Seminář 4: Práce se stavem ve SwiftUI

Tvorba mobilních aplikací

Roman Vyjídáček

Obsah semináře

- Práce se stavem ve SwiftUI
- Komponenty pracující se stavem

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).

```
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

- @Binding umožňuje sdílet stav mezi Views
- Binding neukládá vlastní data, odkazuje na stav jinde
- Typicky se používá pro komunikaci mezi rodičem a potomkem
- Změna hodnoty přes @Binding mění původní stav

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 implementuje 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 implementuje 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()
            )
        }
    }
}
```

@StateObject

- @StateObject slouží jako jediný source of truth pro referenční typy (class) ve SwiftUI
- Používá se pro ObservableObject, který je vlastněn konkrétním View
- SwiftUI vytvoří instanci objektu pouze jednou po dobu životnosti View
- Změna @Published vlastností automaticky aktualizuje závislá Views

Přklad

- State object lze sdílet s podřízenými Views
- Do subview se předává pomocí: @ObservedObject nebo @EnvironmentObject
- K vlastnostem state objektu
- lze získat Binding pomocí $
- @StateObject se vždy deklaruje jako private

```
struct MyView: View {
    @StateObject
    private var model = DataModel()

    var body: some View {
        MySubView()
            .environmentObject(model)
    }
}

struct MySubView: View {
    @EnvironmentObject var model: DataModel

    var body: some View {
        Toggle("Enabled",
               isOn: $model.isEnabled)
    }
}
```

Příklad inicializace pomocí externích dat

- @StateObject lze inicializovat i pomocí externích dat ale pouze jednou během životnosti View
- Změna vstupních parametrů automaticky NEzmění stav objektu

```
struct MyInitializableView: View {
    @StateObject
    private var model: DataModel

    init(name: String) {
        _model = StateObject(
            wrappedValue: DataModel(
              name: name
            )
        )
    }

    var body: some View {
        VStack {
            Text("Name: \(model.name)")
        }
    }
}
```

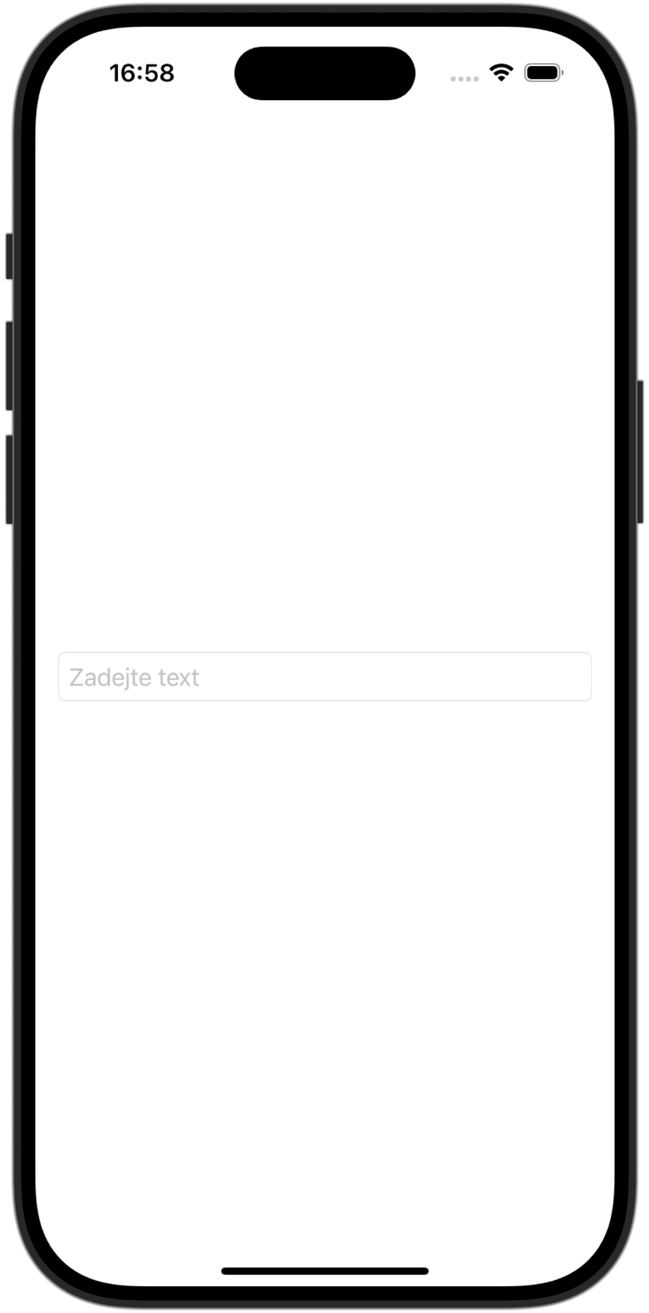
Komponenty pracující se stavem

TextField

- TextField slouží k zadávání textu uživatelem
- Hodnota TextFieldu je vždy vázána na stav pomocí Binding
- Změna textu aktualizuje stav a změna stavu aktualizuje UI

TextField

```
struct ContentView: View {

    @State private var input: String = ""

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

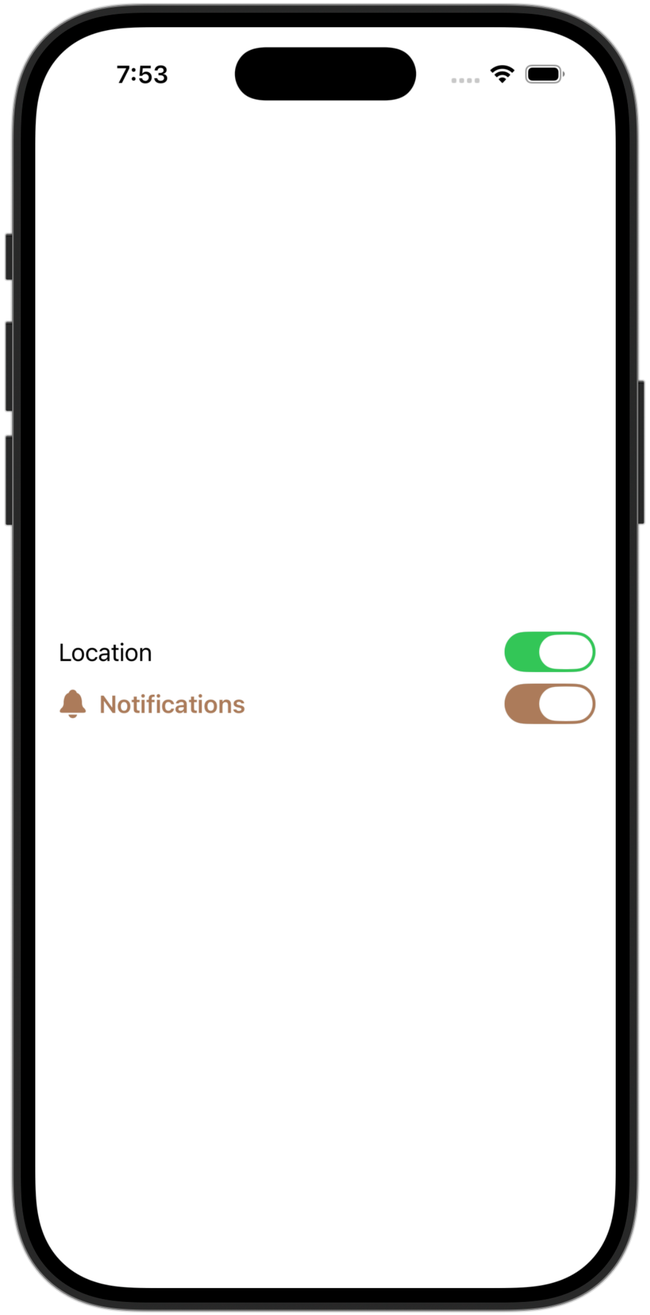


Toggle

- Toggle přijímá Binding na Bool hodnotu
- Popisek Toggle může být text nebo libovolné View
- Toggle lze vytvořit s implicitním nebo vlastním label View
- Vzhled Toggle lze ovlivnit stylem

Toggle

```
struct ContentView: View {
    @State private var isLocationOn = false
    @State private var isNotificationOn = true

    var body: some View {
        Toggle("Location", isOn: $isLocationOn)

        Toggle(isOn: $isNotificationOn) {
            HStack {
                Image(systemName: "bell.fill")
                    .resizable()
                    .scaledToFit()
                    .frame(width: 20, height: 20)
                    .foregroundStyle(.brown)

                Text("Notifications")
                    .font(.headline)
                    .foregroundStyle(.brown)
            }
        }
        .tint(.brown)
    }
}
```

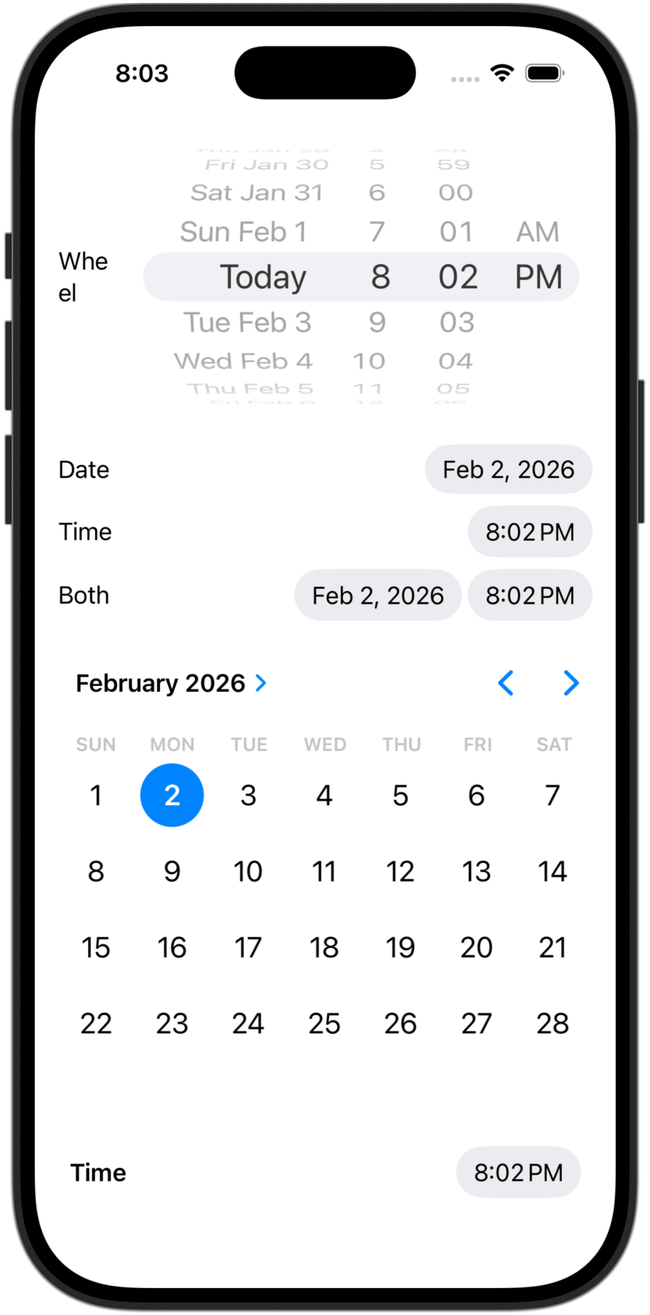


DatePicker

- DatePicker slouží k výběru data a času
- Vybraná hodnota je vždy vázána na stav pomocí Binding<Date>
- Změna data automaticky aktualizuje stav a znovu vykreslí View
- DatePicker lze omezit na datum, čas nebo jejich kombinaci

DatePicker

```
struct ContentView: View {
//    Jen pro ucely prezentace.
//    Kazdy DatePicker musi mit Binding na vlastni promennou
    @State private var date = Date.now

    var body: some View {
        DatePicker("Wheel", selection: $date)
            .datePickerStyle(.wheel)

        DatePicker("Date",
                   selection: $date,
                   displayedComponents: .date)

        DatePicker("Time",
                   selection: $date,
                   displayedComponents: .hourAndMinute)

        DatePicker("Both", selection: $date)

        DatePicker("Graphical", selection: $date)
            .datePickerStyle(.graphical)
    }
}
```

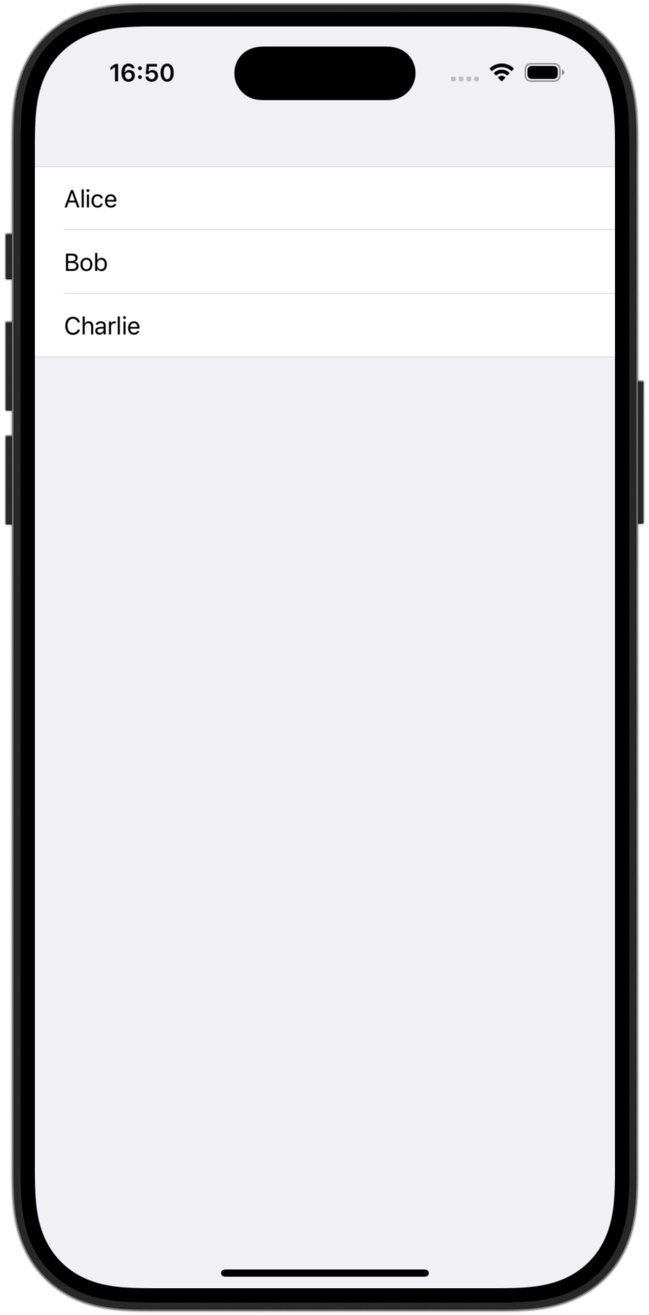


List

- 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

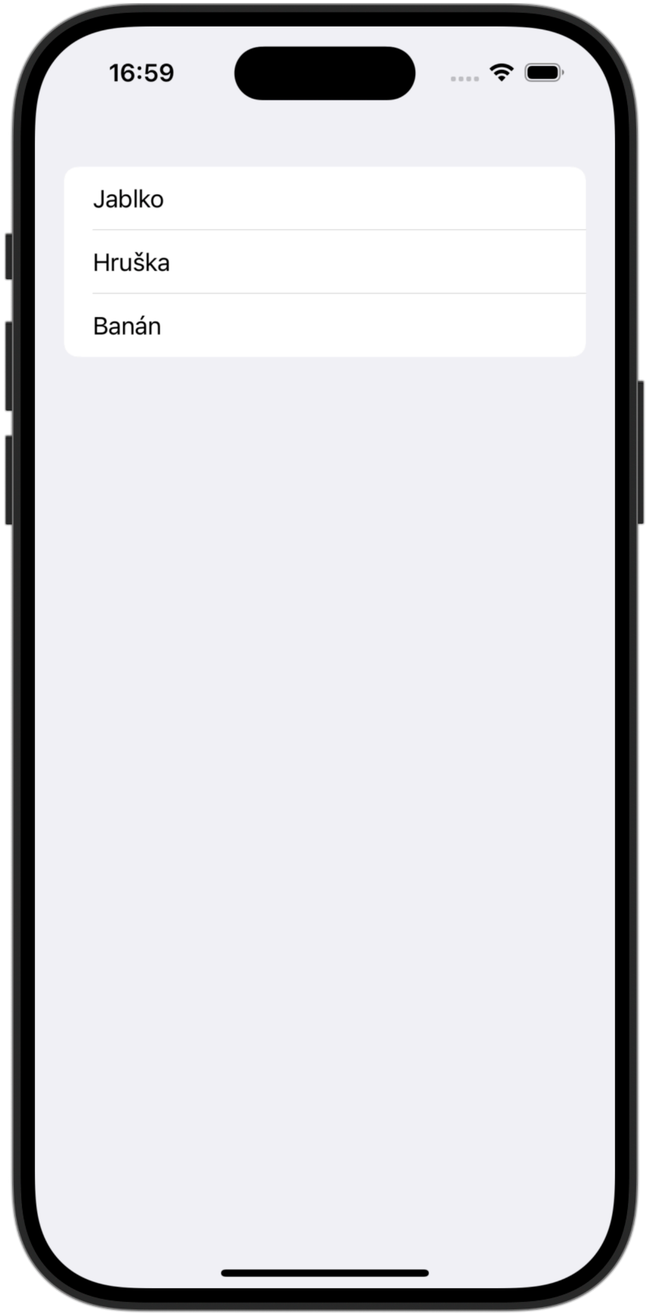
List

```
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)
    }
}
```



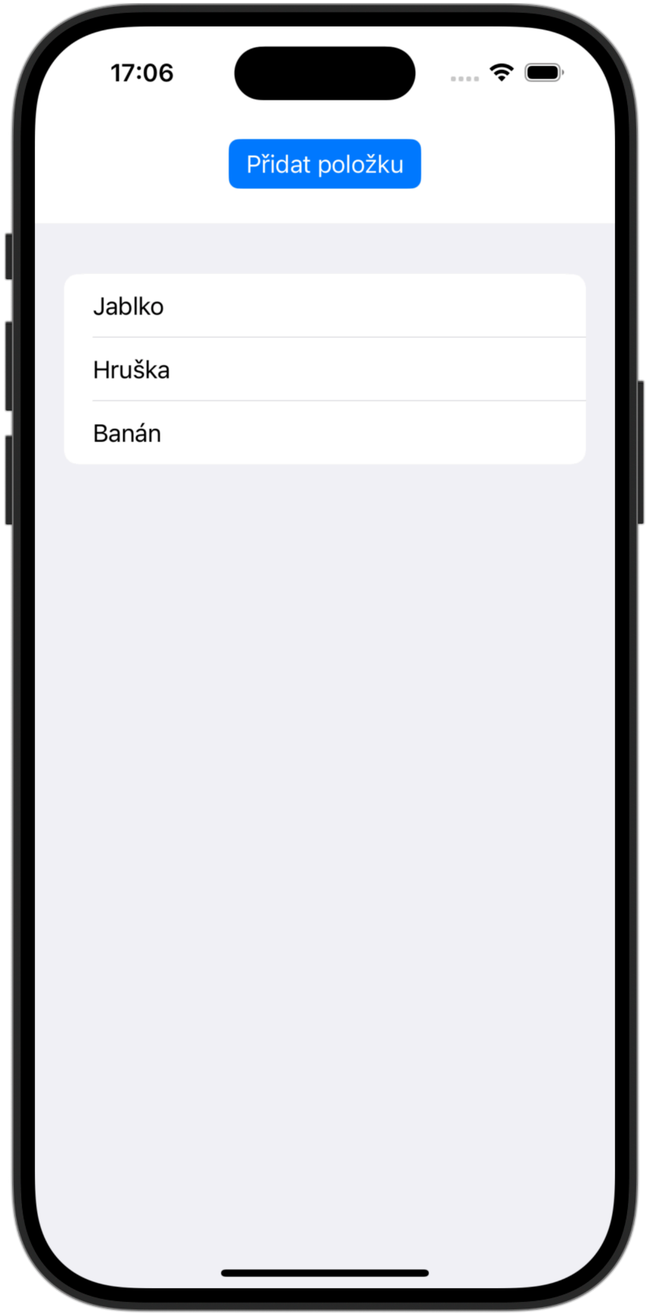
List - dynamická data

```
struct SimpleListView: View {

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

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



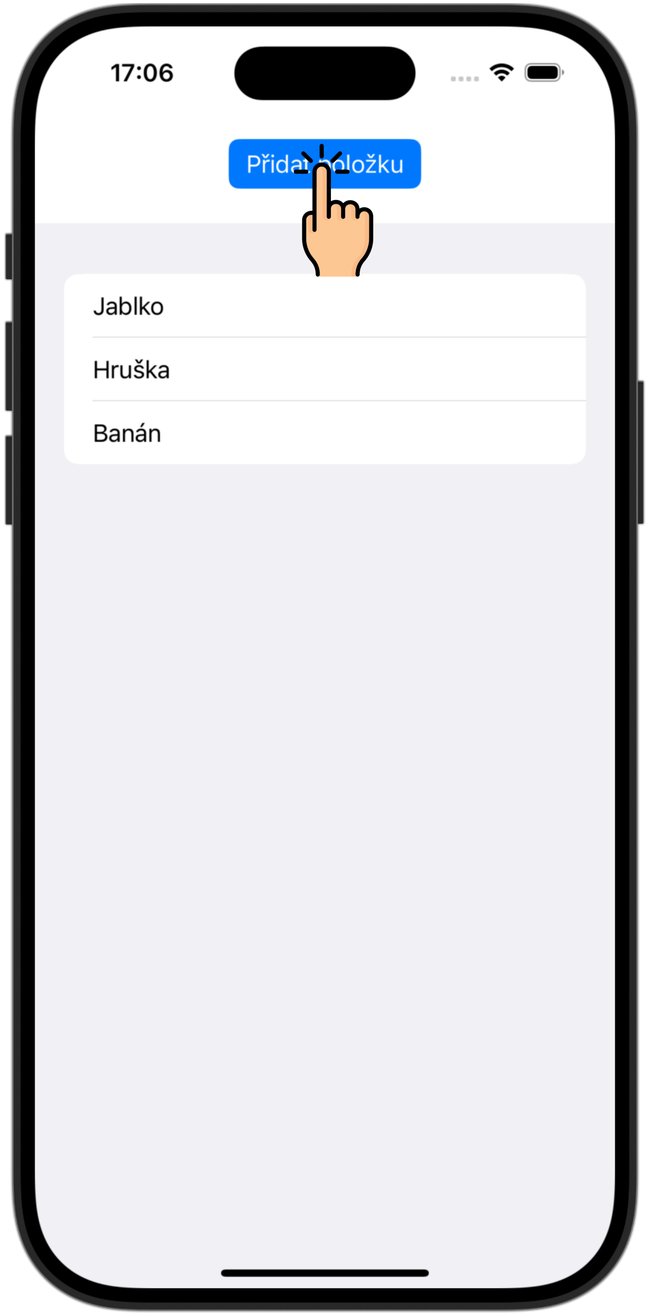
List - přidání položky

```
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)
            }
        }
    }
}
```



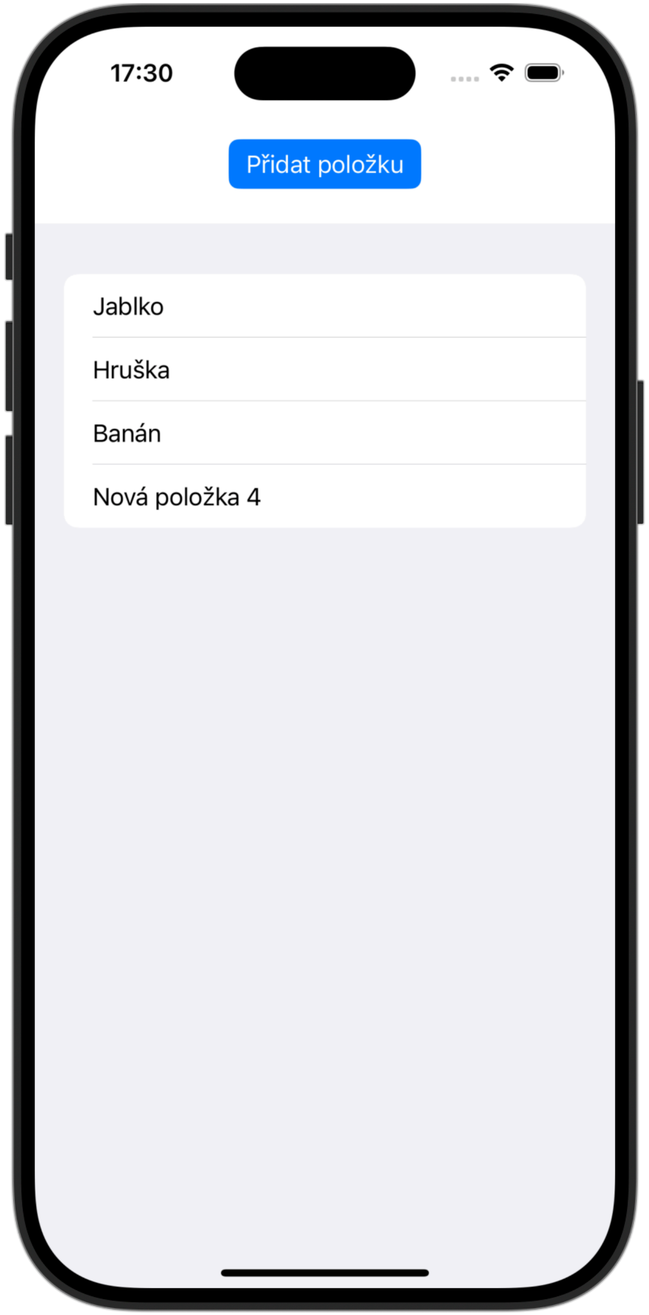
List - přidání položky

```
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)
            }
        }
    }
}
```



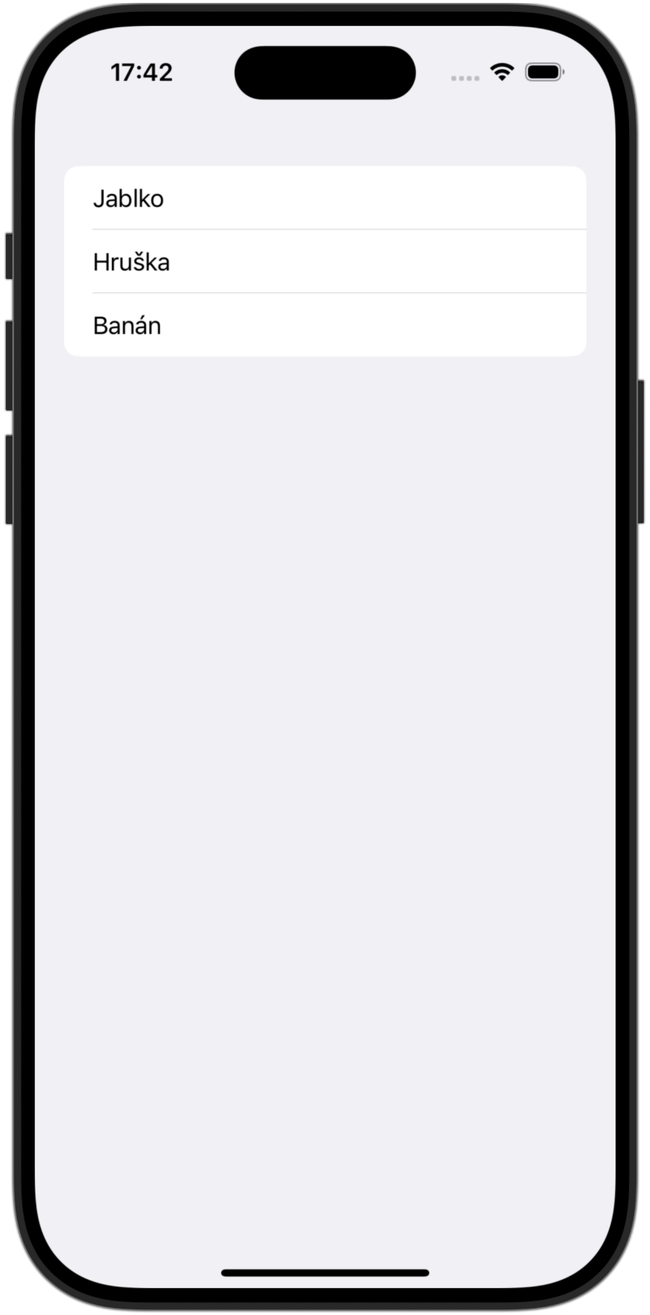
List - přidání položky

```
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)
            }
        }
    }
}
```



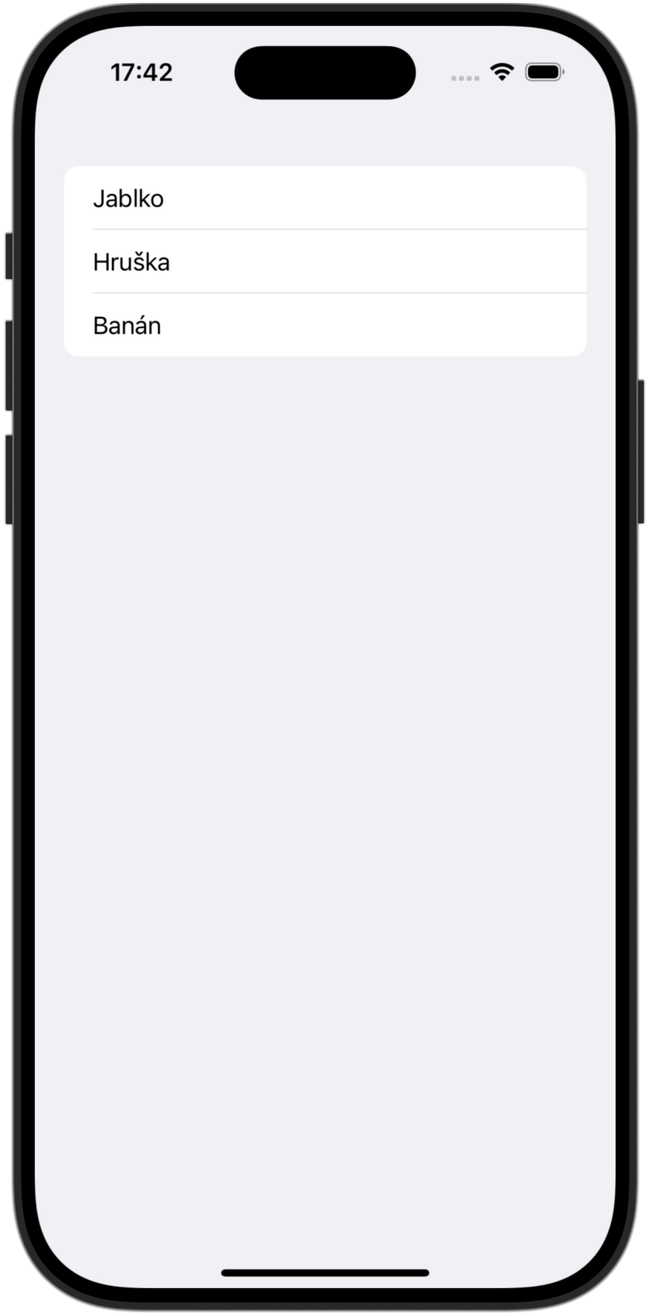
List - smazání položky

```
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)
            }
        }
    }
}
```



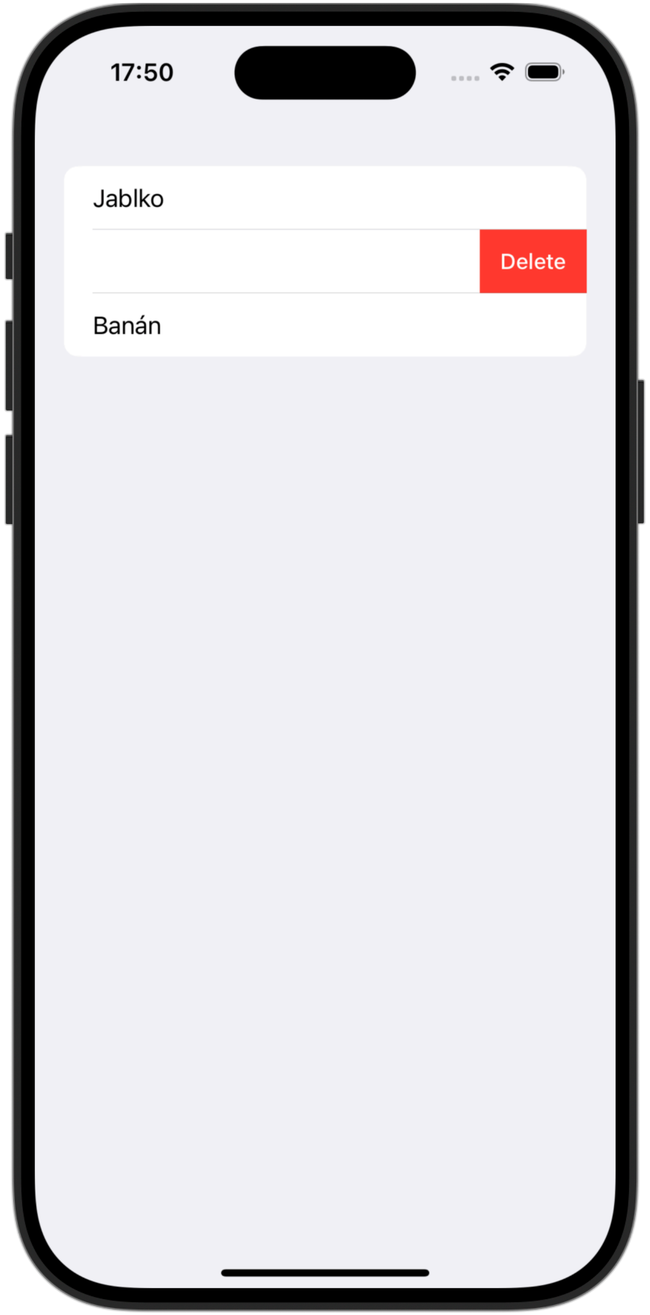
List - smazání položky

```
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)
            }
        }
    }
}
```



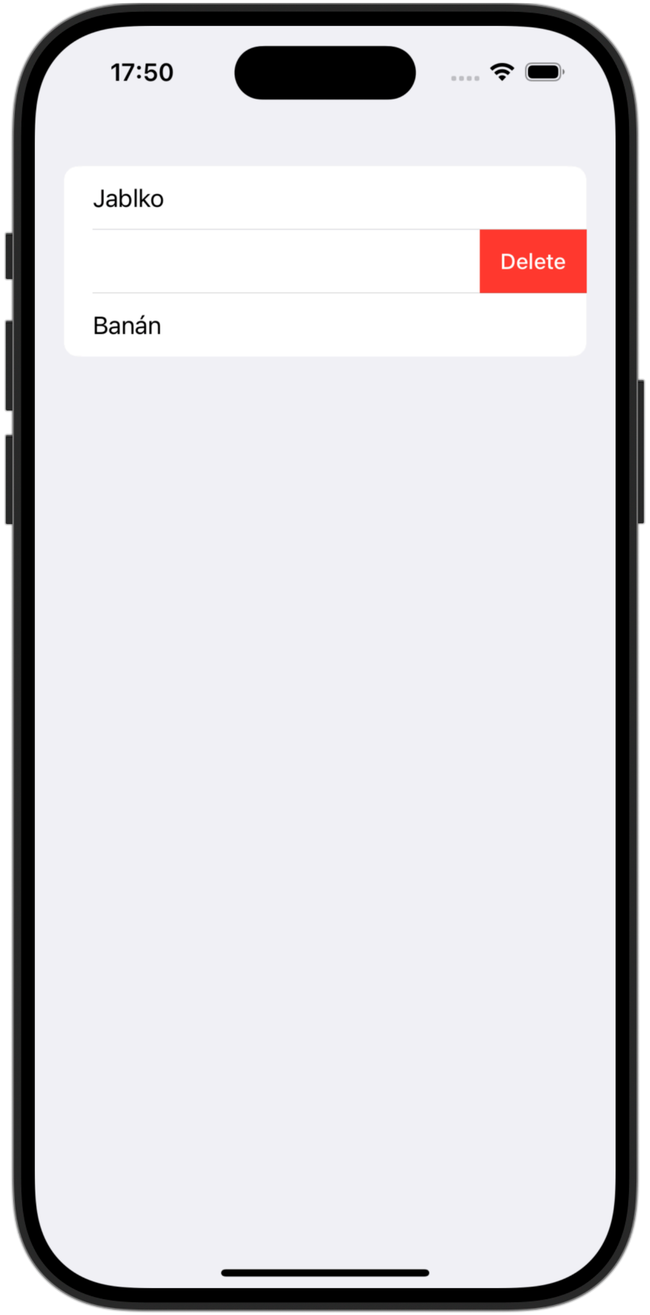
List - smazání položky

```
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)
            }
        }
    }
}
```



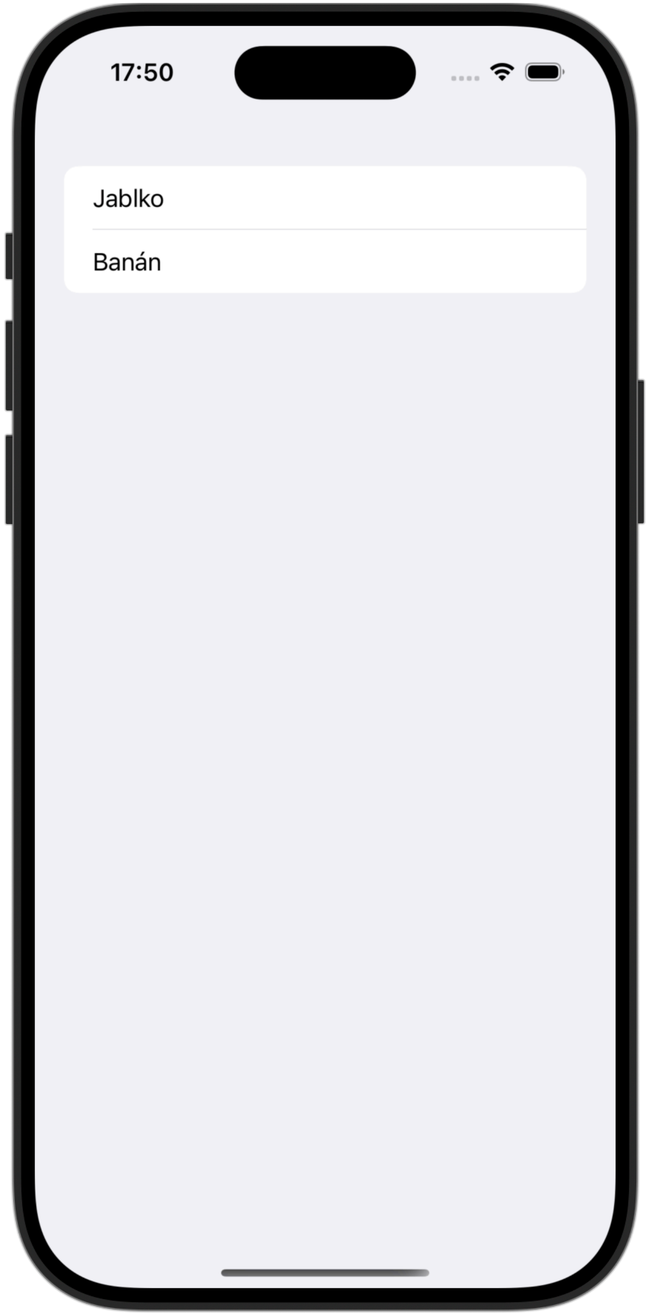
List - smazání položky

```
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)
            }
        }
    }
}
```



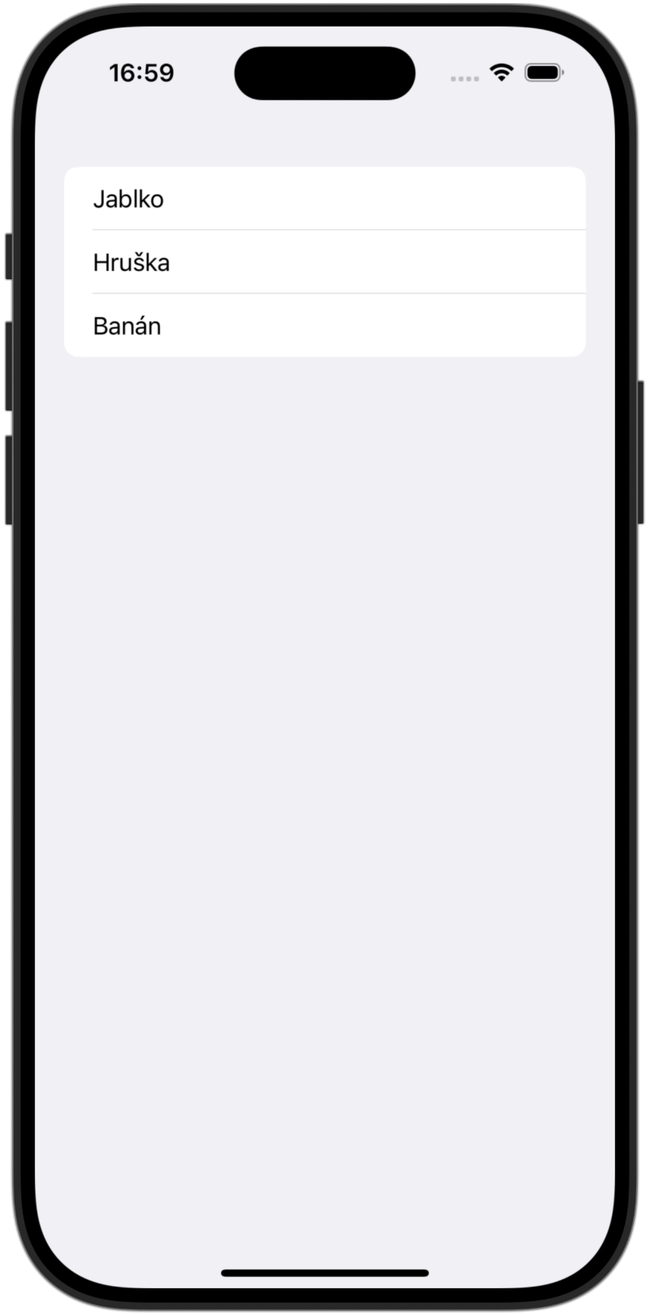
List - smazání položky

```
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.

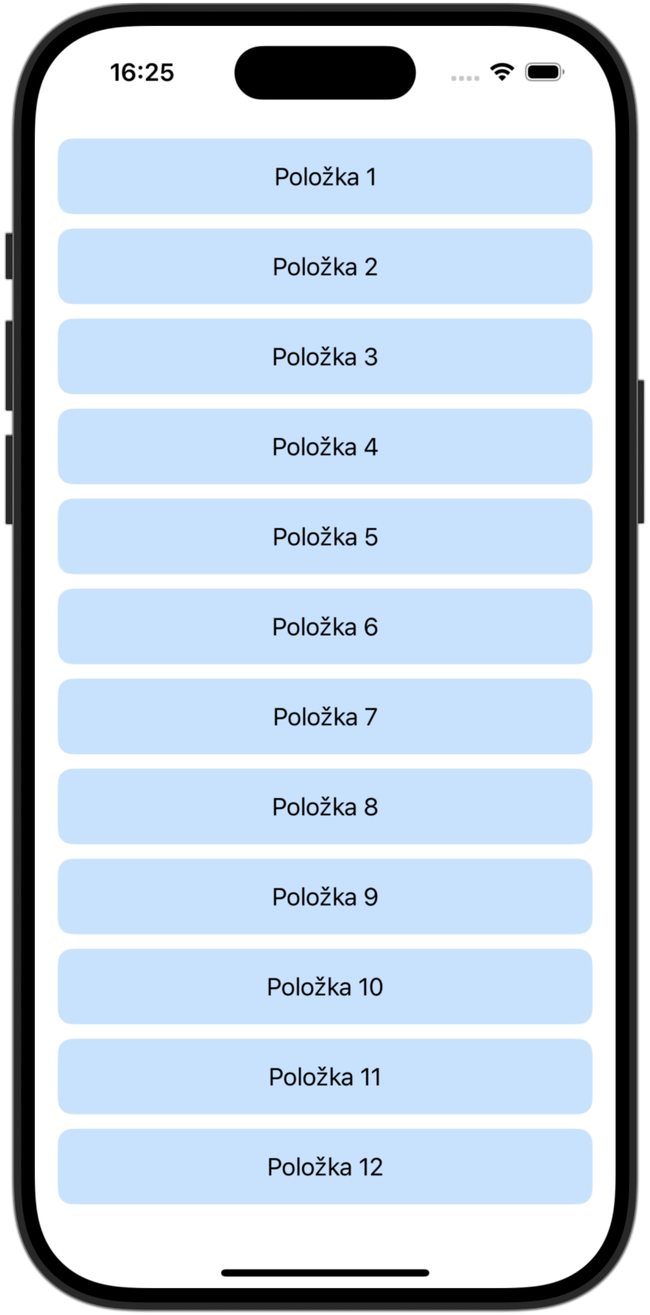
LazyVStack

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()
    }
}
```

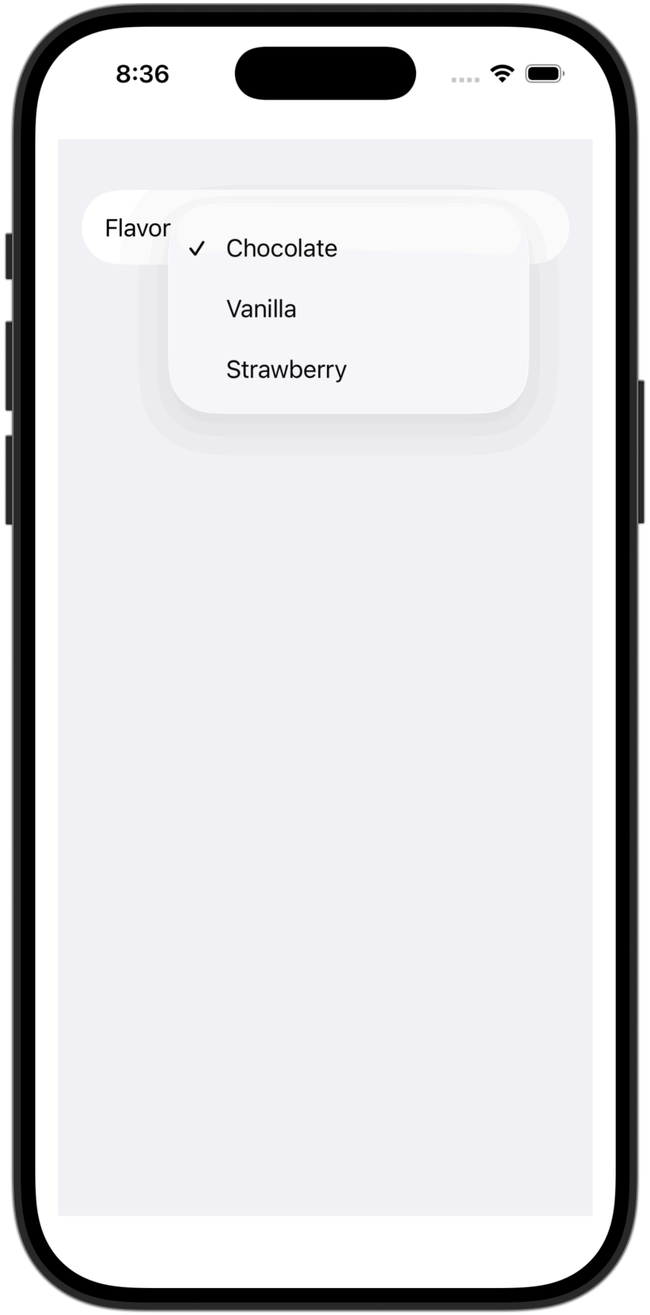


Picker

- Picker slouží k výběru jedné hodnoty z více možností
- Vybraná hodnota je vždy vázána na stav pomocí Binding
- Picker pracuje s hodnotami, nikoliv s indexy

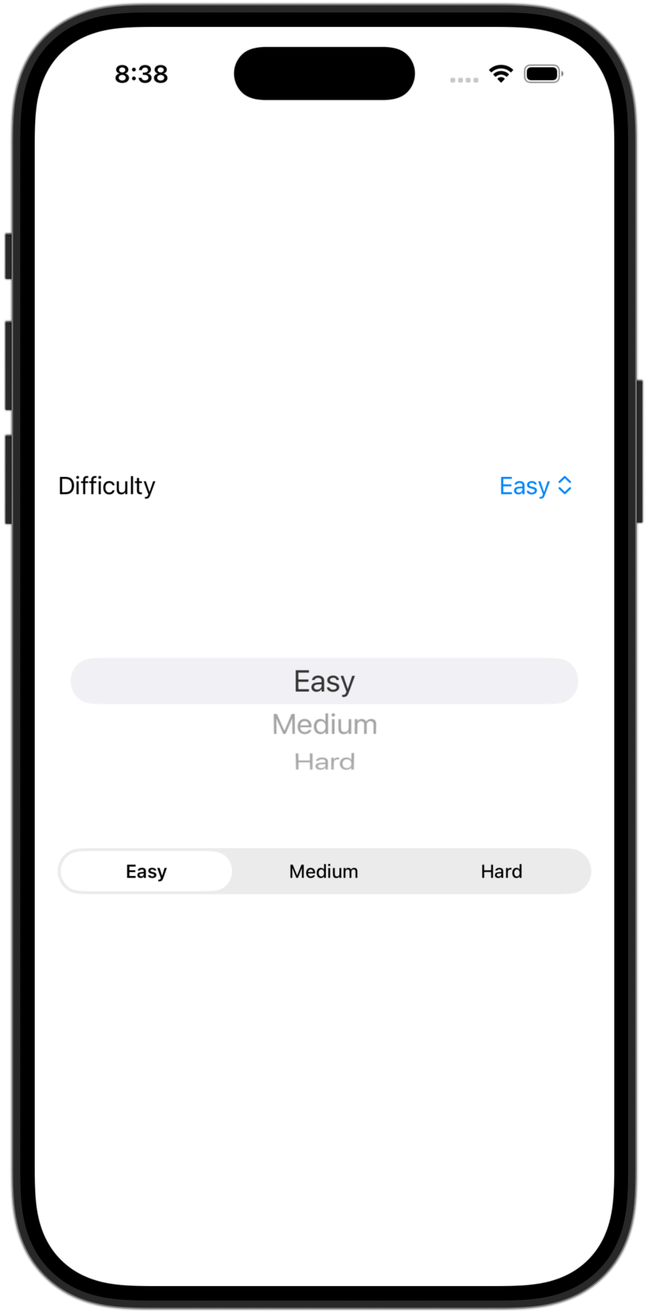
Picker

```
enum Flavor: String, CaseIterable, Identifiable {
    case chocolate, vanilla, strawberry
    var id: Self { self }
}

struct ContentView: View {
    @State private var selectedFlavor: Flavor = .chocolate

    var body: some View {
        List {
            Picker("Flavor", selection: $selectedFlavor) {
                Text("Chocolate").tag(Flavor.chocolate)
                Text("Vanilla").tag(Flavor.vanilla)
                Text("Strawberry").tag(Flavor.strawberry)
            }
        }
    }
}
```



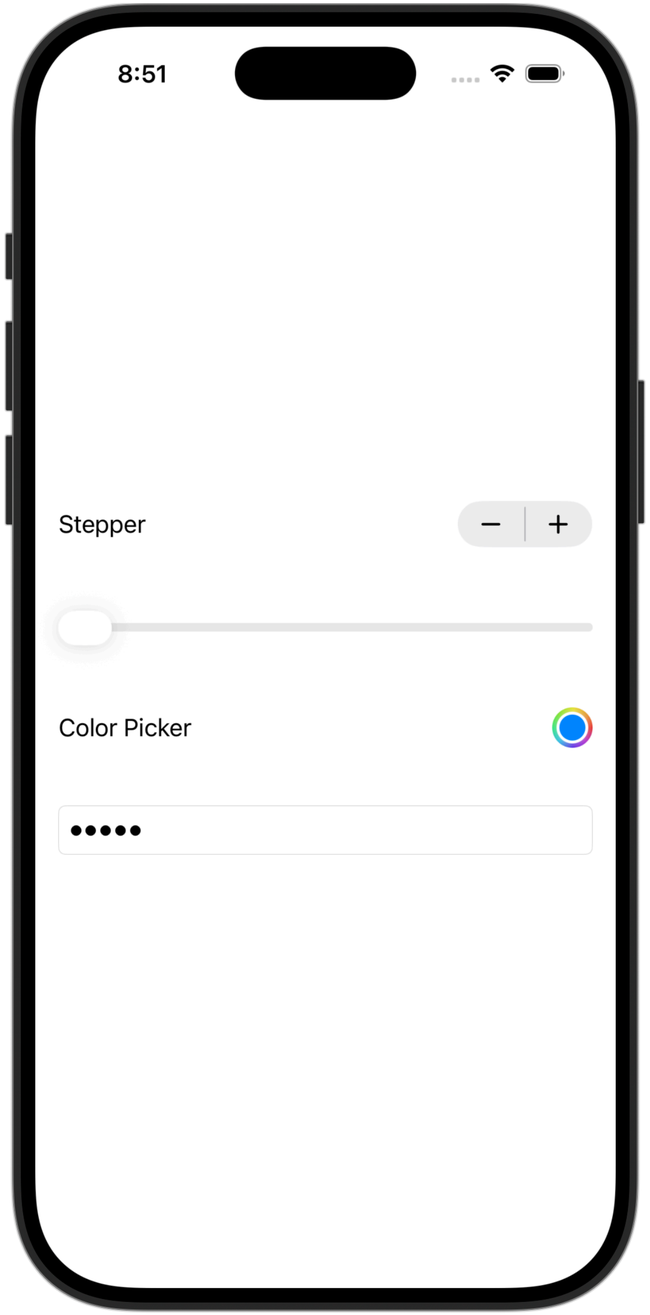
Picker

```
enum Difficulty: String, CaseIterable {
    case easy = "Easy"
    case medium = "Medium"
    case hard = "Hard"
}

struct ContentView: View {
    @State private var difficulty: Difficulty = .easy

    var body: some View {
        VStack {
            HStack {
                Text("Difficulty")
                    .frame(maxWidth: .infinity, alignment: .leading)
                Picker("", selection: $difficulty) {
                    ForEach(Difficulty.allCases, id: \.self) { level in
                        Text(level.rawValue)
                    }
                }
            }

            Picker("Difficulty", selection: $difficulty) {
                ForEach(Difficulty.allCases, id: \.self) { level in
                    Text(level.rawValue)
                }
            }
            .pickerStyle(.wheel)

            Picker("Difficulty", selection: $difficulty) {
                ForEach(Difficulty.allCases, id: \.self) { level in
                    Text(level.rawValue)
                }
            }
            .pickerStyle(.segmented)
        }
    }
}
```



Další komponenty

```
struct ContentView: View {
    @State private var quantity = 0
    @State private var speed = 0.0
    @State private var color: Color = .blue
    @State private var password = "Heslo"

    var body: some View {
        VStack(spacing: 40) {
            Stepper("Stepper ", value: $quantity)

            Slider(
                value: $speed,
                in: 0...100,
                onEditingChanged: { _ in }
            )

            ColorPicker("Color Picker", selection: $color)

            SecureField("Password", text: $password)
                .textFieldStyle(.roundedBorder)
        }
    }
}
```



Úkoly

https://vyjidacek.cz/tmai/seminar4.php