Seminář 2: Úvod do SwiftUI

Tvorba mobilních aplikací

Roman Vyjídáček

Obsah semináře

- SwiftUI vs UIKit
- Práce se stavem ve SwiftUI
- Základní komponenty

SwiftUI vs UIKit

Co je SwiftUI?

- SwiftUI je deklarativní framework pro tvorbu uživatelského rozhraní v iOS, macOS, watchOS a tvOS
- Nahrazuje UIKit a kombinuje návrh UI a kód do jednoho souboru
- Používá Swift jako primární jazyk
- Nahrazuje starší imperativní knihovnu UIKit

Deklarativní vs. Imperativní přístup

SwiftUI (Deklarativní)	UIKit (Imperativní)
Používá struktury (structs)	Používá třídy (classes)
Stav řízen daty (@State, @Binding, @ObservedObject)	Stav řízen explicitně (setNeedsLayout)
Automatická reaktivita	Ruční aktualizace UI
Menší množství kódu	Více řádků kódu, nutná synchronizace

UIKit vs SwiftUI

```swift
import UIKit

class ViewController: UIViewController {
    override func loadView() {
        super.loadView()

        let label = UILabel()
        label.text = "Ahoj, světe!"
        label.font = UIFont.systemFont(ofSize: 34)
        label.translatesAutoresizingMaskIntoConstraints = false

        view.addSubview(label)
        view.backgroundColor = .white

        NSLayoutConstraint.activate([
            label.centerXAnchor.constraint(
               equalTo: view.centerXAnchor
            ),
            label.centerYAnchor.constraint(
               equalTo: view.centerYAnchor
            )
        ])
    }
}
```

UIKit vs SwiftUI

```swift
import SwiftUI

struct ContentView: View {

    var body: some View {
        Text("Ahoj, světe!")
            .font(.largeTitle)
            .padding()
            .frame(maxWidth: .infinity, maxHeight: .infinity)
            .background(Color.white)
    }
}
```

Práce se stavem ve SwiftUI

Jak SwiftUI pracuje se stavem?

- SwiftUI používá deklarativní přístup, což znamená, že nepopisujeme přesně, jak se má UI měnit, ale pouze co se má zobrazit na základě dat.
- Hlavní principy jsou:
  - Data určují UI – pokud se změní data, UI se samo aktualizuje.
  - Jednosměrný tok dat (one-way data flow) – data tečou pouze shora dolů (od modelu k UI).
  - React-like přístup – komponenty sledují změny a reagují automatickým překreslením.

Lokální stav: @State

- Uchovává hodnoty, které patří pouze jednomu View.
- Když se hodnota změní, SwiftUI automaticky překreslí UI.
- Musí být privátní (private) a nesmí se sdílet mezi více View.
- SwiftUI neukládá @State přímo do struct, ale mimo něj.
- Je referenční hodnota, která se propojuje s UI.
- Pokud se změní @State, celý body View se znovu vykreslí.

Použití @State

- @State uchovává číslo count.
- Tlačítko mění hodnotu count.
- SwiftUI detekuje změnu a překreslí celý body View.

Kdy @State NEpoužívat?

- Pokud potřebujeme sdílet stav mezi více View.
- Pokud hodnota nesouvisí s UI (např. síťová data).

```swift
struct CounterView: View {
    // Lokální stav
    @State private var count = 0

    var body: some View {
        VStack {
            Text("Počet: \(count)")
                .font(.largeTitle)

            Button("Zvýšit") {
                // Změna stavu -> překreslí UI
                count += 1
            }
            .padding()
            .background(Color.blue)
            .foregroundColor(.white)
            .cornerRadius(8)
        }
    }
}
```

Sdílený stav: @Binding

- Umožňuje předávat hodnotu mezi dvěma View.
- Není vlastník hodnoty – pouze odkazuje na @State.

Kdy použít @Binding?

- Pokud chceme, aby podřízené View mohlo měnit stav rodiče.
- Když nechceme vytvářet duplikaci stavu.

Použití

- ParentView obsahuje:
  @State private var isOn = false.
- ToggleView ho neukládá – pouze ho získává přes @Binding.
- Když ToggleView změní hodnotu, rodičovský @State se automaticky aktualizuje.

Důležité pravidlo @Binding

- Nemůže existovat sám o sobě – vždy musí být propojen s @State.

```
struct ParentView: View {
    // Hlavní stav
    @State private var isOn = false

    var body: some View {
        // Sdílení pomocí Binding
        ToggleView(isOn: $isOn)
    }
}

struct ToggleView: View {
    // Přijímá stav jako Binding
    @Binding var isOn: Bool

    var body: some View {
        Toggle("Zapnuto", isOn: $isOn)
            .padding()
    }
}
```

Složitější data: @ObservedObject a @Published

Co je @ObservedObject?

- Používá se pro větší datové modely, které obsahují více proměnných.
- Sdílí stav mezi více částmi aplikace (např. ViewModel, další View).
- @ObservedObject sleduje objekt, který dědí od ObservableObject.

Co je @Published?

- @Published říká SwiftUI, které proměnné mají spustit překreslení UI.

Kdy použít @ObservedObject?

- Pokud máme komplexní data (např. uživatelské nastavení).
- Potřebujeme, aby změny stavu ovlivňovaly více View.
- Chceme oddělit logiku od UI.

Použití

- CounterState dědí z ObservableObject a obsahuje @Published var count.
- CounterState používá @ObservedObject, aby sledovalo změny.
- Když se změní state.count, UI se automaticky aktualizuje.

Rozdíl oproti @State?

- @State je lokální pro jedno View.
- @ObservedObject je sdílený napříč View.

```
final class CounterState: ObservableObject {
    // Změna této hodnoty překreslí UI
    @Published var count = 0
}

struct CounterView: View {
    // Propojení s modelem
    @ObservedObject var state = CounterState()

    var body: some View {
        VStack {
            Text("Počet: \(state.count)")
                .font(.largeTitle)

            Button("Zvýšit") {
                state.count += 1
            }
        }
    }
}
```

Globální stav: @EnvironmentObject

Co je @EnvironmentObject?

- Nejlepší způsob, jak sdílet data v celé aplikaci.
- Odstraňuje potřebu předávat @ObservedObject přes všechny View.
- Funguje jako Dependency Injection – View si ho může vyžádat.

Kdy použít @EnvironmentObject?

- Pokud máme globální stav, který se používá na více obrazovkách.
- Pokud nechceme složitě předávat @ObservedObject.

Příklad

- UserSettings je globální model.
- ContentView ho nastaví jako sdílený pomocí .environmentObject(settings).

```
class UserSettings: ObservableObject {
    @Published var username: String = "Default User"
}

struct ContentView: View {
    @StateObject var settings = UserSettings()

    var body: some View {
        NavigationView {
            VStack {
                NavigationLink(
                    "Přejít na Detail",
                    destination: DetailView()
                )
                Text("Uživatel: \(settings.username)")
            }
        }
        // Sdílení přes Environment
        .environmentObject(settings)
    }
}
```

Příklad - pokračování

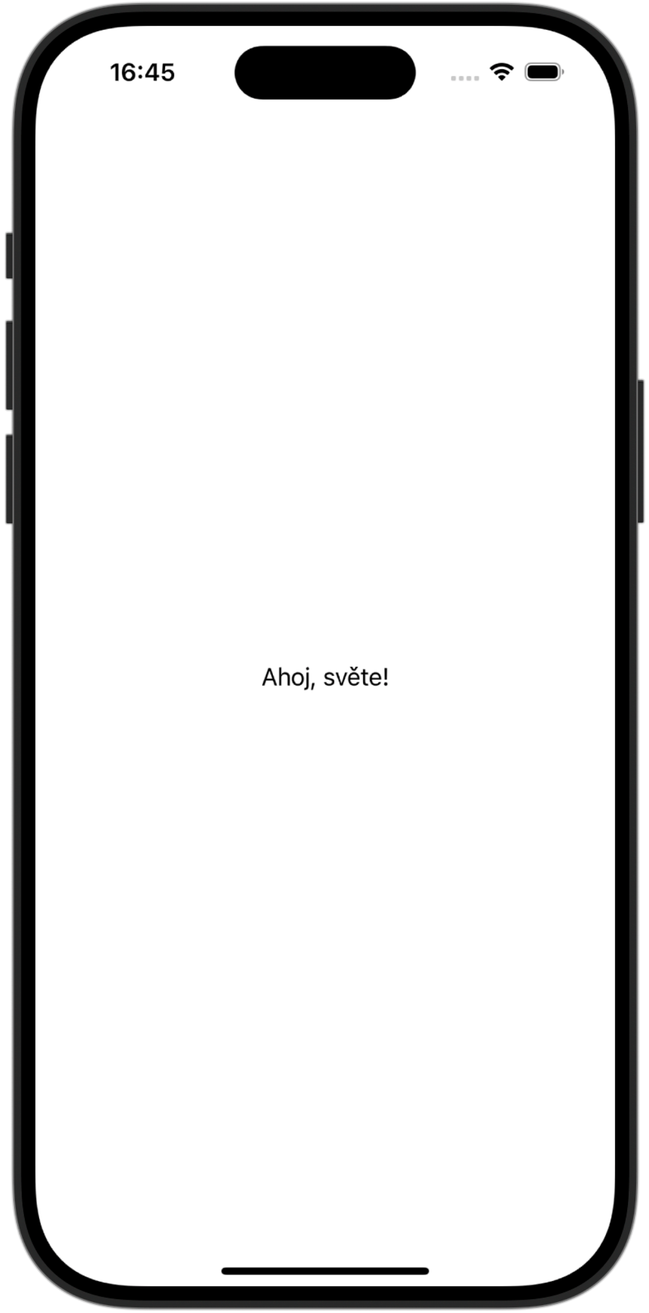
- DetailView získá sdílený objekt pomocí @EnvironmentObject bez nutnosti explicitního předávání.

```
struct DetailView: View {
    // Automatické načtení
    @EnvironmentObject
    var settings: UserSettings

    var body: some View {
        VStack {
            Text("Uživatel: \(settings.username)")
            TextField(
                "Nové jméno",
                text: $settings.username
            )
            .textFieldStyle(
                RoundedBorderTextFieldStyle()
            )
        }
    }
}
```

Základní komponenty

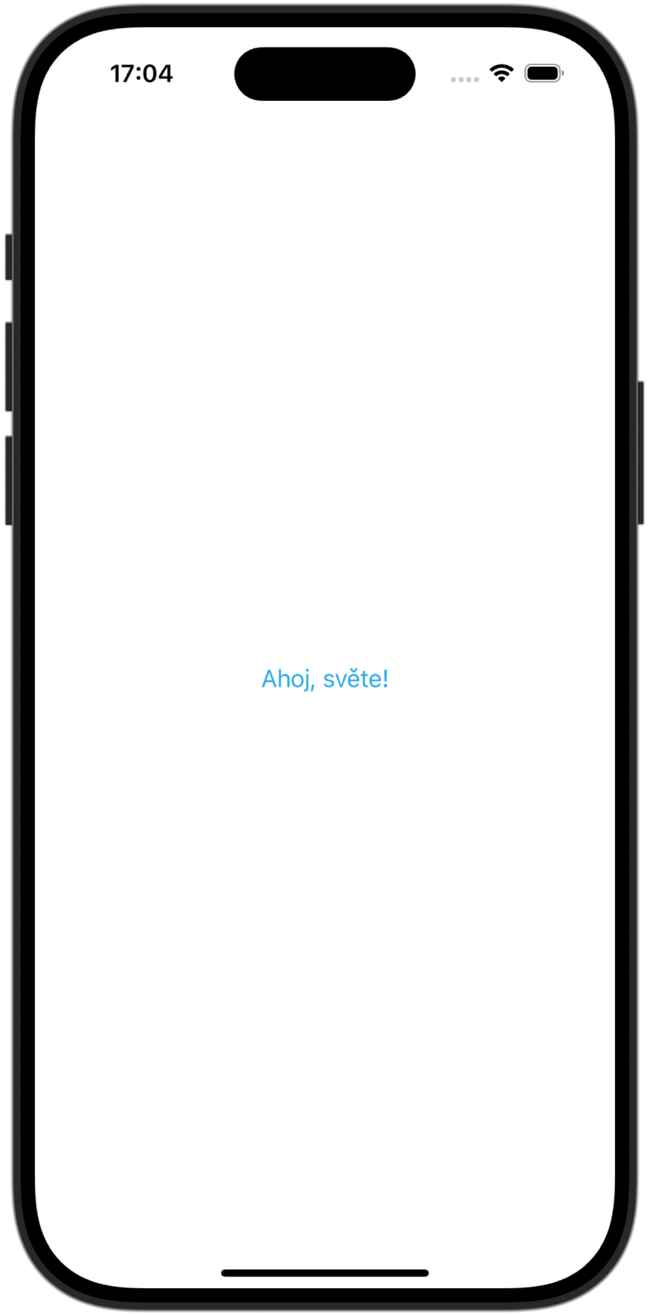
Text

```
// - Slouží k zobrazení statického textu

struct ContentView: View {

    var body: some View {
        Text("Ahoj, světe!")
    }
}
```



Text

```
// - Slouží k zobrazení statického textu

struct ContentView: View {

    var body: some View {
        Text("Ahoj, světe!")
//            Nastavíme barvu textu
            .foregroundStyle(.cyan)
    }
}
```



Text

```
// - Slouží k zobrazení statického textu

struct ContentView: View {

    var body: some View {
        Text("Ahoj, světe!")
//            Nastavíme barvu textu
            .foregroundStyle(.cyan)
//            Zvětšíme font
            .font(.largeTitle)
    }
}
```

Text

```
// - Slouží k zobrazení statického textu

struct ContentView: View {

    var body: some View {
        Text("Ahoj, světe!")
//            Nastavíme barvu textu
            .foregroundStyle(.cyan)
//            Zvětšíme font
            .font(.largeTitle)
//            Nastavíme barvu pozadí
            .background(.yellow)
    }
}
```

Text

```
// - Slouží k zobrazení statického textu

struct ContentView: View {

    var body: some View {
        Text("Ahoj, světe!")
//            Nastavíme barvu textu
            .foregroundStyle(.cyan)
//            Zvětšíme font
            .font(.largeTitle)
//            Nastavíme padding
            .padding()
//            Nastavíme barvu pozadí
            .background(.yellow)

    }
}
```

Text

```
// - Slouží k zobrazení statického textu

struct ContentView: View {

    var body: some View {
        Text("Ahoj, světe!")
//            Nastavíme barvu textu
            .foregroundStyle(.cyan)
//            Zvětšíme font
            .font(.largeTitle)
//            Nastavíme padding
            .padding()
//            Nastavíme barvu pozadí
            .background(.yellow)
//            Nastavíme zakulacení rohů
            .cornerRadius(16)
    }
}
```

Button

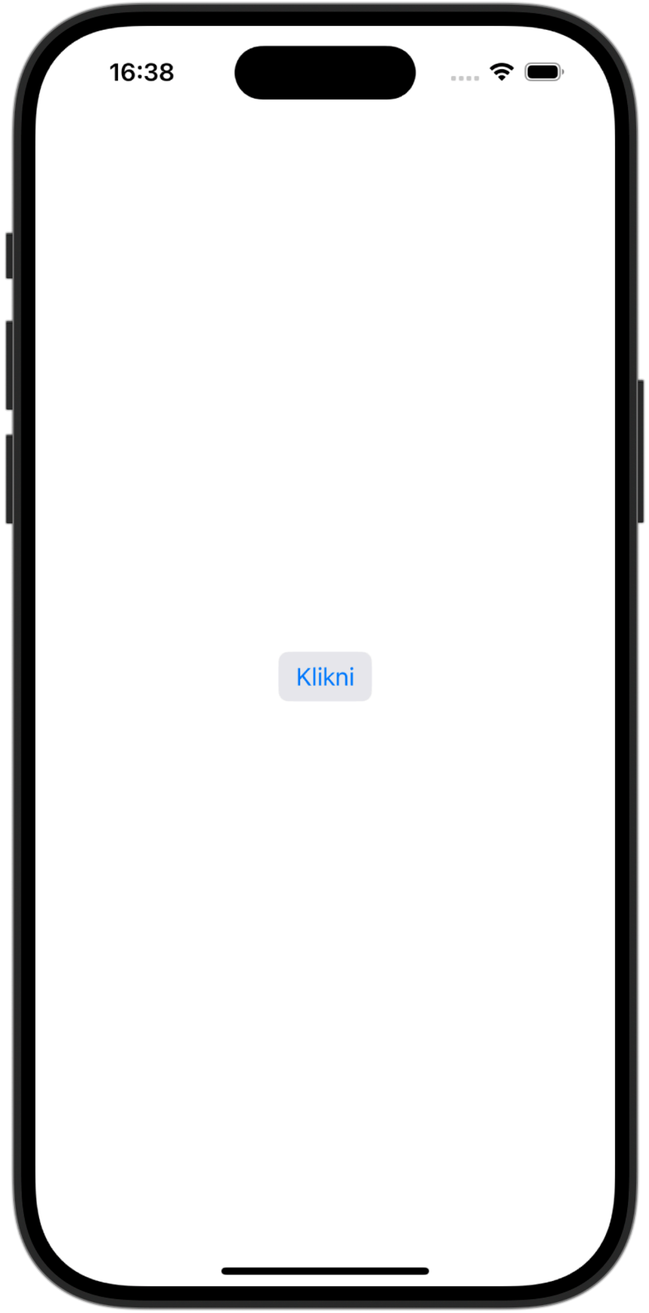
- Komponenta Button vytváří interaktivní tlačítko.

Základní parametry:

- *action*: Closure, která se vykoná po stisknutí tlačítka.
- *label*: Zobrazuje obsah tlačítka, obvykle Text, ale může obsahovat libovolné View.
- title: Text tlačítka

Button

```
struct ContentView: View {

    var body: some View {
        Button("Klikni") {
            print("Tlačítko stisknuto")
        }
//          UI styl tlačítka
        .buttonStyle(.bordered)
    }
}
```



Button

```
struct ContentView: View {

    var body: some View {
        Button(action: {
            print("Tlačítko stisknuto")
        }) {
            Text("Klikni zde")
                .font(.headline)
                .padding()
                .background(Color.green)
                .foregroundColor(.white)
                .cornerRadius(10)
        }
    }
}
```

TextField

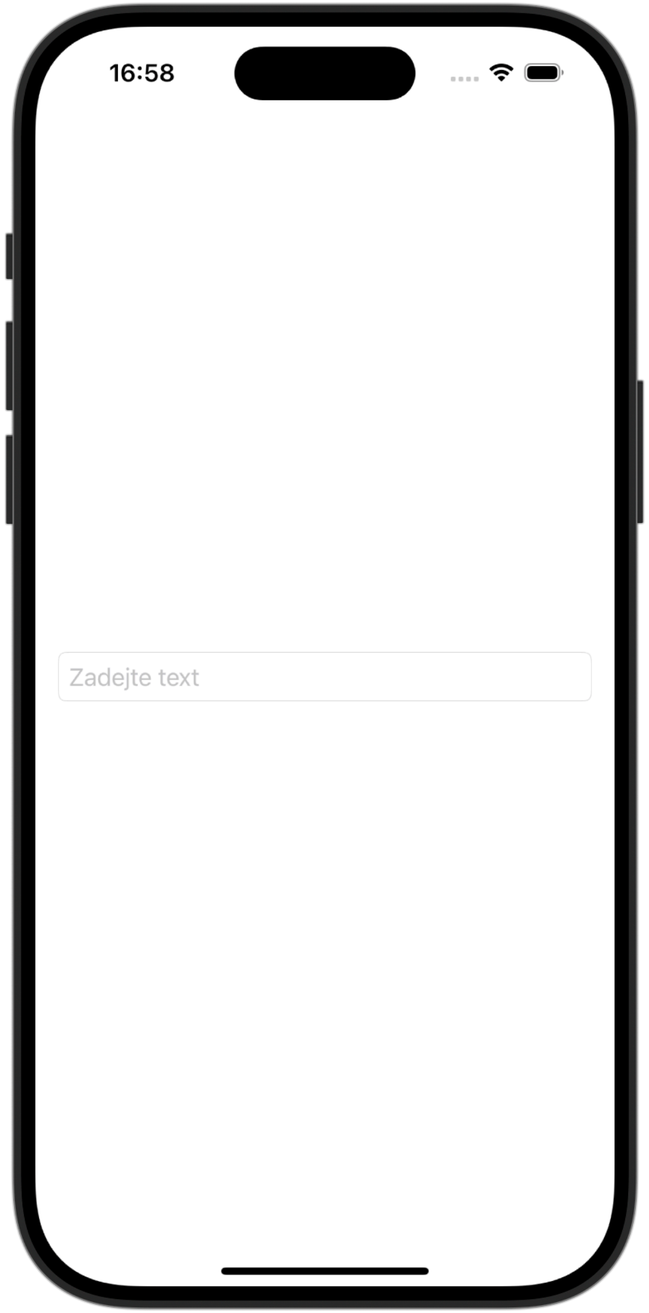
- TextField umožňuje uživateli zadávat text.

Základní parametry:

- *text*: Binding na proměnnou, která uchovává zadaný text.
- *placeholder*: Text, který se zobrazí, pokud je textové pole prázdné.

TextField

```
struct ContentView: View {

    @State private var input: String = ""

    var body: some View {
        TextField("Zadejte text", text: $input)
            .textFieldStyle(.roundedBorder)
            .padding()
    }
}
```

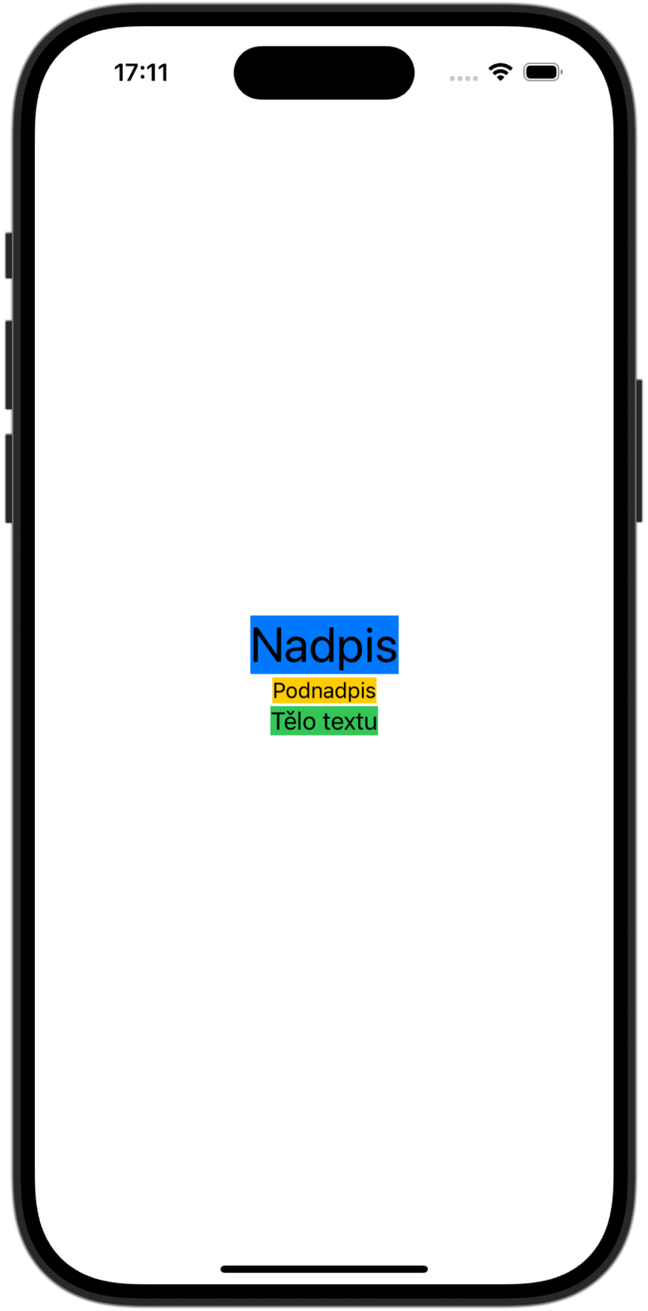


Stacks (HStack, VStack, ZStack)

- Stacky slouží k uspořádání více Views do horizontální (HStack), vertikální (VStack) nebo překrývající se (ZStack) vrstvy.
- Základní parametry:
- *alignment:* Zarovnání Views (např. .leading, .center, .trailing).
- *spacing:* Odstup mezi podřízenými Views.

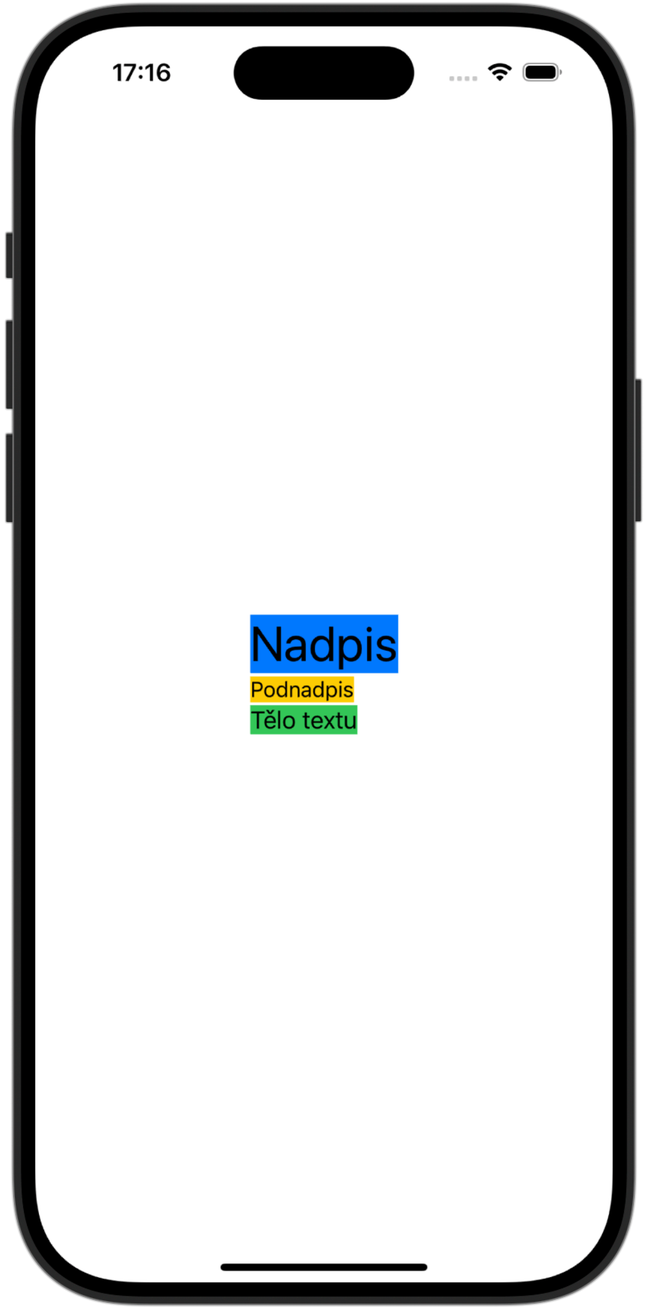
VStack

```
struct ContentView: View {

    var body: some View {
        VStack {
            Text("Nadpis")
                .font(.largeTitle)
                .background(Color.blue)
            Text("Podnadpis")
                .font(.subheadline)
                .background(Color.yellow)
            Text("Tělo textu")
                .font(.body)
                .background(Color.green)
        }
        .padding()
    }
}
```



VStack

```
struct ContentView: View {

    var body: some View {
        VStack(alignment: .leading) {
            Text("Nadpis")
                .font(.largeTitle)
                .background(Color.blue)
            Text("Podnadpis")
                .font(.subheadline)
                .background(Color.yellow)
            Text("Tělo textu")
                .font(.body)
                .background(Color.green)
        }
        .padding()
    }
}
```

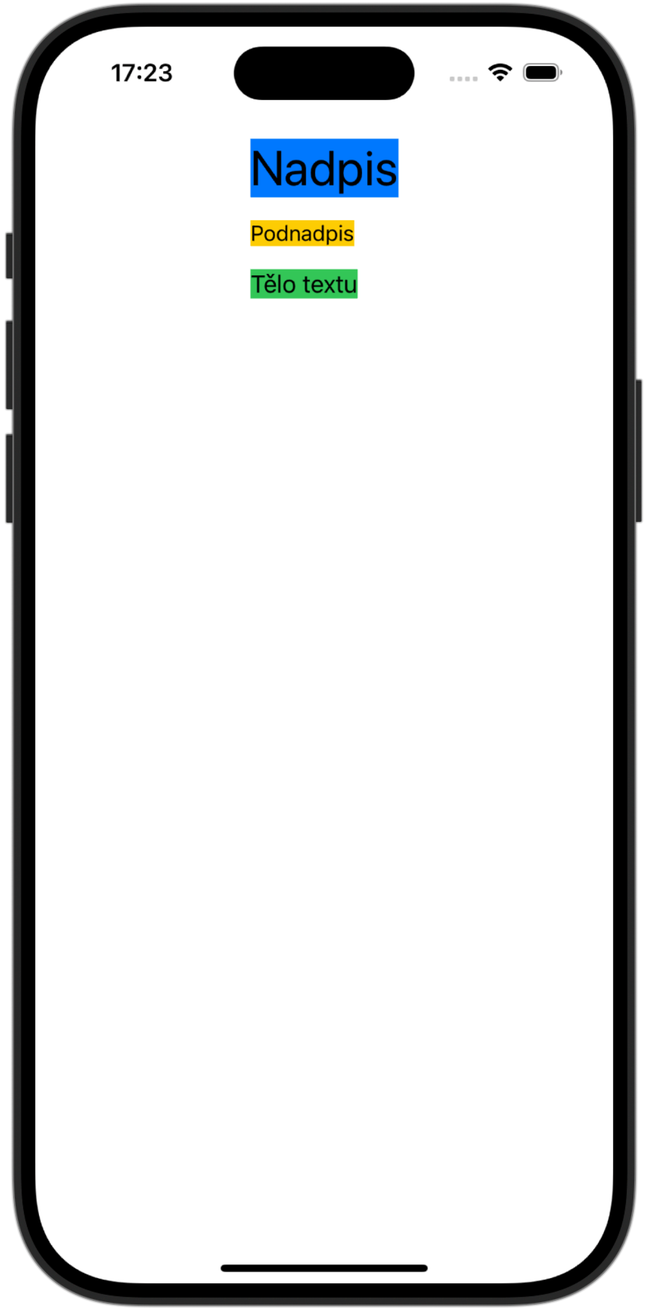


VStack

```
struct ContentView: View {

    var body: some View {
        VStack(alignment: .leading, spacing: 16) {
            Text("Nadpis")
                .font(.largeTitle)
                .background(Color.blue)
            Text("Podnadpis")
                .font(.subheadline)
                .background(Color.yellow)
            Text("Tělo textu")
                .font(.body)
                .background(Color.green)
        }
        .padding()
    }
}
```

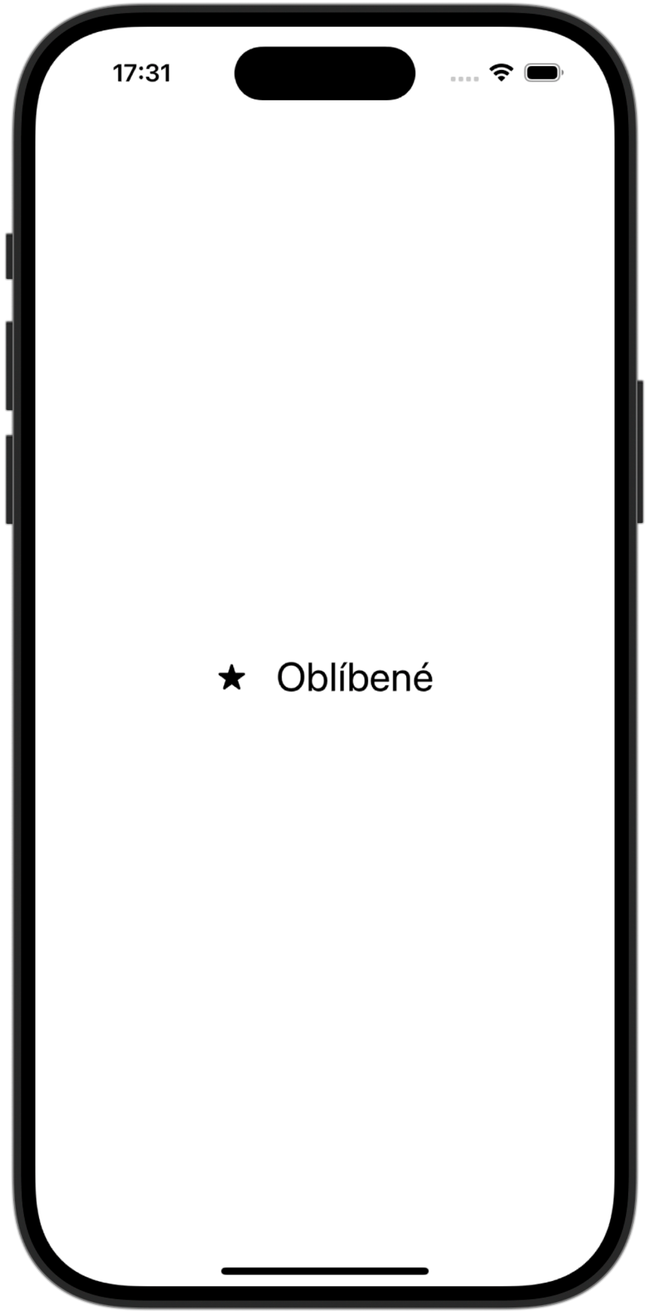
VStack

```
struct ContentView: View {

    var body: some View {
        VStack(alignment: .leading, spacing: 16) {
            Text("Nadpis")
                .font(.largeTitle)
                .background(Color.blue)
            Text("Podnadpis")
                .font(.subheadline)
                .background(Color.yellow)
            Text("Tělo textu")
                .font(.body)
                .background(Color.green)
//              Spacer vyplňuje dostupný prostor
            Spacer()
        }
        .padding()
    }
}
```



HStack

```
struct ContentView: View {

    var body: some View {
        HStack(spacing: 20) {
            Image(systemName: "star.fill")
            Text("Oblíbené")
                .font(.title)
        }
        .padding()
    }
}
```



ZStack

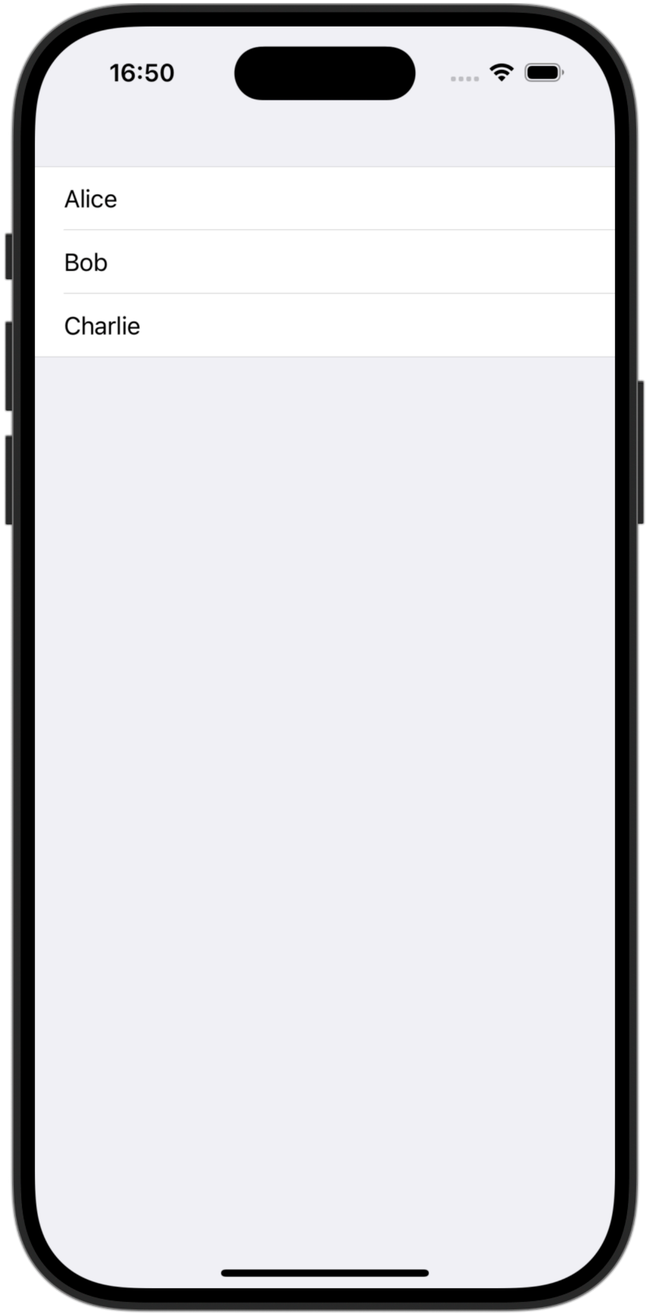
```
struct ContentView: View {

    var body: some View {
        ZStack {
            RoundedRectangle(cornerRadius: 16)
                .fill(Color.cyan)
                .frame(width: 300, height: 200)
            Text("Překryvný text")
                .font(.largeTitle)
                .foregroundColor(.white)
                .padding(10)
                .background(Color.black.opacity(0.7))
                .cornerRadius(5)
        }
    }
}
```

List

- List je komponenta pro zobrazení seznamů v SwiftUI.
- Funguje podobně jako UITableView v UIKit.
- Automaticky spravuje vykreslování, výkon, lazy načítání a interaktivitu.
- Recykluje Views → znovu použije již vytvořené View, ale s jinými hodnotami.
- Musíme zajistit, že hodnoty jsou unikátní:
  - Model implementuje protocol Identifiable (vyžaduje property id)
  - Ručně definujeme která property je unikátní

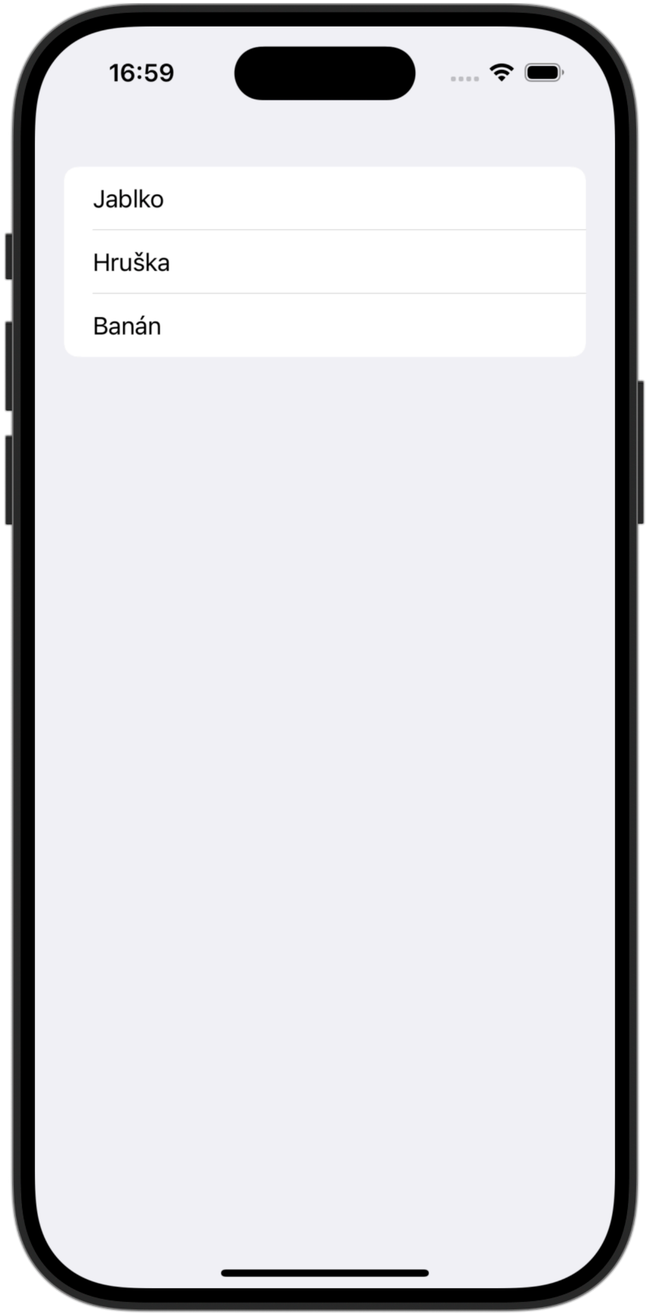
List

```
struct SimpleListView: View {
    let names = ["Alice", "Bob", "Charlie"]

    var body: some View {
//        Pomocí id: \.self říkáme že name je unikátní
        List(names, id: \.self) { name in
            Text(name)
        }
//        Definujeme styl zobrazení
        .listStyle(.grouped)
    }
}
```



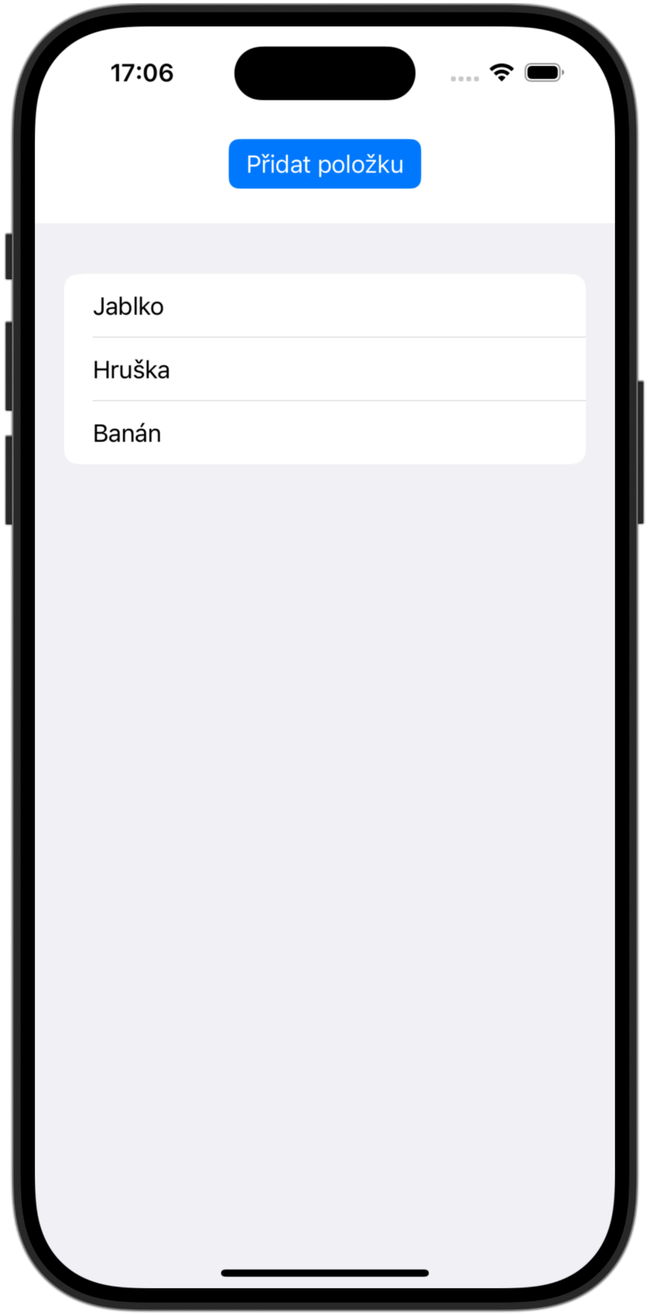
List - dynamická data

```
struct SimpleListView: View {

    @State
    private var items = ["Jablko", "Hruška", “Banán"]

    var body: some View {
        List(items, id: \.self) { item in
            Text(item)
        }
    }
}
```



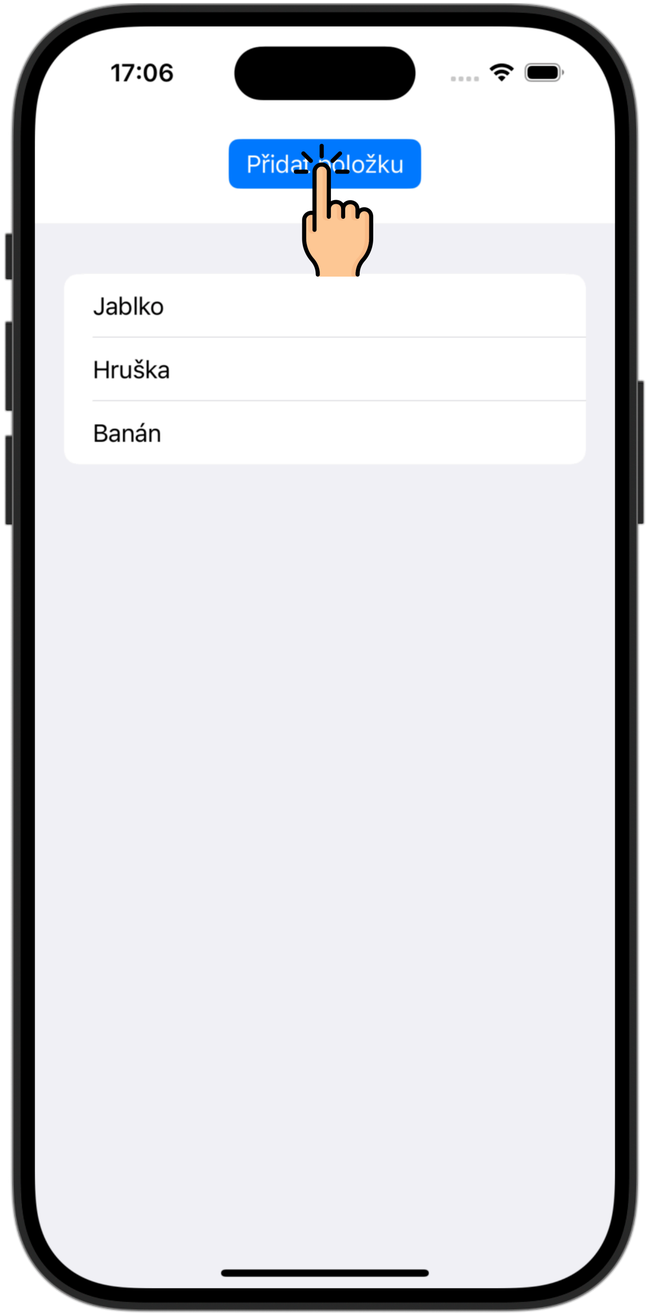
List - přidání položky

```
struct SimpleListView: View {

    @State
    private var items = ["Jablko", "Hruška", "Banán"]

    var body: some View {
        VStack {
            Button("Přidat položku") {
                items.append(
                    "Nová položka \(items.count + 1)"
                )
            }
            .buttonStyle(.borderedProminent)
            .padding()

            List(items, id: \.self) { item in
                Text(item)
            }
        }
    }
}
```



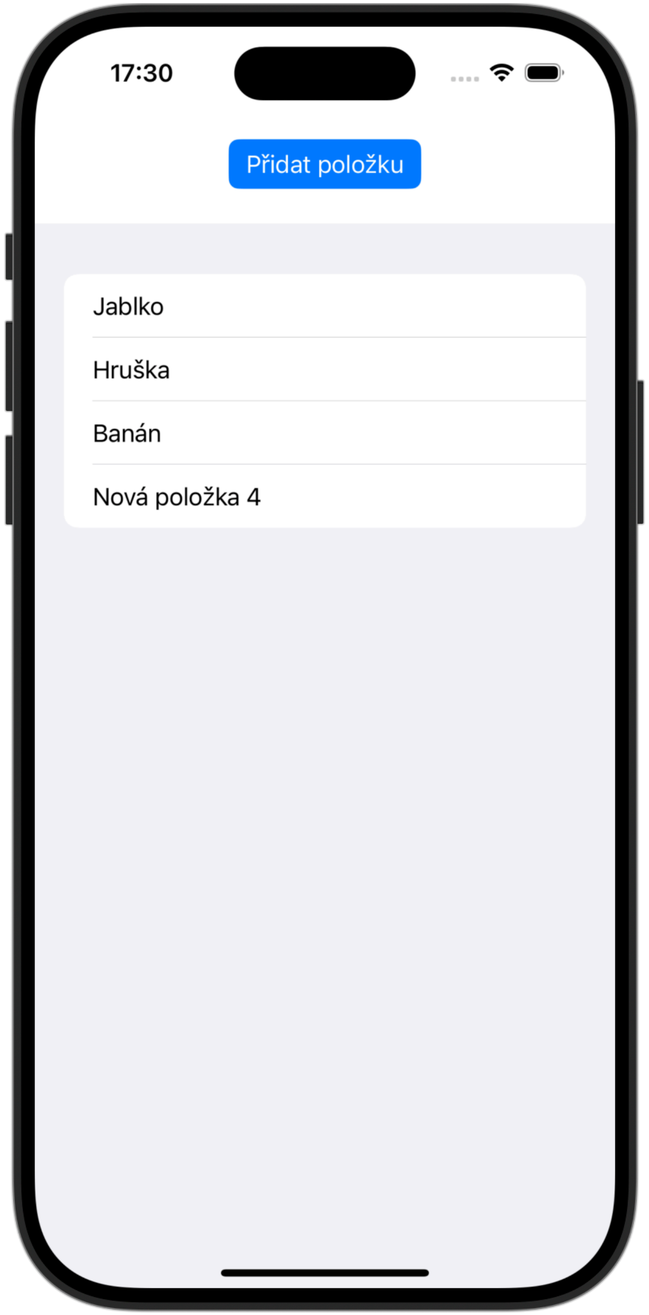
List - přidání položky

```
struct SimpleListView: View {

    @State
    private var items = ["Jablko", "Hruška", "Banán"]

    var body: some View {
        VStack {
            Button("Přidat položku") {
                items.append(
                    "Nová položka \(items.count + 1)"
                )
            }
            .buttonStyle(.borderedProminent)
            .padding()

            List(items, id: \.self) { item in
                Text(item)
            }
        }
    }
}
```



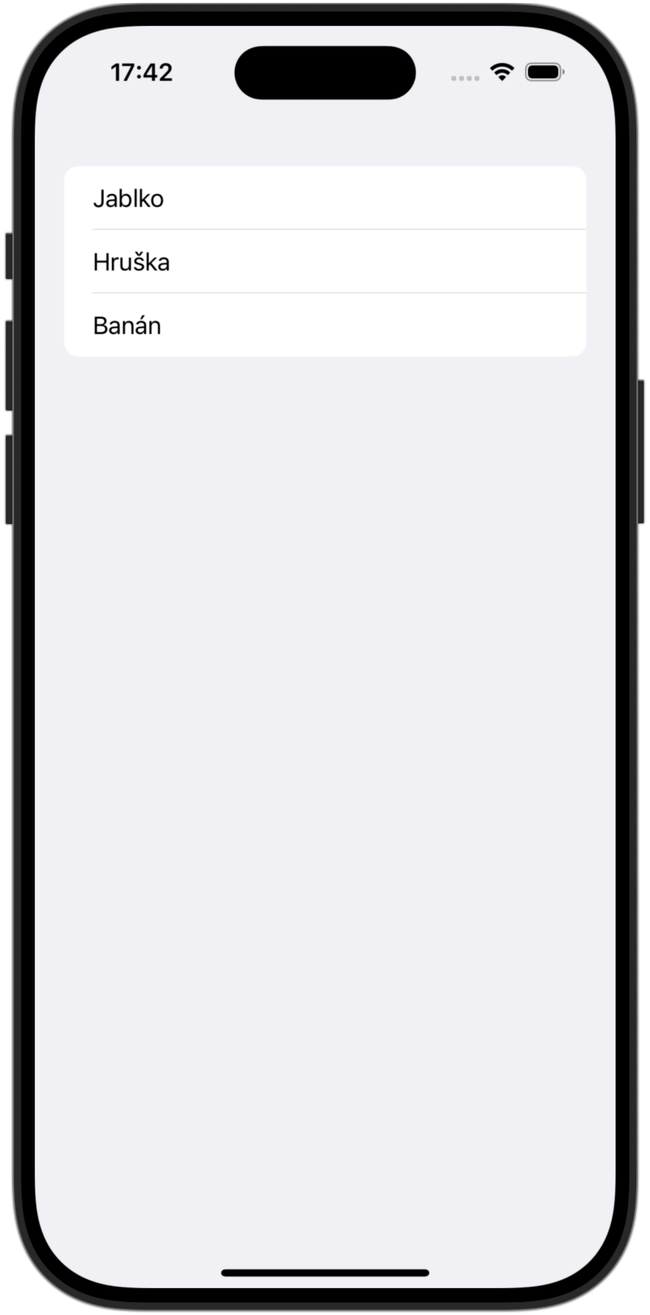
List - přidání položky

```
struct SimpleListView: View {

    @State
    private var items = ["Jablko", "Hruška", "Banán"]

    var body: some View {
        VStack {
            Button("Přidat položku") {
                items.append(
                    "Nová položka \(items.count + 1)"
                )
            }
            .buttonStyle(.borderedProminent)
            .padding()

            List(items, id: \.self) { item in
                Text(item)
            }
        }
    }
}
```



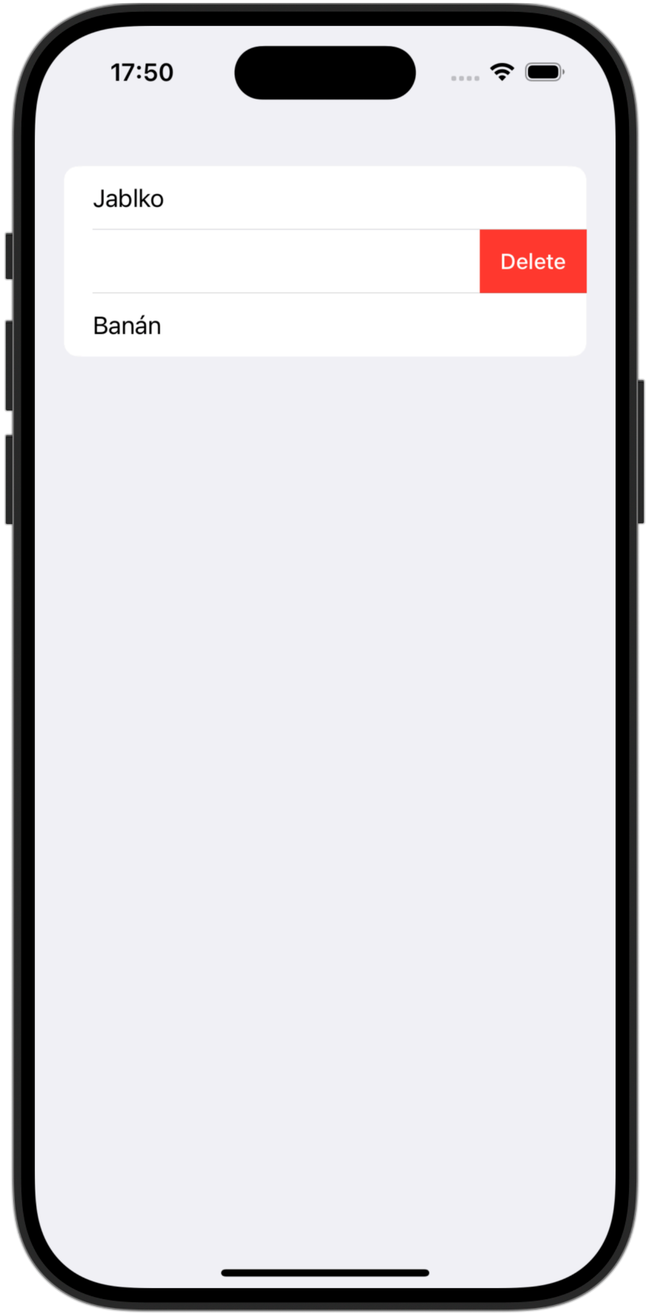
List - smazání položky

```
struct SimpleListView: View {

    @State
    private var items = ["Jablko", "Hruška", "Banán"]

    var body: some View {
        List {
            ForEach(items, id: \.self) { item in
                Text(item)
            }
            .onDelete { indexSet in
                items.remove(atOffsets: indexSet)
            }
        }
    }
}
```



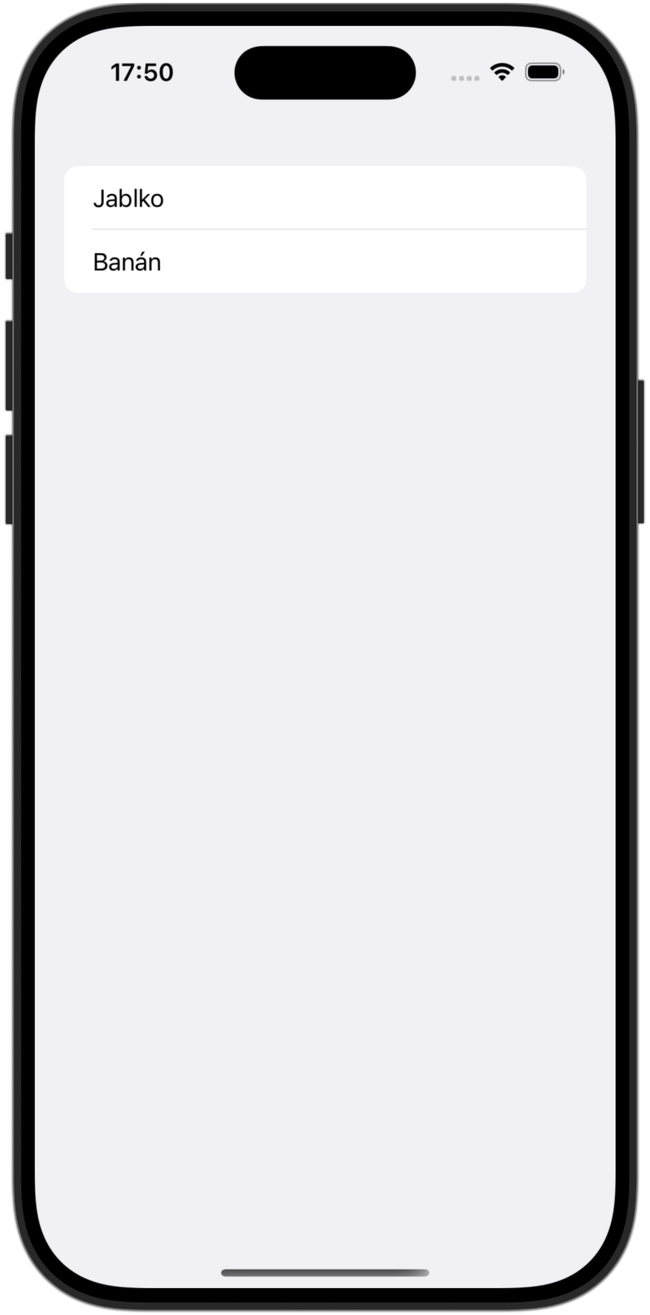
List - smazání položky

```
struct SimpleListView: View {

    @State
    private var items = ["Jablko", "Hruška", "Banán"]

    var body: some View {
        List {
            ForEach(items, id: \.self) { item in
                Text(item)
            }
            .onDelete { indexSet in
                items.remove(atOffsets: indexSet)
            }
        }
    }
}
```



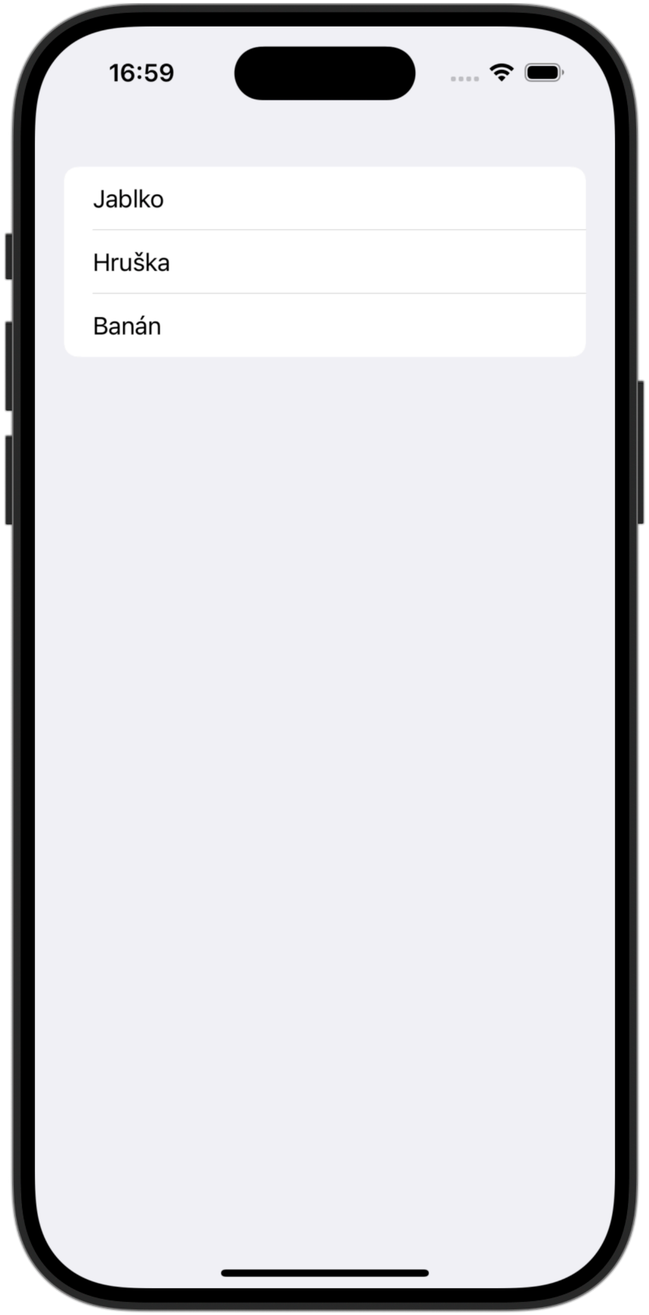
List - smazání položky

```
struct SimpleListView: View {

    @State
    private var items = ["Jablko", "Hruška", "Banán"]

    var body: some View {
        List {
            ForEach(items, id: \.self) { item in
                Text(item)
            }
            .onDelete { indexSet in
                items.remove(atOffsets: indexSet)
            }
        }
    }
}
```



List - reakce na gesta

```
struct SimpleListView: View {

    @State
    private var items = ["Jablko", "Hruška", "Banán"]

    var body: some View {
        List {
            ForEach(items, id: \.self) { item in
                Text(item)
//                Pozor tap gesture funguje jen na text!
                  .onTapGesture {
                      print("Vybrána položka \(item)")
                  }
            }
        }
    }
}
```



LazyVStack

- LazyVStack je vertikální stack, který vykresluje pouze viditelné prvky, čímž zlepšuje výkon.
- Funguje uvnitř ScrollView – sám neposkytuje posouvání.

Proč používat LazyVStack?

- Zlepšuje výkon u velkých seznamů (např. 100+ prvků).
- Ušetří paměť tím, že vykresluje jen to, co je vidět.
- Je nutný v ScrollView, protože List už scrollování obsahuje.

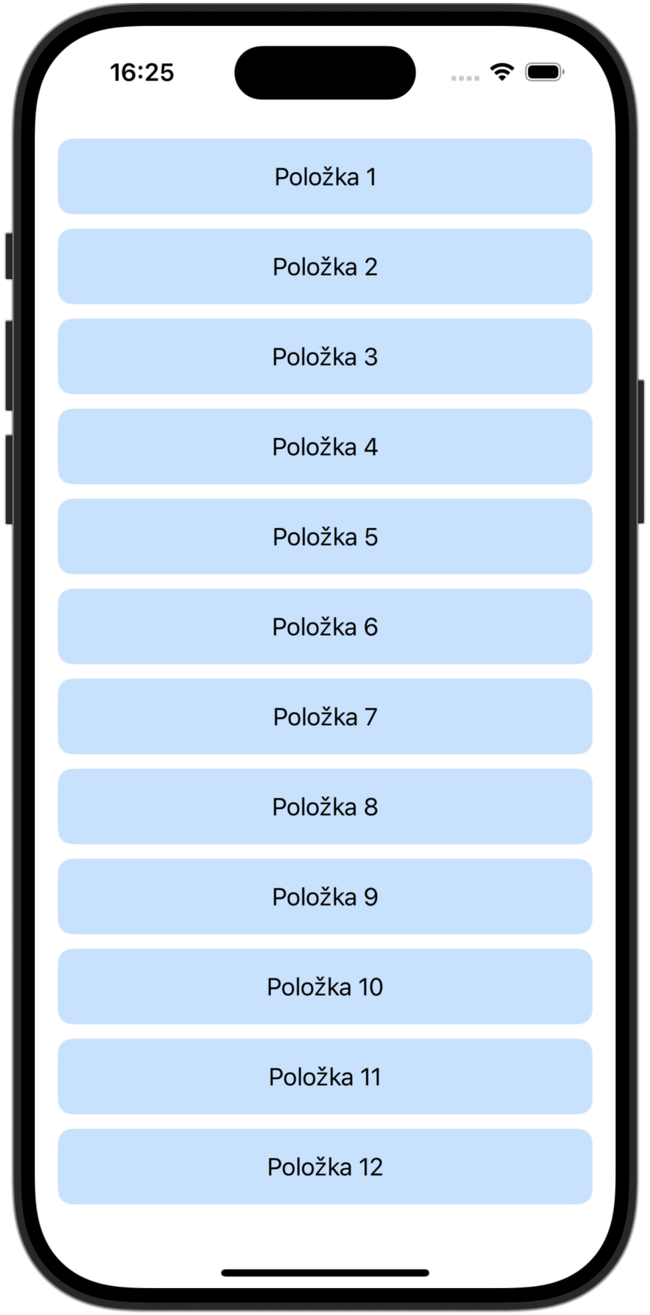
LazyVStack

Kdy použít LazyVStack místo List?

- Pokud potřebuješ plnou kontrolu nad vzhledem (např. vlastní stylizace).
- Pokud chceš kombinovat více různých stacků v jednom ScrollView.
- Pokud chceš nested scrollování (např. ScrollView s více LazyVStack uvnitř).

LazyVStack

```
struct LazyVStackExample: View {
    let items = Array(1...100)

    var body: some View {
        ScrollView {
            LazyVStack(spacing: 10) {
                ForEach(items, id: \.self) { item in
                    Text("Položka \(item)")
                        .padding()
                        .frame(maxWidth: .infinity)
                        .background(.blue.opacity(0.2))
                        .cornerRadius(10)
                        .onTapGesture {
                            print(
                                "Vybrána položka \(item)"
                            )
                        }
                }
            }
        }
        .padding()
    }
}
```



Preview

- Umožňují vývojářům zobrazit živý náhled uživatelského rozhraní přímo v Xcode během vývoje.
- Okamžitá zpětná vazba při úpravách kódu.
- Možnost testovat různé konfigurace, například světlý/tmavý režim, různé velikosti textu nebo lokalizace.

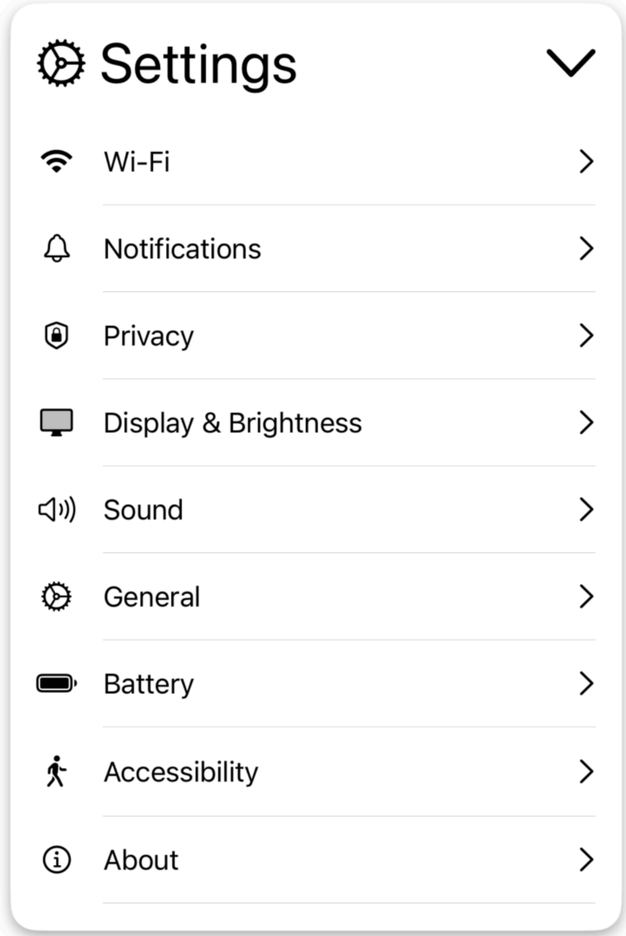
```
struct ContentView: View {

    var body: some View {
        Text("Ahoj, světe!")
    }
}

#Preview {
    ContentView()
}
```

Úkol

- Naprogramujte ExpandableSectionView:
  - Po kliknutí na nadpis se zobrazí obsah této sekce.
  - Nadpis, Obrázek i obsah jsou uživatelem definované.

Nápověda:

- Ikony naleznete v SF Symbols.
- Obsah sekce → Generické View ve SwiftUI.
- Klikatelnou (hlavička) oblast označte pomocí .contentShape(…).



Úkol

- Naprogramujte ScheduleView, které bude zobrazovat rozvrh studenta podobně jako v UPlikaci.
  - Data budou statická není nutné je stahovat ze serveru
  - Implementujte pouze zobrazení rozvrhových akcí (Oblast v červeném obdélníku)