Seminář 3: Navigace a datové modely

Tvorba mobilních aplikací

Roman Vyjídáček

Obsah semináře

- Architektura aplikace
- Navigace
- UserDefaults a Keychain

Architektura aplikace

Softwarová architektura

- Softwarová architektura je struktura systému, která definuje způsob organizace kódu, komunikaci mezi částmi aplikace a rozdělení odpovědností

Proč je architektura klíčová?

- Udržovatelnost – Snadnější oprava chyb a přidávání funkcí.
- Testovatelnost – Možnost snadného testování jednotlivých částí aplikace.
- Škálovatelnost – Možnost rozšiřování aplikace bez velkých zásahů do kódu.
- Znovupoužitelnost – Kód lze snadno využít v jiných částech projektu.
- Minimalizace závislostí – Snížení propojení mezi částmi kódu, což usnadňuje vývoj.

Architektonické vzory v iOS

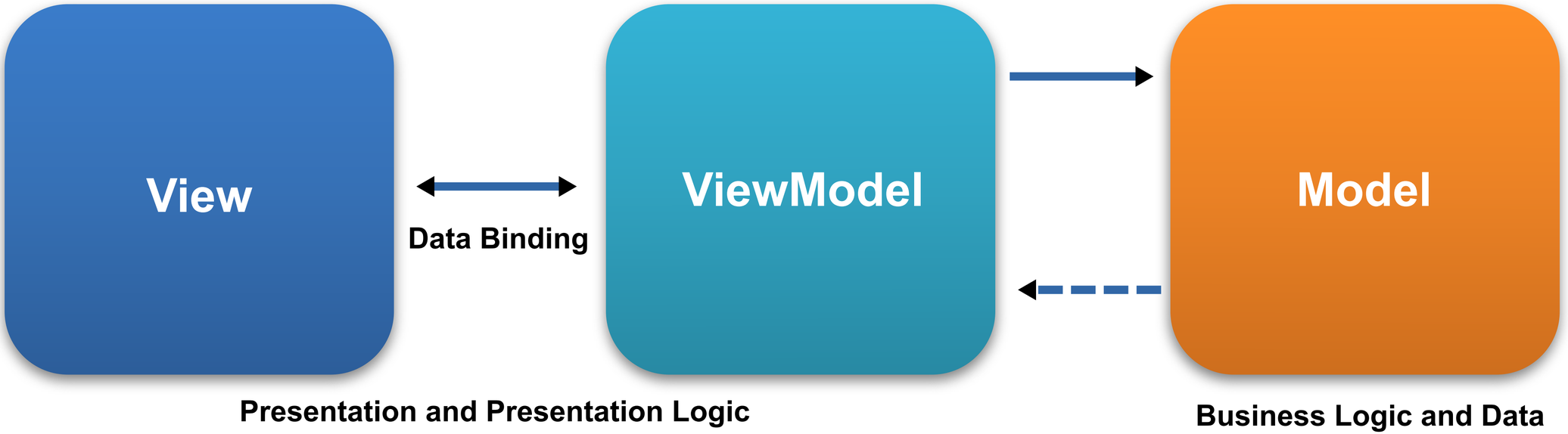
V iOS vývoji existuje několik oblíbených architektonických přístupů:

Architektura	Popis
MVC (Model-View-Controller)	Tradiční Apple přístup, ale často vede k Massive View Controller
MVVM (Model-View-ViewModel)	Odstraňuje problémy MVC – odděluje UI od logiky pomocí ViewModel
VIPER	Striktně odděluje vrstvy, ideální pro velké aplikace, ale složitější na implementaci
Clean Architecture	Rozděluje kód na Presentation, Domain, Data, což minimalizuje závislosti

V moderním vývoji iOS aplikací se nejčastěji používá MVVM + Clean Architecture

Architektury iOS aplikace (MVVM + Clean Architecture)

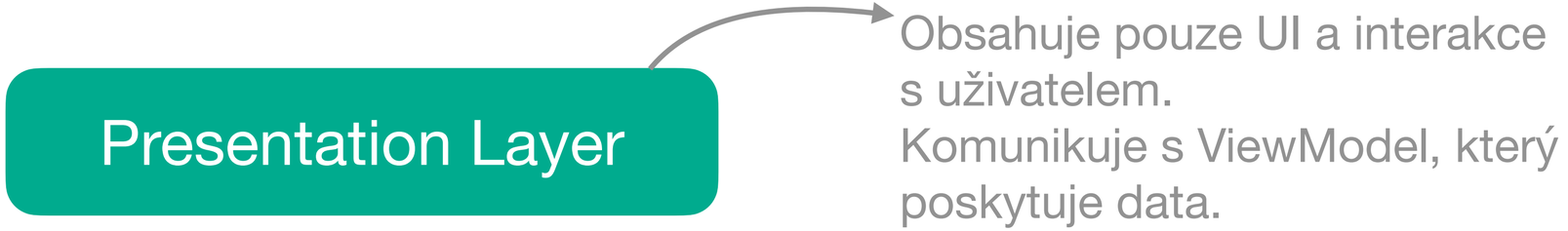
Tento přístup odděluje aplikační vrstvy, čímž zajišťuje modularitu a lepší správu kódu.



https://en.wikipedia.org/wiki/Model–view–viewmodel

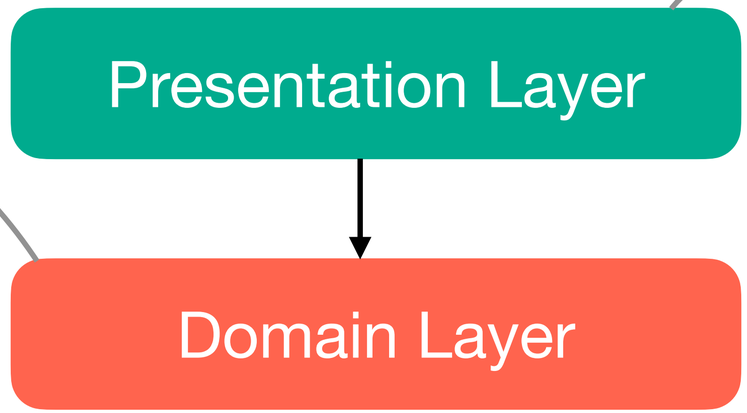
Architektury iOS aplikace (MVVM + Clean Architecture)

Architektury iOS aplikace (MVVM + Clean Architecture)



Architektury iOS aplikace (MVVM + Clean Architecture)

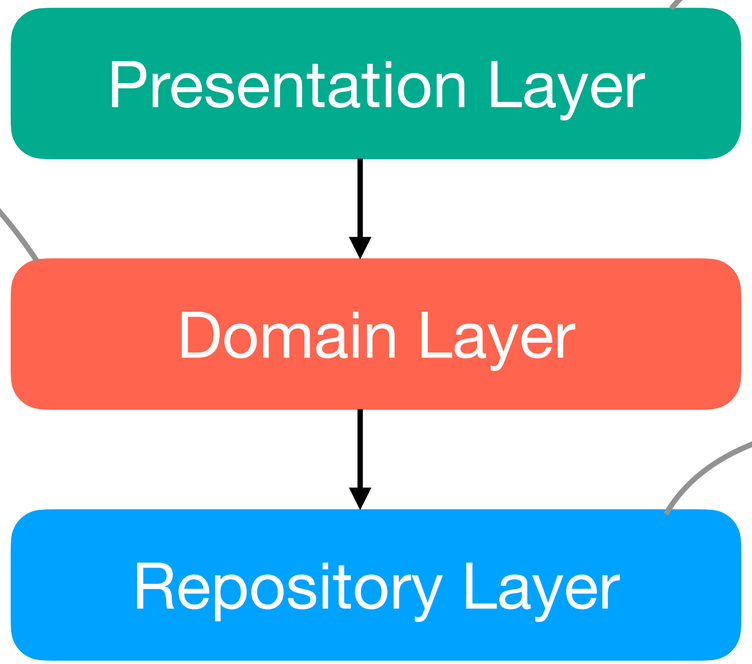
Obsahuje Use Cases a View Models, které obsahují aplikační logiku.
Nezávisí na žádné konkrétní implementaci databáze nebo API.



Obsahuje pouze UI a interakce s uživatelem.
Komunikuje s ViewModel, který poskytuje data.

Architektury iOS aplikace (MVVM + Clean Architecture)

Obsahuje Use Cases a View Models, které obsahují aplikační logiku.
Nezávisí na žádné konkrétní implementaci databáze nebo API.

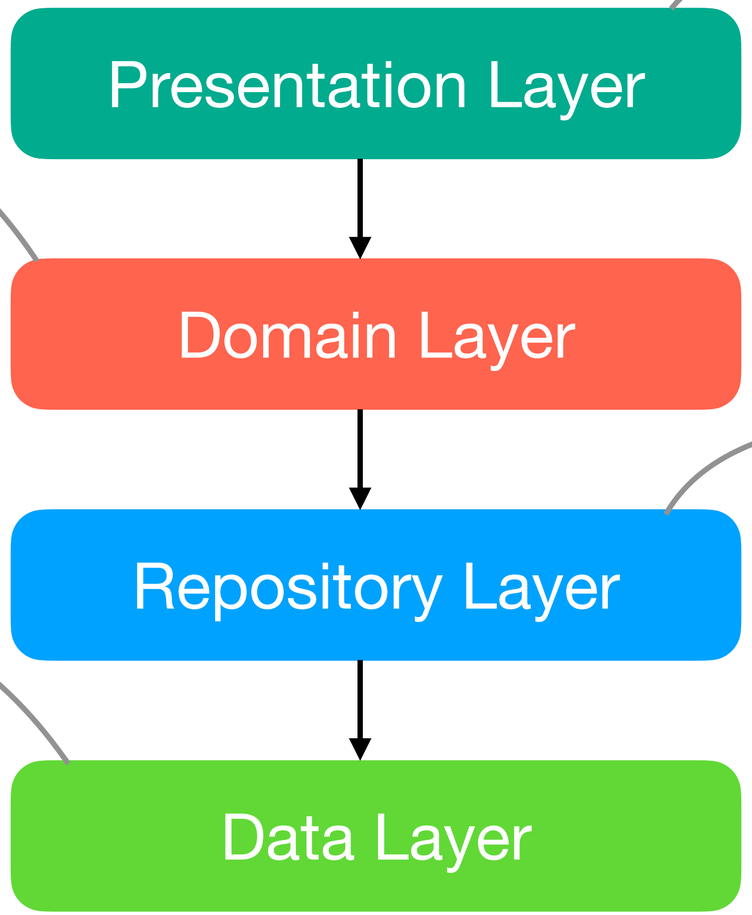


Obsahuje pouze UI a interakce s uživatelem.
Komunikuje s ViewModel, který poskytuje data.

Poskytuje Domain vrstvě abstrakci nad daty.
Může získávat data z různých zdrojů (API, databáze).

Architektury iOS aplikace (MVVM + Clean Architecture)

Obsahuje Use Cases a View Models, které obsahují aplikační logiku.
Nezávisí na žádné konkrétní implementaci databáze nebo API.



Obsahuje pouze UI a interakce s uživatelem.
Komunikuje s ViewModel, který poskytuje data.

Poskytuje Domain vrstvě abstrakci nad daty.
Může získávat data z různých zdrojů (API, databáze).

Obsahuje API volání, databázi (CoreData, Realm), práci se soubory.
Díky této vrstvě lze snadno změnit zdroj dat bez ovlivnění aplikace

State

- ContentViewState umožňuje sledování změn ve SwiftUI pomocí @Published.
- Vlastnost isLoading označuje, zda probíhá načítání, a při její změně se automaticky aktualizuje navázané view.

```
final class ContentViewState: ObservableObject {

    @Published var isLoading = false
}
```

View

- Podobně jako v architekturách MVC a MVP, „view“ představuje strukturu, rozvržení a vzhled toho, co uživatel vidí na obrazovce.
- Zobrazuje reprezentaci modelu (dat) a reaguje na interakce uživatele (kliknutí, psaní, gesta atd.).
- Interakce nepředává přímo modelu, ale posílá je „view modelu“ přes datové vazby.
- Datová vazba (properties, callbacky) propojuje „view“ a „view model“ a umožňuje přenos dat i událostí.

```
struct ContentView: View {

    @ObservedObject
    private var state: ContentViewState
    private let viewModel: ContentViewModel

    var body: some View {
        if state.isLoading {
            ProgressView()
                .progressViewStyle(.circular)
        } else {
            Button("Load Data") {
                viewModel.loadData()
            }
        }
    }

    static func build() -> ContentView {
        let viewModel = ContentViewModel()
        return ContentView(state: viewModel.state,
                           viewModel: viewModel)
    }
}
```

ViewModel

- ViewModel je prostředník mezi modelem a SwiftUI pohledem – vystavuje veřejné vlastnosti a metody, které může view používat.
- Na rozdíl od MVC nebo MVP neobsahuje ViewModel odkaz na view – ve SwiftUI se view přímo váže na data pomocí @StateObject, @ObservedObject apod.
- SwiftUI používá datové vazby (binding), které automaticky zajišťují přenos dat mezi ViewModelem a View.
- ViewModel často reprezentuje stav dat a může být chápán jako datový objekt (Data Transfer Object).

```
final class ContentViewModel {

    let state = ContentViewState()

    func loadData() {
        state.isLoading = true
    }
}
```

Doménový model

- Reprezentace klíčových entit a pravidel specifických pro oblast (doménu), kterou aplikace řeší.
- Oddělení obchodní logiky od uživatelského rozhraní a datové vrstvy.

Klíčové vlastnosti Doménového modelu

- Nezávislost: Nemá závislosti na UI (SwiftUI) ani na konkrétní databázové implementaci.
- Čistota: Zaměřuje se na obchodní logiku, validaci a pravidla domény.
- Testovatelnost: Snadno testovatelný díky absenci závislostí na dalších vrstvách aplikace.

Integrace Doménového modelu ve SwiftUI

- ViewModel: Spojuje Doménový model s uživatelským rozhraním, transformuje data pro zobrazení.
- SwiftUI View: Pozoruje ViewModel a reaguje na změny dat pomocí datových vazeb.

Usecase

- Zaměřuje se na jednu konkrétní akci nebo cíl uživatele
- Kombinací menších Use Case lze vytvářet složitější funkce systému
- V naší architektuře je Usecase umístěn mezi ViewModel a Repository
- Má pouze jednu veřejnou metodu use(…)

```
final class GetCurrentLocationUsecase {

    func use() -> CLLocation {
        // Zde ziskame aktualni pozici
        // napriklad z GPSManager
    }
}

class ViewModel {

    private let getCurrentLocationUsecase:
GetCurrentLocationUsecase

    func loadData() {
        let location =
getCurrentLocationUsecase.use()
    }
}
```

Repository

- Repository Pattern poskytuje abstraktní vrstvu mezi obchodní logikou aplikace a datovými zdroji, jako jsou databáze či API
- Oddělení obav (Separation of Concerns): Umožňuje, aby doménová vrstva nebyla závislá na konkrétní implementaci datového zdroje
- Flexibilita: Usnadňuje změnu nebo kombinaci různých datových zdrojů bez nutnosti úprav v obchodní logice
- Testovatelnost: Umožňuje snadné vytváření mock implementací pro jednotkové testování.

```
protocol UserRepository {

    func fetchUsers() async throws -> [User]
    func addUser(_ user: User) async throws
}

class UserRepositoryImpl: UserRepository {

    func fetchUsers() async throws -> [User] {
        // Načtení uživatelů z UserDefaults
    }
    func addUser(_ user: User) async throws {
        // Uložení uživatele do UserDefaults
    }
}

class UserRepositoryMock: UserRepository {
    func fetchUsers() async throws -> [User] {
        return User.mockArray
    }

    func addUser(_ user: User) async throws {
        // zde potřebujeme Mock
        // implementaci DBManager
    }
}
```

Manager

- Třída, která se stará o konkrétní funkčnost, např. síťovou komunikaci, ukládání do souborů, notifikace, GPS apod.
- Zapouzdřuje logiku, která nesouvisí přímo s UI ani doménovou logikou.

Role Managera

- Oddělení odpovědností:
  - Zajišťuje, že ViewModel není zahlcen technickými detaily (např. voláním URLSession, prací s diskem).
- Opakovatelnost:
  - Lze ho znovu použít napříč různými ViewModely nebo i aplikacemi.
- Snadnější testování:
  - Může být nahrazen mockem při testování ViewModelu.

Navigace

Navigace

- Navigace určuje, jak se uživatel pohybuje mezi obrazovkami.
- Ve SwiftUI se používá komponenta NavigationStack (od iOS 16), dříve NavigationView.
- Navigace může být spouštěna uživatelem (např. pomocí NavigationLink) nebo programově (změnou cesty).
- Cílem navigace je zobrazení nové obrazovky nebo komponenty v reakci na akci uživatele.

Typy navigace

- Hierarchická (drill-down): Uživatel postupuje skrze vrstvy obsahu, například ze seznamu položek k detailu konkrétní položky.
- Laterální (tab-based): Přepínání mezi nezávislými sekcemi aplikace pomocí záložek.

Základní NavigationStack

- Komponenta zavedená v iOS 16, která nahrazuje NavigationView pro správu hierarchické navigace.
- NavigationStack obaluje obsah a spravuje navigační cestu.
- NavigationLink naviguje na jinou obrazovku s danou hodnotou.

Hlavní vlastnosti

- Lepší správa navigačního stavu.
- Podpora pro programatické řízení navigace.

```
NavigationStack {
    NavigationLink("Tap Me") {
        Text("Detail View")
    }
}
```

Problém s inline NavigationLink

- NavigationLink(destination:) okamžitě vytváří cílové View.
- Může to zpomalit aplikaci, pokud se destinace vytváří ve smyčce (např. List).
- Správnější přístup: použít NavigationLink(value:) + navigationDestination().
- Cílové View se vytváří až v momentě, kdy se na něj naviguje.
- Zlepšuje to výkon a řeší problém se "zbytečnými" Views.

```
NavigationStack {
    List(0..<5) { i in
        NavigationLink("Číslo \(i)", value: i)
    }
    .navigationDestination(for: Int.self)
    { value in
        Text("Zvolili jste \(value)")
    }
}
```

Programatická navigace

- Pomocí @State nebo @Observable proměnné můžeme manipulovat s navigační cestou.
- Tím lze např. přeskočit do detailu bez uživatelské interakce.
- Vhodné pro reakce na logiku (např. po úspěšném uživatelském přihlášení).
- Proměnná path je pole hodnot, které reprezentují historii navigace.
- Každý append() v poli provede navigaci.

```
@State private var path = [Int]()

NavigationStack(path: $path) {
    VStack {
        Button("Zobraz 1") { path.append(1) }
        Button("Zobraz 2") { path.append(2) }
    }
    .navigationDestination(for: Int.self)
    { value in
        Text("Detail \(value)")
    }
}
```

Navigace s různými datovými typy

- Pomocí NavigationPath můžeme uchovávat více různých typů v cestě.
- Např. Int, String, nebo vlastní modely (které implementují Hashable).
- Každý typ musí mít definovaný navigationDestination.
- Vhodné např. pro aplikace, kde typ navigace závisí na datech.

```
@State private var path = NavigationPath()

NavigationStack(path: $path) {
    List {
        Button("Zvol číslo") { path.append(42) }
        Button("Zvol text") { path.append("Swift") }
    }
    .navigationDestination(for: Int.self) { value in
        Text("Zvolené číslo: \(value)")
    }
    .navigationDestination(for: String.self) { value in
        Text("Zvolený text: \(value)")
    }
}
```

Modalní prezentace

Co je modalní prezentace?

- Zobrazení nové obrazovky na vrchu stávajícího obsahu, často pro získání uživatelského vstupu nebo zobrazení doplňujících informací.

Typy modalních prezentací

- Sheet: Obrazovka se vysune ze spodní části displeje a částečně překryje původní obsah.
- Full-Screen Cover: Nová obrazovka překryje celý displej.

Sheet

- Pokrývá část obrazovky.
- Uživatel může zavřít tažením dolů.
- Pro doplňkové úkoly nebo informace, které nevyžadují plnou pozornost uživatele.

```
struct ContentView: View {
    @State private var isSheetPresented = false

    var body: some View {
        Button("Zobrazit sheet") {
            isSheetPresented.toggle()
        }
        .sheet(isPresented: $isSheetPresented) {
            SheetView()
        }
    }
}

struct SheetView: View {
    @Environment(\.dismiss) var dismiss

    var body: some View {
        VStack {
            Text("Toto je sheet")
            Button("Zavřít") {
                dismiss()
            }
        }
        .padding()
    }
}
```

Full-Screen Cover

- Pokrývá celou obrazovku.
- Vyžaduje explicitní akci pro zavření (např. tlačítko “Zavřít”).
- Pro úkoly vyžadující plné soustředění, jako je přehrávání videa nebo vyplnění formuláře.

```
struct ContentView: View {
    @State private var isFullScreenPresented = false

    var body: some View {
        Button("Zobrazit na celé obrazovce") {
            isFullScreenPresented.toggle()
        }
        .fullScreenCover(isPresented:
$isFullScreenPresented) {
            FullScreenView()
        }
    }
}

struct FullScreenView: View {
    @Environment(\.dismiss) var dismiss

    var body: some View {
        VStack {
            Text("Toto je full-screen cover")
            Button("Zavřít") {
                dismiss()
            }
        }
        .padding()
    }
}
```

TabView (Tab Bar)

- TabView umožňuje rozdělit aplikaci do více sekcí pomocí spodní lišty.
- Každý pohled v TabView může mít svou vlastní navigaci (např. NavigationStack).
- TabView se hodí pro aplikace s více nezávislými obrazovkami: např. „Domů“, „Nastavení“, „Profil“.
- Každá karta má vlastní NavigationStack.
- Ikony a názvy sekcí jsou zobrazeny ve spodní liště (tab baru).

```
TabView {
    NavigationStack {
        Text("Domácí obrazovka")
    }
    .tabItem {
        Label("Domů", systemImage: "house")
    }

    NavigationStack {
        Text("Nastavení aplikace")
    }
    .tabItem {
        Label("Nastavení", systemImage: "gear")
    }
}
```

Doporučené přednášky

- https://developer.apple.com/videos/play/wwdc2022/10054/
- https://developer.apple.com/videos/play/wwdc2022/10001/

Alerty

- Alert je systémové vyskakovací okno, které informuje uživatele nebo žádá potvrzení.
- Slouží pro:
  - potvrzení akce (např. smazání),
  - upozornění na chybu,
  - zobrazování důležitých informací.
  - Zobrazují se modálně a blokují ostatní interakce.

- Základní typy tlačítek:
  - default – běžné potvrzení,
  - cancel – zavření bez akce,
  - destructive – varuje uživatele (červené tlačítko).

Alert

- Alert se zobrazí, pokud je showAlert == true.
- Obsahuje nadpis a dvě akce – potvrzení a zrušení.

```
@State private var showAlert = false

Button("Smazat položku") {
    showAlert = true
}
.alert("Opravdu chcete smazat?",
       isPresented: $showAlert) {
    Button("Ano", role: .destructive) {
        // akce mazání
    }
    Button("Zrušit", role: .cancel) { }
}
```

Pokročilejší alert s textem

- message: umožňuje přidat vlastní text pod nadpis.
- Hodí se např. pro chybová hlášení nebo varování.

```
.alert("Přihlášení se nezdařilo",
       isPresented: $showError) {
    Button("OK", role: .cancel) { }
} message: {
    Text("Zadané údaje nejsou správné.")
}
```

ConfirmationDialog

- ConfirmationDialog je alternativou k Alert, vhodnou pro nabídku více voleb.
- Zobrazí se jako akční list (Action Sheet) – typicky ze spodní části obrazovky.
- Hodí se pro situace typu: „Vyberte způsob“, „Zvolte akci“, „Sdílet přes…“

- Hlavní výhody:
  - Lze definovat více tlačítek s různými rolemi.
  - Může obsahovat title, volitelný message a skupinu akcí.

ConfirmationDialog

- titleVisibility určuje, zda se má nadpis zobrazit.
- Každé tlačítko může mít roli (default, cancel, destructive).
- Vhodné pro iPhone i iPad – adaptivní zobrazení.

```
@State private var showDialog = false
@State private var selectedOption = ""

Button("Zobraz nabídku") {
    showDialog = true
}
.confirmationDialog("Vyber akci",
                    isPresented: $showDialog,
                    titleVisibility: .visible) {

    Button("Upravit") {
        selectedOption = "edit"
    }

    Button("Smazat", role: .destructive) {
        selectedOption = "delete"
    }

    Button("Zrušit", role: .cancel) { }
}
```

Kdy použít Alert vs. ConfirmationDialog?

Situace	Použij
Potvrzení jediné akce	Alert
Upozornění na chybu	Alert
Nabídka více možností	ConfirmationDialog
Výběr akce (např. „Sdílet přes…“)	ConfirmationDialog
Potvrzení destruktivní akce	Alert s role: .destructive

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")
```

Úkol - Přihlášení a správa úkolů

- Vytvořte jednoduchou iOS aplikaci ve SwiftUI, která umožní uživateli:
  - přihlásit se,
  - zobrazit své úkoly,
  - bezpečně uložit přihlašovací informace.

Úkol - Přihlášení a správa úkolů

- Přihlašovací obrazovka (LoginView)
  - Obsahuje pole pro zadání:
    - uživatelského jména,
    - hesla (SecureField),
    - checkbox „Zapamatovat si mě“.
  - Po kliknutí na tlačítko „Přihlásit“:
    - pokud je „Zapamatovat si mě“ aktivní:
      - heslo se uloží do Keychainu,
      - uživatelské jméno do UserDefaults,
    - jinak se uložené hodnoty smažou (Při dalším spuštění je nutné nové přihlášení).
- Pokud jsou údaje již uloženy, přihlášení proběhne automaticky.
- Ověření správnosti umístěte do správné vrstvy aplikace a logika bude mockovaná.

Úkol - Přihlášení a správa úkolů

- Profil uživatele (ProfileView)
    - přihlášené jméno (z UserDefaults),
    - zobrazit informace o uživateli (UserProfile je mockovaný),
    - možnost „Odhlásit se“ (smazání Keychain + UserDefaults).
- Seznam úkolů (TaskListView)
    - Po kliknutí na „Zobrazit úkoly“ se zobrazí seznam úkolů přiřazených aktuálnímu uživateli.
    - Každý úkol obsahuje:
        - název, popis, termín, stav splnění.
    - Úkoly se načítají z mockovaného repozitáře.

Úkol - Přihlášení a správa úkolů

- Detail úkolu (TaskDetailView)
  - Zobrazí podrobnosti úkolu.
  - Možnost označit jako hotový/nehotový.
- Snažte se vytvořit přívětivé UI.