            // Zpracování stažených dat
            await MainActor.run { /* Update UI */ }
        } catch {
            // Ošetření chyby
            print("Chyba při stahování dat:", error)
        }
    }
}
```

# CoreData

# Co je Core Data?

- Persistence Layer od Applu pro ukládání a správu modelových objektů.
- Poskytuje vysokou úroveň abstrakce nad ukládáním dat do různých typů úložišť (SQLite, Binary, In-Memory…).
- Umožňuje pracovat s „objekty“ a jejich vztahy místo s tabulkami a sloupci.

**Kdy a proč používat Core Data?**

- Když potřebujete spravovat komplexnější model s více entitami, vztahy a filtry.
- Vhodné pro offline režim aplikace – uživatel může pracovat i bez síťového připojení.
- Má vestavěnou cache a dokáže se postarat o průběžné ukládání změn.

# Základní pojmy a architektura

**NSManagedObjectModel (MOM)**

- Definuje schéma dat: entity, jejich atributy, vztahy.
- Obvykle vytvořeno pomocí .xcdatamodeld souboru v Xcode.

**NSPersistentStoreCoordinator (PSC)**

- Spravuje konkrétní úložiště, kam se data ukládají (např. SQLite).
- Koordinuje komunikaci mezi NSManagedObjectContext a fyzickým úložištěm.

# Základní pojmy a architektura

**NSManagedObjectContext (MOC)**

- „Pracovní prostor“ pro vaše data v paměti (buffer).
- Sleduje změny, vytváří či maže objekty, provádí dotazy (fetch) a save.
- Existují různé styly MOC (např. hlavní UI kontext, privátní zázemí pro background operace).

**NSManagedObject**

- Třída (dynamicky generovaná nebo vlastní subclassa) reprezentující 1 záznam (instanci entity).
- V sobě nese atributy (např. jméno, věk) a vztahy (např. mnoho ke mnoha).

# Nastavení Core Data Stacku

- Od iOS 10+ (a Xcode generuje i template) můžete použít NSPersistentContainer, který sestaví většinu stacku za vás.
- container.viewContext – hlavní kontext, obvykle vázaný na hlavní (UI) vlákno.
- Pro background operace používejte container.newBackgroundContext().

```
import CoreData

class PersistentContainer {
    static let shared = PersistentContainer()

    let container: NSPersistentContainer

    private init() {
        // Název .xcdatamodeld
        container = NSPersistentContainer(
                            name: “MyModelName"
                        )
        container.loadPersistentStores {
         storeDescription, error in
            if let error = error {
                fatalError("Nepodařilo se načíst úložiště:
                            \(error)")
            }
        }
    }

    var viewContext: NSManagedObjectContext {
        return container.viewContext
    }
}
```

# Práce s entitami - Vytvoření záznamu

- Definujte entitu v .xcdatamodeld. Příklad: Person se sloupci name (String) a age (Int).
- Při použití Class Definition se vygeneruje NSManagedObject subclass

```
// Vygenerováno Xcode
class Person: NSManagedObject {
    @NSManaged public var name: String?
    @NSManaged public var age: Int16
}

let context = PersistentContainer.shared.viewContext

// Vytvoření instance
let newPerson = Person(context: context)
newPerson.name = "Alice"
newPerson.age = 30

// Uložení do persistentního úložiště
do {
    try context.save()
} catch {
    print("Chyba při ukládání: \(error)")
}
```

# Sort Descriptor

- NSSortDescriptor je objekt, který určuje způsob řazení výsledků při načítání dat z databáze.
- Při vytváření NSSortDescriptor(key:ascending:) určíte:
  - key: název atributu, podle kterého se třídí (např. "name", "age").
  - ascending: true pro vzestupné řazení, false pro sestupné.
- Do NSFetchRequest můžete přidat více NSSortDescriptor objektů, abyste určili více úrovní řazení (např. nejdříve podle jména a poté podle data).

```swift
// Seřadí záznamy podle 'name' vzestupně
let sortByName = NSSortDescriptor(
    key: "name", ascending: true
)
fetchRequest.sortDescriptors = [sortByName]
```

# Predicate

- NSPredicate je logický výraz (filtr), který určuje podmínku pro výběr (filtraci) záznamů z databáze.
- Predikáty používají formátovací řetězce typu "atribut == hodnotě", "atribut > hodnotě" atp.
- Mohou obsahovat logické operátory (AND, OR, NOT), porovnání řetězců (např. CONTAINS, BEGINSWITH), porovnání čísel, datumů apod.
- V NSFetchRequest jej nastavíte do vlastnosti predicate.

```
// Seřadí záznamy podle 'name' vzestupně
let sortByName = NSSortDescriptor(
    key: "name", ascending: true
)
fetchRequest.sortDescriptors = [sortByName]


// Vyfiltruje všechny záznamy,
// kde 'age' je větší než 20
fetchRequest.predicate = NSPredicate(
    format: "age > %d", 20
)
```

# Práce s entitami - Načítání (Fetch)

```swift
// 1) Vytvoření dotazu na entitu 'Person'
let fetchRequest: NSFetchRequest<Person> = Person.fetchRequest()

// 2) Definice podmínky (predikátu):
//    Chceme pouze záznamy, kde age > 20
fetchRequest.predicate = NSPredicate(format: "age > %d", 20)

// 3) Definice třídění:
//    Seřadit záznamy podle jména (name) vzestupně
fetchRequest.sortDescriptors = [NSSortDescriptor(key: "name", ascending: true)]

// 4) Provedení dotazu (fetch)
do {
    let results = try context.fetch(fetchRequest)

    // 5) Iterace přes získaná data
    results.forEach { person in
        print("Name: \(person.name ?? ""), Age: \(person.age)")
    }
} catch {
    print("Chyba při načítání: \(error)")
}
```

# SwiftData

- Framework pro ukládání dat do perzistence, představený Apple v roce 2023.
- Navazuje na myšlenku Core Data, avšak nabízí moderní Swift syntaxi a těsnější propojení se SwiftUI.
- Kladně řeší mnoho komplikací z Core Data (méně „boilerplate“ kódu, snazší modelování, lepší spolupráce s reaktivním UI).

**Hlavní rysy**

  - @Model anotace pro deklaraci modelů.
  - ModelContext a ModelContainer – místo NSManagedObjectContext a NSPersistentContainer.
  - Automatická integrace s SwiftUI přes property wrappers typu @Environment(\.modelContext) nebo @Query.

# Základní pojmy a architektura

**@Model**

- Přidáváte na struct (či class), který má reprezentovat vaši entitu.
- SwiftData se stará o to, aby byl tento typ schopen být uložen do úložiště.
- V kódu tak používáte přirozené Swift konstrukce (bez nutnosti @NSManaged).

**ModelContext**

- Podobně jako NSManagedObjectContext – je to prostor, ve kterém pracujete s daty (vkládáte, mažete, upravujete).
- Udržuje informace o změnách a stará se o uložení do persistence.

# Základní pojmy a architektura

## ModelContainer

- Nahrazuje roli NSPersistentContainer.
- Zastřešuje konfiguraci (typ perzistentního úložiště, cesty, migrace) a vytváří ModelContext.

## Pozorování změn

- SwiftData automaticky sleduje změny ve vašich @Model objektech a aktualizuje SwiftUI.
- Podobně jako @Published nebo @State, ale tentokrát přímo propojené s databází.

# Nastavení a inicializace SwiftData

- modelContainer(for: [Person.self]) říká, že budeme používat SwiftData s entitou Person.
- SwiftData implicitně vytvoří úložiště (např. SQLite) v rámci aplikace, ale lze konfigurovat i ručně (např. pro testování in-memory).

```
import SwiftUI
import SwiftData

@main
struct MySwiftDataApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
                // Připojíme ModelContainer
                // pomocí SwiftUI modifikátoru
                .modelContainer(for: [Person.self])
        }
    }
}
```

# Definice modelu pomocí @Model

- Označení @Model zajišťuje, že SwiftData rozpozná tuto třídu jako entitu, kterou lze uložit.
- Vlastnosti name a age se automaticky stanou odpovídajícími sloupci v databázi.
- SwiftData dokáže třídu pozorovat, sledovat změny a propisovat do databáze.

**Vlastní identifikátory**

- Implicitně SwiftData přidává vlastní ID (např. id typu UUID). Můžete jej definovat i sami.
- SwiftData se postará o generování či sledování klíčů.

```swift
import SwiftData

@Model
class Person {
    var name: String
    var age: Int

    // SwiftData vyžaduje aspoň jeden init
    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}
```

# Upozornění

**Následující slidy ukazují použití SwiftData, přímo podle návrhu Apple. Tato architektura však nezapadá do námi zvolené.**

https://medium.com/@dikidwid0/implement-swiftdata-in-swiftui-using-mvvm-architecture-pattern-aa3a9973c87c

# Vkládání

- @Environment(\.modelContext) nám injektuje ModelContext, podobně jako @Environment(\.managedObjectContext) u Core Data.
- context.insert(...) vytvoří záznam v kontextu, context.save() zapíše data fyzicky do úložiště.

```swift
import SwiftUI
import SwiftData

struct ContentView: View {
    // Získáme kontext z Environmentu
    @Environment(\.modelContext) private var context

    var body: some View {
        Button("Přidat osobu") {
            let newPerson = Person(name: "Alice", age: 25)
            // Vložíme osobu do kontextu
            context.insert(newPerson)

            // Uložíme do persistence
            do {
                try context.save()
            } catch {
                print("Chyba při ukládání: \(error)")
            }
        }
    }
}
```

# Query

```
// @Query Property Wrapper
struct PeopleView: View {
    // Jednoduchá query: Všechny osoby,
    // seřazené podle jména vzestupně
    @Query(sort: \.name, order: .forward)
    private var people: [Person]

    var body: some View {
        List(people) { person in
            Text("\(person.name), \(person.age)")
        }
    }
}

// Pokročilejší filtrování
@Query(filter: #Predicate<Person> { $0.age > 20 },
       sort: \.name, order: .forward)
private var adults: [Person]

// Mazání
do {
    context.delete(somePerson)
    try context.save()
} catch {
    print("Chyba při mazání: \(error)")
}
```

# Vztahy mezi modely

- Author.books je pole typu [Book], SwiftData automaticky rozpozná vztah.
- Book.author je reference Author?.
- SwiftData se postará o uložení obou entit a jejich vzájemného vztahu.

```
@Model
class Author {
    var name: String
    var books: [Book] = []

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

@Model
class Book {
    var title: String
    var author: Author?

    init(title: String, author: Author? = nil) {
        self.title = title
        self.author = author
    }
}

let jk = Author(name: "J.K. Rowling")
let hp = Book(title: "Harry Potter", author: jk)

context.insert(jk)
context.insert(hp)
try context.save()
```

# Migrace a verze modelu

- Aktuálně (v prvních verzích SwiftData) není k dispozici plnohodnotná migrace jako v Core Data (kdy definujete explicitní verze schématu).
- SwiftData se snaží migrovat automaticky, pokud přidáte/odeberete vlastnosti.
- Pro rozsáhlé změny datového modelu (zvláště v produkční aplikaci) bude nutné sledovat, jak Apple SwiftData dále rozvíjí. Možná se dočkáme robustnějšího API pro řízení migrací v budoucnu.

# Concurrency

**Asynchronní operace:**

- Stejně jako u SwiftUI a Swift Concurrency – model kontext běží na různých vláknech dle potřeby.

**Současné přístupy:**

- SwiftData při použití @Environment(\.modelContext) v UI automaticky zajišťuje, že úpravy z jiných vláken se správně projeví.

- Doporučení Applu zní: „Změny, které ovlivňují UI, dělejte v hlavním kontextu. “ Pro rozsáhlé background zpracování se zatím doporučuje, abyste měli samostatný ModelContext.

# Výhody a nevýhody oproti Core Data

**Výhody**

- Jednodušší API: Méně boilerplate, deklarativní práce s daty (podobná SwiftUI mentalitě).
- Těsné propojení s SwiftUI: @Query zajišťuje automatické načítání a reaktivní aktualizace UI.
- Moderní Swift: @Model, property wrappers, predikáty v Swift syntaxi.

**Nevýhody / Omezení**

- Méně zralé: Jde o nový framework, který se bude pravděpodobně rychle vyvíjet a měnit.
- Migrace: Neexistuje tak robustní řešení jako v Core Data (Lightweight/Custom Migration).
- Podpora starších verzí iOS: SwiftData je k dispozici až od iOS 17, macOS 14, watchOS 10… Nelze jednoduše back-deploy.
- Pokročilá kontrola nad SQL a výkonem: Pokud potřebujete přesně řídit indexy, exekuční plány atd., SwiftData je momentálně spíše high-level (a oficiální dokumentace to zatím moc neřeší).

# GRDB

**SQLite wrapper pro Swift:**

- Zjednodušuje komunikaci se SQLite, ať už používáte SQL příkazy přímo, nebo pracujete s vyššími abstrakcemi (typově bezpečné rozhraní, protokol FetchableRecord, PersistableRecord atd.).

**Lehká a rychlá knihovna:**

- Větší kontrola nad databází, menší „magie“ ve srovnání s Core Data.
- Vhodné pro aplikace, které potřebují explicitnější model ukládání, bez overheadu objektové nadstavby.

**Podpora pro concurrency:**

- Využívá databázové fronty (DatabaseQueue, DatabasePool) a transakce pro bezpečnou práci z více vláken.

# Nastavení a inicializace

- Nejprve je potřeba instalovat GRDB (např. pomocí Swift Package Manager).
- DatabaseQueue udržuje sériový přístup k databázi. Pokud potřebujete současné čtení z více vláken, použijte DatabasePool.

```
import GRDB

class DatabaseManager {
    static let shared = DatabaseManager()
    let dbQueue: DatabaseQueue

    private init() {
        do {
            // Vytvoření (nebo otevření) souboru SQLite
            // (např. v Document directory)
            let dbURL = try FileManager.default
                .url(for: .documentDirectory,
                     in: .userDomainMask,
                     appropriateFor: nil,
                     create: true
                )
                .appendingPathComponent("mydatabase.sqlite")

            // Inicializace DatabaseQueue
            dbQueue = try DatabaseQueue(path: dbURL.path)

            // Můžeme provést migrace (popsáno níže)
            try setupMigrations(on: dbQueue)

        } catch {
            fatalError("Chyba při vytváření DB: \(error)")
        }
    }
}
```

# Definice modelu (Record)

- Codable usnadňuje mapping JSON ↔ Person (pokud potřebujete).
- GRDB automaticky dovodí sloupce z klíčů, pokud jsou stejné jako property.
- Pokud potřebujete něco speciálního, implementujte metody
  - encode(to:) / init(from:)
  - encode(to:in:) / init(row:)

```
struct Person: FetchableRecord, PersistableRecord, Codable {
    var id: Int64?
    var name: String
    var age: Int

    // Název tabulky (pokud se neliší od názvu structu)
    static var databaseTableName: String {
        return "person"
    }

    // Primární klíč, abychom při update/delete věděli, podle
čeho se řídí
    static let databasePrimaryKey = "id"
}
```

# Použití

```
// Insert
try DatabaseManager.shared.dbQueue.write { db in
    var newPerson = Person(id: nil, name: "Alice", age: 30)
    try newPerson.insert(db)
    // teď newPerson.id bude obsahovat autoincremented ID
}

// Fetch (Select)
let people = try DatabaseManager.shared.dbQueue.read { db in
    try Person.fetchAll(db) // SELECT * FROM person
}

// Podmínky (WHERE)
let adults = try DatabaseManager.shared.dbQueue.read { db in
    try Person
        .filter(Column("age") > 20)
        .order(Column("name").asc)
        .fetchAll(db)
}
```

# Použití

```
// Update
try DatabaseManager.shared.dbQueue.write { db in
    var alice = try Person
        .filter(Column("name") == "Alice")
        .fetchOne(db)!

    alice.age = 31
    try alice.update(db)  // UPDATE person SET age = 31 WHERE id = ...
}

// Delete
try DatabaseManager.shared.dbQueue.write { db in
    if let oldPerson = try Person.fetchOne(db, key: 1) {
        try oldPerson.delete(db) // DELETE FROM person WHERE id = 1
    }
}
```

# Migrace

- Umožňuje verzovat schéma a postupně jej aktualizovat (přidávat tabulky, měnit jejich strukturu atp.).
- Přidáváte pojmenované migrace, které popíšete v closure.
- Při prvním spuštění aplikace se provede veškeré nové migrace, a poté se uloží jejich jména do databázové tabulky grdb_migrations.

```swift
var migrator = DatabaseMigrator()

migrator.registerMigration("createPerson") { db in
    try db.create(table: "person") { t in
        t.autoIncrementedPrimaryKey("id")
        t.column("name", .text).notNull()
        t.column("age", .integer)
    }
}

migrator.registerMigration("addEmailToPerson") { db in
    try db.alter(table: "person") { t in
        t.add(column: "email", .text)
    }
}

// Při startu aplikace
try migrator.migrate(dbQueue)
```

# Možnosti persistence

- Core Data / SwiftData – pokud vám vyhovuje nativní Apple technologie a integrace se SwiftUI (SwiftData je zatím jen iOS 17+).
- SQLite nebo jeho Swifty wrappers (FMDB, SQLite.swift, GRDB) – pokud chcete detailní kontrolu nad SQL, vyšší výkon, či menší „magii“ okolo.
- Realm – pokud preferujete NoSQL přístup, rychlou integraci a potřebujete snadné sdílení dat mezi platformami (včetně synchronizačního řešení).
- WCDB, YapDatabase – alternativní knihovny nad SQLite s určitými specialitami (výkon, synchronizace, indexy).
- V praxi se nejčastěji setkáte s Core Data (včetně nového SwiftData), dále pak s Realm a některým z wrapperů pro SQLite (zejména GRDB).

# UserDefaults a Keychain

# UserDefaults

- Jednoduché úložiště pro uchování malého množství dat mezi spuštěními aplikace.

**Typické použití:**

- uchování uživatelských preferencí (tmavý režim, jazyk),
- poslední stav přepínačů, skóre, pozice,
- nastavení (např. zapamatovat e-mail).
- Data se ukládají jako klíč–hodnota (String, Bool, Int, Double, URL, Data, Array, Dictionary).

# UserDefaults

**Výhody:**

- jednoduché použití,
- přístupné z kteréhokoliv místa v aplikaci,
- přetrvávají mezi spuštěními.

**Nevýhody:**

- není určeno pro velké objemy nebo citlivá data,
- není šifrováno.

# UserDefaults

- Hodnoty se ukládají pomocí set(_:forKey:).
- Načítají se pomocí metod jako bool(forKey:), string(forKey:), atd.
- Doporučuje se definovat klíče jako konstanty.

```swift
let storage = UserDefaults.standard

// Uložení hodnot
storage.set(true, forKey: "isDarkMode")
storage.set("Pepa", forKey: "username")

// Načtení hodnot
let darkMode = storage.bool(forKey: "isDarkMode")
let username = storage.string(forKey: "username")
```

# Keychain

- Keychain je systémové šifrované úložiště určené pro uchování citlivých dat:
  - přihlašovací údaje,
  - tokeny,
  - přístupové klíče apod.
- Data uložená v Keychainu:
  - přežívají restart aplikace i zařízení,
  - jsou šifrována a chráněna systémem,
  - mohou být sdílena mezi aplikacemi (se správným nastavením).
- Keychain není přímo součástí SwiftUI – používá se přes Keychain Services API nebo knihovny (např. KeychainAccess(https://github.com/kishikawakatsumi/KeychainAccess)).

# Knihovna KeychainAccess

- Vhodné pro bezpečné uchování dat, která nemají být čitelná v UserDefaults.
- Pokud není knihovna povolena, lze pracovat s Keychain Services přes SecItemAdd, SecItemCopyMatching, apod. (ale je to mnohem složitější).
- Před použitím ve skutečné aplikaci doporučeno prostudovat zabezpečení a chování na různých verzích iOS.

```
import KeychainAccess

let keychain = Keychain(service: "cz.upol.app")

// Uložení
try? keychain.set("tajneHeslo123",
                  key: "userPassword")

// Načtení
let password = try? keychain.get("userPassword")
```