SwiftUI
Combine

Martin Hrubý, FIT VUT, Programování zařízení
Apple, akad. rok 2019/20, #covid19

Fakta

- SwiftUI ohlášeno na WWDC 2019. Wow-effect.
- Určeno pro iOS 13, macOS Catalina.
  - Další změny v AppDelegate. SceneDelegate.
- Stále ”testing”, tj. **SwiftUI není hotovo**!
- Aktuálně lze vývoj v XCode provádět:
  - ve Storyboard stylu,
  - ve SwiftUI stylu.
- Apple označuje **SwiftUI za budoucnost**.

Východiska

- Řízení vnitřního stavu VC a procesů.
  - Kvanta delegátů, přeposílání zpráv, async řízení programu.
- Pracnost programování triviálností.
  - Storyboard, Layout views, DataSource, tabulky….
  - Tuny zbytečného kódu, mnohdy bez koncepce.
- Rozpor s IBOutlet — let/var, UILabel?
- Blbu-vzdorné programování.
  - Daří se to? Objective-C versus Swift. Storyboard versus SwUI.

MVVM, M-V-VM

- MVC, Smalltalk, 70tá léta.
- Presentation model -> MVVM.
- M—Model, V—View, VM—ViewModel.
- VM — ”Model předžvýkaný pro View”.
  - **Drží stav uživatelského rozhraní aplikace**.
  - Řekli bychom, je to interní stav všeho, co jsme dřív nazývali Controller+Views.
- Kde zmizel Controller? Je tam nějaký kód?

MVVM

- View má přístup (referencuje) na VM, ale ne naopak.
  - … tj, VM nezapisuje do V. VM se nezajímá o V.
  - dodejme pro pořádek, že i View má svůj vnitřní stav (Text, *TextField*, Switcher). Řekněme — *mikro-stav*.
- VM má přístup na M, ale ne naopak.
  - … tj, M nereferencuje VM, nezapisuje do VM.
- V nemá přímý přístup na M. Pouze bindings.

Jak vypadá přenos stavu?

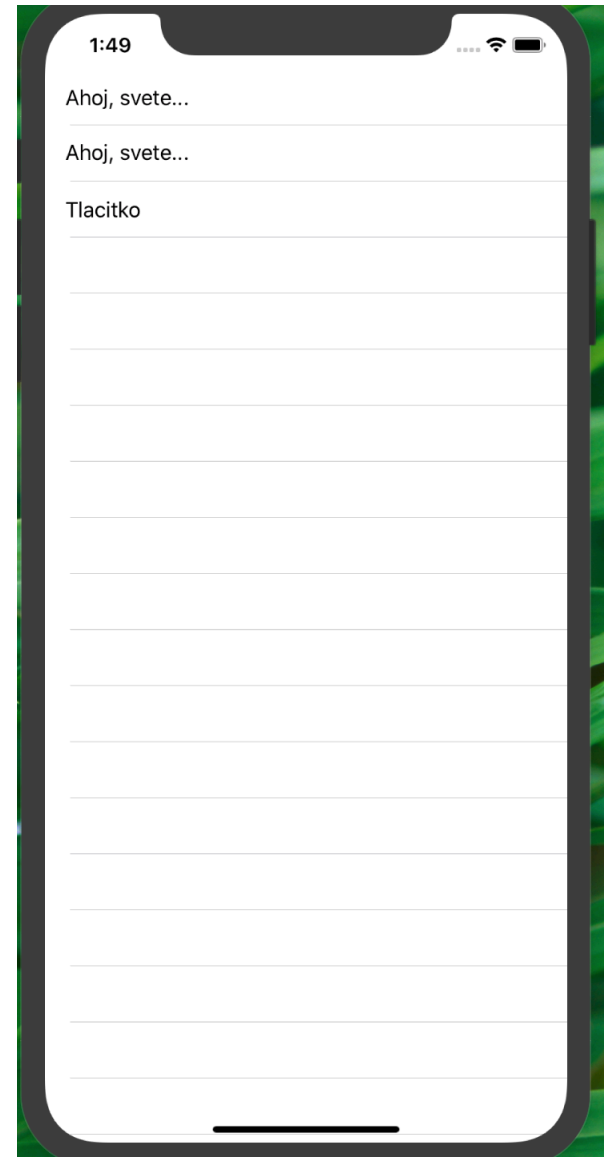
- *VM má přístup na M, ale ne naopak.*
- Binding (KVC/KVO). Implementace *Combine*.
  - Publisher/Subscriber/Subscription.
  - Model něco publikuje. @Published, ObservableObject.
  - VM se napojí (binding). Dostává automaticky zprávy.
- V má přístup na VM…a máme z toho pipeline.
  - Dotažení KVO k dokonalosti. Cocoa v macOS (NSLabel byl observer nějaké keyPath…).

Myšlenka SwiftUI

- *Deklarativní* programování UI/řízení aplikace.
- Oddělení stavu VC a jeho podoby (View).
  - Jasně definovaný stav VM (ViewModel) a jasně definované View. Automatické zrcadlení VM a Views.
  - Stále *programujeme obsluhu událostí* (UI, OS).
  - "View" je *struct*, aby nikoho nenapadlo předávat referenci :)
- Deklarativní návrh procesů (Reaktivní programování, Combine).

Demo

```
// je to struktura! Dusledky...
struct ContentView: View {
    // vnitrni stav view
    @State var obsah = "Ahoj, svete"
    // podoba view
    var body: some View {
        //
        List {
            Text(obsah)
            TextField("zadej",text: $obsah)

            Button(action: { self.akce() }) {
                Text("Tlacitko")
            }
        }
    }
    // neni "mutating"...proc
    func akce() { obsah = "..." }
}
```



Demo, důsledky

- ContentView je struct, má jenom 1 vlastníka.
- Jak to vlastně funguje
  - Je tam vůbec něco *referencovatelného*???
- Když mám precizně formulovaný stav té ”obrazovky”, pak ji mohu *kdykoli rekonstruovat*.
  - V je důsledkem VM, VM je důsledkem M.
  - hodnotu ContentView mohu kdykoli zahodit, znovu instanciovat a dostanu teoreticky to samé!

Zasazení SwiftUI do aplikace

- SceneDelegate.
  - Instancuje *UIWindow* (známe, rootViewController).
  - Instancuje počáteční SwiftUI-View.
  - Instancuje UIHostingController(swiftui-view) -> rootVC pro UIWindow.
  - makeKeyAndVisible()
- Scene -> SceneDelegate -> UIWindow -> rootViewController -> SwiftUI-View

View, protokol

- struct Moje: *View* {}, *View* je protokol.
- Vnitřní stav struktury:
  - let/var instanční proměnné — ty stále smím mít,
  - @State, @Environment, @ObservedObject
- Instanční metody.
- var body: *some View* {} — vypočtená property.
- Inicializátory *init(…)*.

Property wrapper

- @wrapper(argumenty) var jmeno : Type = initValue
- Vznikne instanční proměnná ”jmeno” typu ”wrapper”, která zapouzdřuje hodnotu Type.
- … může to být:
  - struct,
  - *class* — pak lze referencovat.

Konvenční model @State

- @MyState var jmeno: String = ”petr…”
  - _jmeno je objekt typu MyState<String>; @property v Obj-C.

```
//
@propertyWrapper class MyState<Element> {
    //
    fileprivate var hidden: Element
    fileprivate var delegate: MyStateDelegate?
    //
    var wrappedValue: Element {
        //
        get { hidden }
        set {
            //
            hidden = newValue;
            // posli zpravu majiteli, ze doslo ke zmene
            // delegate?()
        }
    }
}
```

some View

- Tzv. opaque type (zamlžený,rozmazaný, skrytý).
- func cosi() -> *View* { return Text(”Ahoj”) }
- func cosi() -> *some View* {}

```
// protokol View.
public protocol View {
    // obsahuje asoc. typ "Body", coz musi byt nekdo,
    // kdo implementuje protokol View ;)
    associatedtype Body : View
    // a dale implementuje vypoctenou property
    // s vyslednym typem Body
    var body: Self.Body { get }
}
```

some…

- Generické typy: struct, class.
- Protokoly neumožňují přímo šablonovat, ale zavádí asociované typy, které musí implementátor specifikovat.
- *protocol Hello {}*
- *var p: Hello;* je někdo, kdo implementuje Hello.
- *func giveme() -> Hello;*

tohle je košer…

```
//
protocol Hello {
    //
    func sayHello();
}
//
struct H1 : Hello {
    //
    func sayHello() {
        //
    }
}
//
func someone() -> Hello {
    //
    return H1()
}
```

není košer…

```
// nejaky protokol
protocol Hello {
    // ktery si vsak vymini asoc.typ, ktery je...
    associatedtype Element: CustomStringConvertible
    //
    var sayHello: Element { get }
}
// korektni implementace protokolu
struct H1 : Hello {
    // skoro nadbytecne rikat:
    // typealias Element = String
    //
    var sayHello: String { "Hello-String" }
}
// error, Hello neni konkretni typ
func someone() -> Hello {
    // nebo: if (…) return H1(); else H2()
    return H1()
}
//
let p = someone()
```

je košer…

```
// some Hello je opaque typ, je to
// propustka/benevolence/jaktonazvat prekladace
func someone() -> some Hello {
    //
    return H1()
}
// drzim nekoho, kdo implementuje Hello
// a NIC VIC dalsiho o nem behem PREKLADU nevim
// vcetne toho, co je jeho Element
let p = someone()
// ...
print(p.sayHello.description)
// v runtime uz nejakou (dynamicky) odpoved dostaanu
print(type(of: p.sayHello))
```

ještě jednou…

```
//
struct Experiment: Hello {
    //
    typealias Element = Int
    //
    var value: Int
    var sayHello: Int { 12345}
    //
    static func experiment() -> some Hello { return Experiment(value: 3) }
}
// p2 je typu "some Hello" a nic vic
// behem prekladu nevim
let p2 = Experiment.experiment()
// p2 je konkretni hodnota, u niz neznam konkretni obsah
print(p2.sayHello.description)
// tj. nelze
// p2.sayHello + 3 // nelze
// print(p2.value)
```

struktury…

- SwiftUI by vypadlo jinak, kdyby bylo postavené na třídách (*final* class XX: View).
- *struct ContentView: View {}*
  - Pokud vám chybí MVC, nazývejme to ViewController (je tam model, jsou tam views, jsou tam akce…). *Složený View*.
  - Referencovatelnost není.
  - Dědičnost není — až na extensions nad protokoly.

Úvod do komponent…

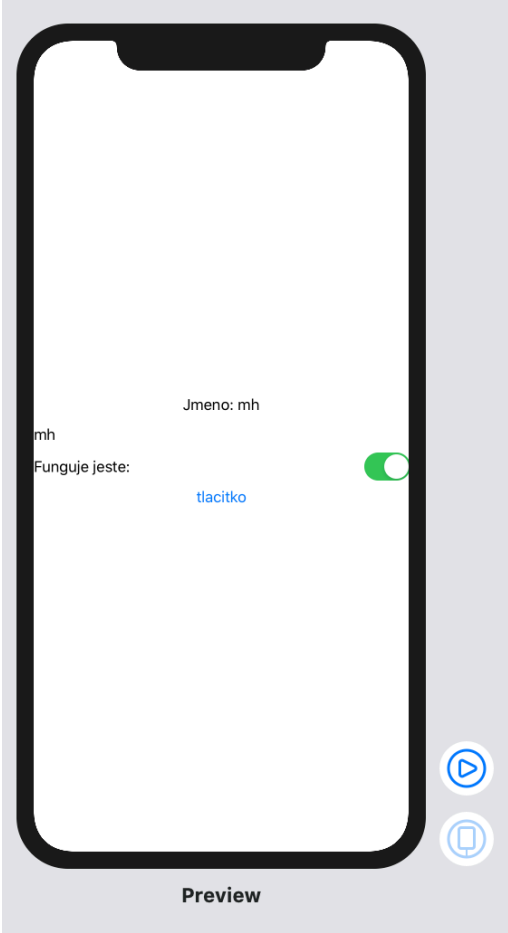
- Chceme sestrojit UI s nějakým obsahem.
- Tvoříme struct:View a ty komponujeme.
- Uvidíme syntaxi Swiftu mírně pokřivenou systémem ”builderů”.
- Připomeňme: List { cosi } je List.init(blk: ()->())
- Maximálně ”beze-stavové” (veškerý stav objektů je vytěsněn ven).

Komponenty SwiftUI

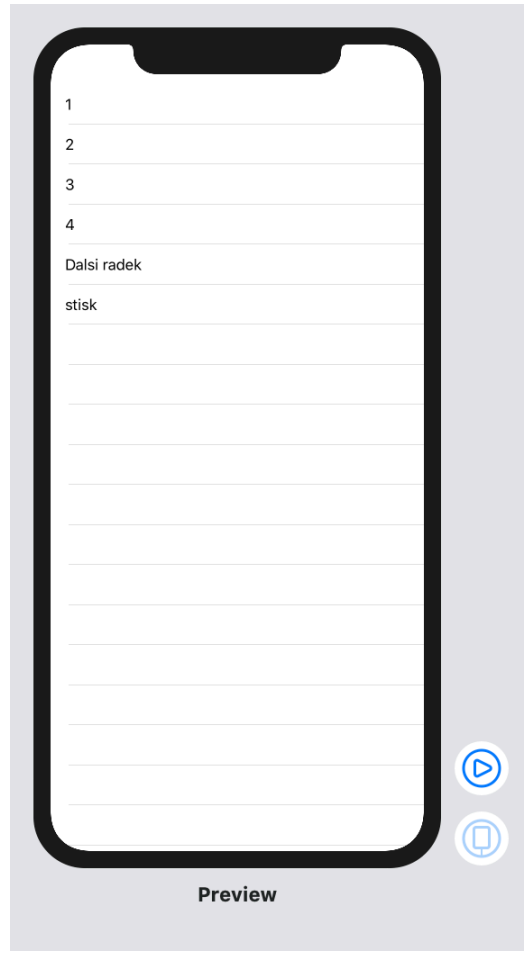
- *Text* — UILabel. *TextField* — UITextField.
- *Toggle* — UISwitch.
- *Button* — UIBarButtonItem, UIButton, …
- *VStack*, *HStack*, *Spacer*.
- List/Form. Section — UITableView.
- NavigationView, TabBar.
- Geometrii Views a layout neřeším. Topologie 😍

Demo

```
//
struct ContentView: View {
    //
    @State var jmeno = "mh"
    @State var funguje = true
    //
    var body: some View {
        //
        VStack {
            //
            Text("Jmeno: \(jmeno)”);
            TextField("jmeno", text: $jmeno)
            Toggle(isOn: $funguje) { Text("Funguje jeste: ")}
            Button(action: { self.akce() }, label: { Text("tlacitko") })
        }
    }
    //
    func akce() { }
}
```



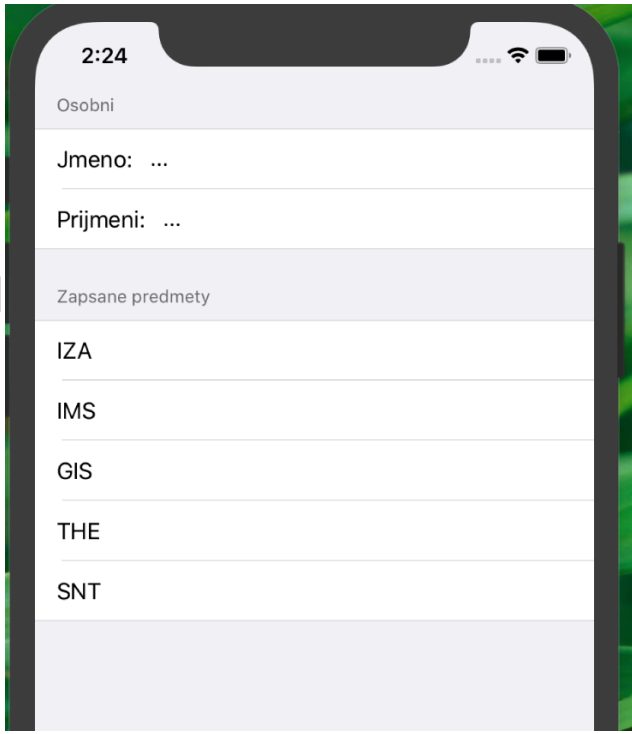
Demo: tabulka #1




```
// je to struktura! Dusledky...
struct ContentView: View {
    // podoba view
    var body: some View {
        // chci tabulku
        List {
            // 4 radky obsahu
            ForEach([1,2,3,4], id:\.self) { i in Text("\(i)")}
            // 5. radek
            Text("Dalsi radek")
            // 6. radek
            Button(action: {}, label: {Text("stisk")})
        }
    }
}
```

Demo: tabulka #2

```
struct ContentView: View {
    //
    @State var jmeno = "..."
    @State var prijmeni = "..."
    @State var predmety = ["IZA", "IMS", "GIS", "THE", "SNT"]
    // podoba view
    var body: some View {
        // formular (UITableView, style Groupped)
        Form {
            //
            Section(header: Text("Osobni")) {
                //
                List {
                    //
                    HStack { Text("Jmeno: "); TextField("jmeno", text: $jmeno)}
                    HStack { Text("Prijmeni: "); TextField("jmeno", text: $prijmeni)}
                }
            }
            //
            Section(header: Text("Zapsane predmety")) {
                //
                List(predmety, id: \.self) { i in Text(i) }
            }
        }
    }
}
```



Znovu-použitelnost Views…

- Views lze komponovat z menších, standardizovaných uživatelských Views.
- Strukturované programování (procedury).
  - SwiftUI ruší InterfaceBuilder/StoryBoard a vrací programování UI do zdrojového kódu. **UI vytváříme kódem ve Swiftu, dělejme to PROSÍM strukturovaně.**
- Strukturované Views.

• • •

```
// Prototyp TableViewCell
struct TextTableCell : View {
    // pamatuju si vstup
    let left: String
    let right: String
    // ...
    init(_ l:String, _ r: String) {
        left = l; right = r;
    }
    // toto lze kdykoli zrekonstruovat
    var body: some View {
        //
        HStack{ Text(left); Spacer(); Text(right) }
    }
}
//
struct MyTable: View {
    //
    var body: some View {
        //
        List {
            //
            TextTableCell("jmeno", "mh")
            TextTableCell("narozen", "nedavno...")
        }
    }
}
```

...

```
//
func textTableCell(_ l:String, _ r: String) -> some View {
    // [return]
    HStack{ Text(l); Spacer(); Text(r) }
}
//
struct MyTable2: View {
    //
    var body: some View {
        //
        List {
            //
            textTableCell("jmeno", "mh")
            textTableCell("narozen", "nedavno...")
        }
    }
}
```

Co je stav ”View”

- Máme vnitřní stav struktury View — *ViewModel*.
- Viditelná podoba (View) je kdykoli konstruovatelná z ”var body”.
- Body se s každou změnou stavu generuje znova (volá se metoda),
  - optimalizace…
- Zaměřme se na programování toho *stavu (VM)*.

Magické propojky @

- @State, @Binding, @Environment — V / VM.
- @ObservedObject — V / VM.
- protocol ObservableObject
- @Published — M.

Propojky Modelu

- @Published — publikuji tuto property.
  - při aktualizaci obsahu jsou ”čtenáři” notifikováni. Observable.
  - pouze třídy mohou mít published property.
  - Model by měl být tvořen třídami (v hlavních rysech…).
- *ObservableObject* — publikuji celý objekt.
  - při aktualizaci nějaké property objektu…
- ObservableObject třída obsahující @Published properties je Model.

Propojky ViewModel—View

- @State — je stavová proměnná ViewModelu. View reaguje na její aktualizaci (překreslením).
- @Binding *X* — View A předává B vazbu na svoji @State *Y* proměnnou:
  - X a Y — jsou ”read-write” stavové proměnné,
  - změny do X se kopírují do Y a naopak,
  - se všemi důsledky.

Demo @State

```
//
struct CopyText : View {
    // viewModel pro CopyText
    @State var textForEdit: String = "neco"
    //
    var body: some View {
        //
        List {
            // cte stavovou promennou
            Text(textForEdit);
            // zapisuje...
            TextField("", text: $textForEdit)
        }
    }
}
```

Náhled implementace @State

```
// Predstava o implementaci @State
@propertyWrapper struct State<Value> {
    // komu hlasim zmeny ViewModelu
    // neni to ta View, je to nekdo nad ni
    // (nejaky ViewController ;))
    // e.g. UIHostingController
    let delegate: NejakyReferencovatelny
    //
    var wrappedValue: Value {
        // zmeny se musi hlasit
        didSet { delegate.hlaseni() }
    }
}
```

@Binding; struct Binding<Value>

```
// Prototyp TableViewCell
struct TextEditTableCell : View {
    // pamatuju si vstup
    let left: String
    @Binding var right: String
    // toto lze kdykoli zrekonstruovat
    var body: some View {
        //
        HStack{ Text(left); Spacer(); TextField(left, text: $right) }
    }
}
//
struct ContentView: View {
    //
    @State var jmeno = "mh"
    @State var narozen = "nedavno..."
    //
    var body: some View {
        //
        List {
            // exportovanim @State se vytvoří @Binding
            TextEditTableCell(left: "Jmeno", right: $jmeno)
        }
    }
}
```

Operátor $

```
// Predstava o implementaci @State
@propertyWrapper struct State<Value> {
    // ...
    // ...
    var wrappedValue: Value
    // operator "$" vola tuto property
    var projectedValue: Binding<Value> {
        // self is immutable...
        return Binding<Value>(get: { self.wrappedValue }, set:
{ self.wrappedValue = $0 })
    }
}
```

IBActions ve strukturách View

- Chceme do Views dostat nějaký akční kód.
- Funkce (instanční, třídní/statické).
- Inicializátor.
- .onXXX — události stylu ViewWillAppear.
- *.sheet* — vyskakování dalších Views modálně.
- Akce na stisknutí tlačítek.

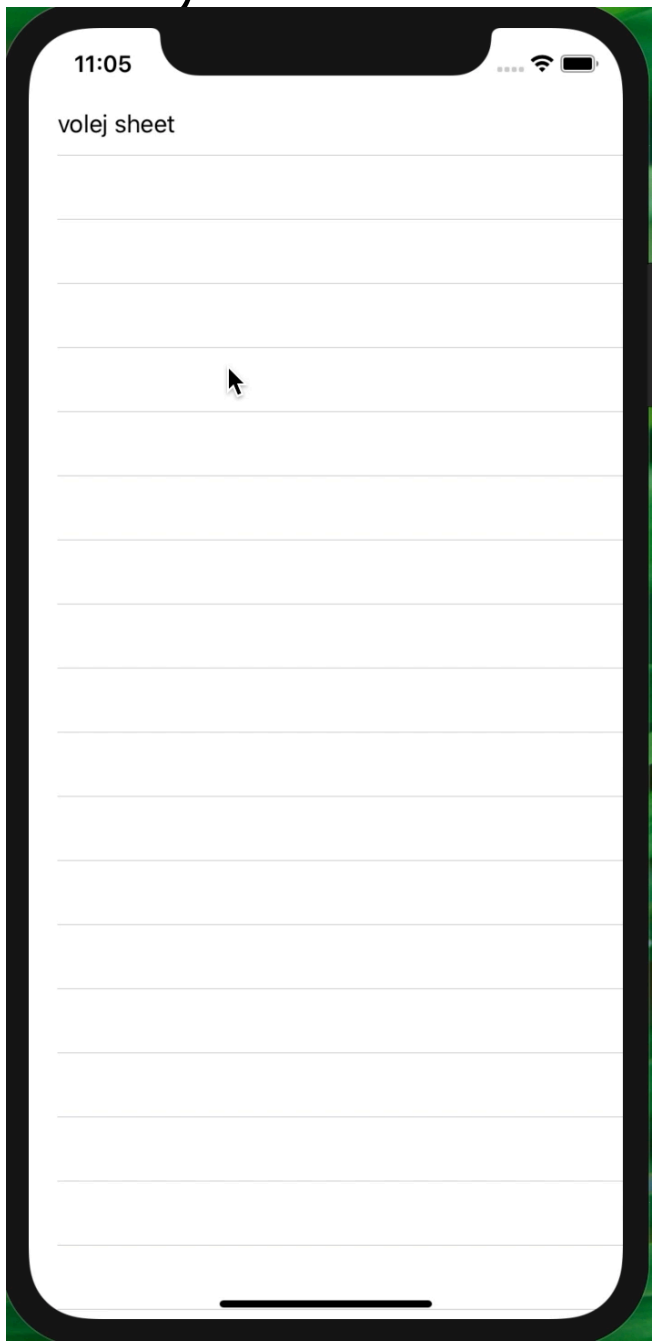
Komplexní demo…

```
// Model v aplikaci. Je zapouzdreno do tridy
class MojeData: ObservableObject {
    // resp, do singletonu
    static let shared = MojeData()
    // datovy obsah, musi/nemusi byt @Published
    @Published var poznamky = [String]()
    // ...
    func add(_ s: String) { poznamky.append(s) }
}
```

```swift
struct ContentView: View {
    // stavam se observerem celeho objektu
    @ObservedObject var model = MojeData.shared
    // stav V rikajici, zda-li je "sheet" viditelny
    // !!!
    @State var sheetON: Bool = false
    //
    var body: some View {
        //
        List {
            // z pole generuju tabulku
            ForEach(model.poznamky, id:\.self) { poz in
                Text(poz)
            }
            // tlacitko, sheetON:false -> sheetON:true
            Button(action: { self.sheetON.toggle() }) { Text("volej sheet")}
        }
        // deklaruji (!!!), ze ma byt viditelny sheet, pokud
        .sheet(isPresented: $sheetON) {
            // vytvor sheet a propoj
            Detail(sheetON: self.$sheetON, mojeData: self.model)
        }
    }
}
```

```swift
// modalne prezentovany View
struct Detail : View {
    // sem edituj nejaky nazev/text poznamky
    @State var textik: String = "neco zadej"
    // binding na ContentView::sheetON napr.
    @Binding var sheetON: Bool
    // vec vkusu: 1) bud dostanu ref, 2) MojeData.shared
    let mojeData: MojeData
    //
    var body: some View {
        //
        List {
            TextField("", text: $textik);
            // @escaping
            // modifikuju Model, modifikuju ViewModel
            Button(action: {
                self.mojeData.add(self.textik)
                self.sheetON.toggle() }
            ) { Text("Pridej") }
        }
    }
}
```

Výsledek



Opakování…

- Viděli jsme napojení jednoho View na druhý:
  - datově: @State->@Binding, @Binding->@Binding
  - datové napojení VM na M, @Published, observable…
  - ”spuštění” jednoho V druhým. Je to spuštění? Deklarativní pohled na věc: deklaruju, že pokud je *sheetON==true*, pak tam má být, jinak ne.
  - změny VM -> dynamika.
- Srovnejte s MVC/Storyboardingem.

Navigace

- Modální prezentaci jsme viděli (.sheet).
- *NavigationView*; UINavigationViewController.
- *NavigationLink*(destination: some View).
  - významově je to tlačítko (jako Button).
- Pozn.: jak pro NavigationView formulovat stav zanoření Views, je to jeho forma ViewModel.
  - dynamika NavView/NavLink se ještě bude měnit.

Navigace, demo

```
// View pro detail (...)
struct Detail : View {
    // model, obsah
    let item: String
    // view pro zobrazeni detailu
    var body: some View { Text("Detail je \(item)")}
}
//
struct ContentView: View {
    //
    let data = ["ahoj", "jak", "se"]
    //
    var body: some View {
        // vytvor UINavigationViewController
        NavigationView {
            // nad UITableViewControllerem
            // s "data" jako dataSource
            List (data, id:\.self) { item in
                // nad kazdou UITableViewCell konstruuj
                // segue do DetailVC
                NavigationLink(destination: Detail(item: item)) { Text(item)}
            }
        }
    }
}
```

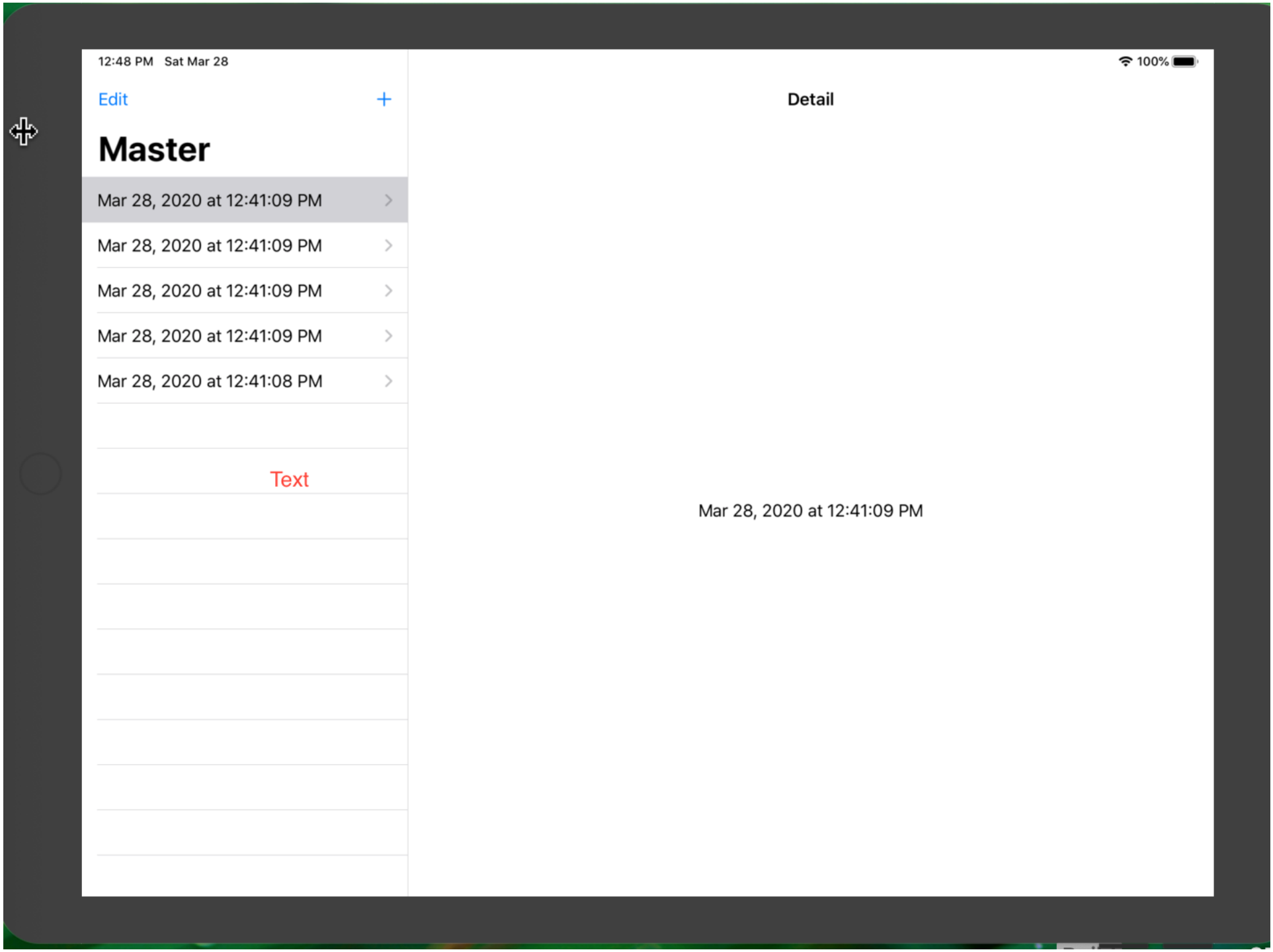
Master-Detail

- Není explicitní UISplitViewController.
  - MasterViewController, DetailViewController.
- Řízení implementuje NavigationView:
  - MasterView(); DetailView();
  - .navigationViewStyle(DoubleColumnNavigationViewStyle())

Master—Detail

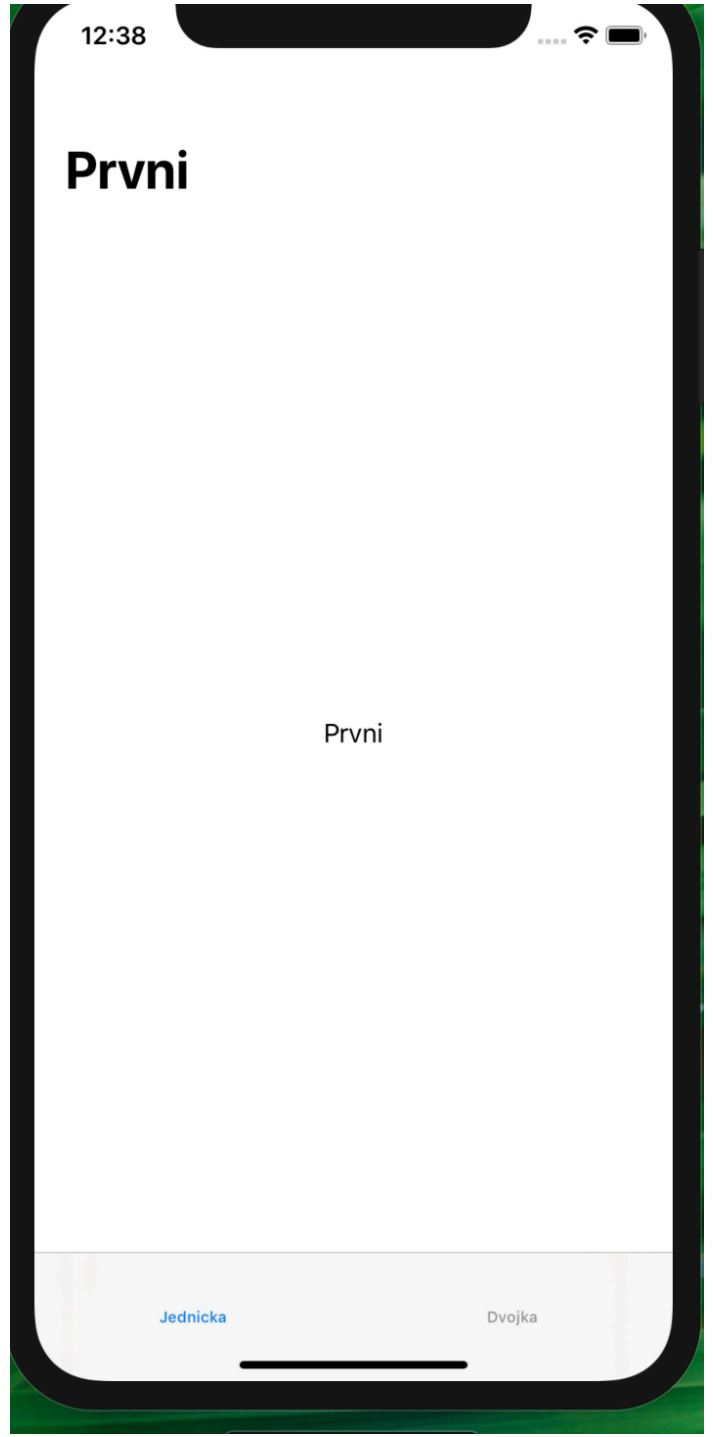
```
struct ContentView: View {
    @State private var dates = [Date]()

    var body: some View {
        NavigationView {
            MasterView(dates: $dates)
                .navigationBarTitle(Text("Master"))
                .navigationBarItems(
                    leading: EditButton(),
                    trailing: Button(
                        action: {
                            withAnimation { self.dates.insert(Date(), at: 0) }
                        }
                    ) {
                        Image(systemName: "plus")
                    }
                )
            DetailView()
        }.navigationViewStyle(DoubleColumnNavigationViewStyle())
    }
}
```

```
struct MasterView: View {
    @Binding var dates: [Date]

    var body: some View {
        List {
            ForEach(dates, id: \.self) { date in
                NavigationLink(
                    destination: DetailView(selectedDate: date)
                ) {
                    Text("\(date, formatter: dateFormatter)")
                }
            }.onDelete { indices in
                indices.forEach { self.dates.remove(at: $0) }
            }
        }
    }
}

struct DetailView: View {
    var selectedDate: Date?

    var body: some View {
        Group {
            if selectedDate != nil {
                Text("\(selectedDate!, formatter: dateFormatter)")
            } else {
                Text("Detail view content goes here")
            }
        }.navigationBarTitle(Text("Detail"))
    }
}
```


12:48 PM Sat Mar 28
100%
Edit
Master
Detail
Mar 28, 2020 at 12:41:09 PM
Mar 28, 2020 at 12:41:09 PM
Mar 28, 2020 at 12:41:09 PM
Mar 28, 2020 at 12:41:09 PM
Mar 28, 2020 at 12:41:08 PM
Text
Mar 28, 2020 at 12:41:09 PM

TabView+NavigationView

```
// Jedna "obrazovka" integrovana do TabView
struct JedenTAB : View {
    //
    var body: some View {
        // navic zapouzdrena do Nav
        NavigationView {
            // ... nejaky obsah
            Text("Prvni")
            // konfigurace Nav
            .navigationBarTitle("Prvni")
        }
        // konfigurace smerem do TabView
        .tabItem { Text("Jednicka") }
        .tag(0)
    }
}
// hlavni "obrazovka" aplikace
struct ContentView: View {
    //
    var body: some View {
        //
        TabView {
            // konstrukce 1. tabu
            JedenTAB()
            // ...
            DvaTAB()
        }
    }
}
```


12:38
Prvni
Prvni
Jednicka
Dvojka

Modifikátory — .xyz(…)

- ViewModifier. Funkcionálně:
  - Představa: SomeView.modif() -> AnotherView
  - Modifikátor přepracuje View a vrací obecně jiný.
- Komu se to vlastně posílá?
  - .navigationBarTitle(”title”)
  - View se buduje ViewBuilderem. Součástí toho jsou modifikátory. Logika (např pro Navigation) v tom na první pohled není.

View & ViewModifier

- I když se to nezdá, SwiftUI je zcela zbudováno v současném Swiftu, což jenom dokládá možnosti metaprogramování ve Swiftu.
- ViewBuilder.
- VStack { V1; V2; V3 } -> VStack([V1,V2,V3])
- ViewModifier: Content -> some View

```swift
//
struct Moje<Content:View> : View {
    //
    fileprivate var holder: () -> Content
    //
    init(@ViewBuilder content: @escaping () -> Content) {
        //
        self.holder = content
    }
    //
    var body: some View {
        //
        VStack {
            //
            Text("Ukazeme si neco...")
            //
            holder()
        }
    }
}
//
struct ContentView: View {
    //
    var body: some View {
        //
        Moje {
            //
            List { Text("Radek1"); Text("Radek2") }
        }
    }
}
```

Polo-programování "body"

```
//
struct ContentView: View {
    @State var isON = false
    //
    var body: some View {
        //
        VStack {
            //
            Toggle(isOn: $isON) { Text("Zapni to") }
            // komponenta je zarazena podminene
            if isON == true {
                //
                Text("Je to zapnuto")
            }
        }
    }
}
```

Bridge na UIKit

- SwiftUI neobsahuje všechny Views z UIKitu.
- Ty lze aspoň napojit.
- CollectionView.

Model pro SwiftUI

- Bude chtít programovat nějaké *chování* VM/M.
- Procesy, které se budou spouštět v reakci na změny VM.
- @IBAction — akce napojené UIViews.
  - Na V je napojeno VM, VM je napojeno na M.
  - Na úrovni Modelu budeme **deklarovat chování**.
  - … a budeme tomu říkat Combine.

Observable, Model

```
// chceme neco referencovatelneho
class MojeData : ObservableObject {
    // tady vznika "Publisher" nad hodnotou
    @Published var jmeno = "mh"
    @Published var heslo = "cosi"
}
//
struct ContentView: View {
    //
    @ObservedObject var model = MojeData()
    //
    var body: some View {
        //
        List {
            //
            TextField("jmeno", text: $model.jmeno)
            TextField("heslo", text: $model.heslo)
            Text(model.jmeno)
        }
    }
}
```

Combine

Úvod do Combine

- Future—Promise:
  - Chci asynchronně zahájit výpočet hodnoty a dostat zprávu, až se hodnota zjistí. Co když se to řetězí?
- Reaktivní programování.
  - Zřetězení asynchronních událostí. K čemu to může být dobré?
- Publisher—Subject—Subscriber:
  - Chci se napojit na zdroj dat ”Publisher” a odebírat výsledky.
  - Tok dat. Tok událostí.

Notifikační centrum

- Přihlásím se jako odběratel typu zprávy.
- Pokud někdo do NC pošle zprávu, jsou upozorněni (sync zprávy) její odběratelé (observers, followers, …).
- Zpracovat zprávu sync/async vůči odesílateli.
- Poslat zprávu zadaným vláknem.
- Nezkoumám, kdo je mým observerem.

Publisher

- Atomizované NotificationCenter.
- Objekt, který rozesílá zprávu (hodnotu) odběratelům (subscribers).
  - Typicky sync. Můžeme ovlivnit vlákno (Scheduler).
- Formálně je to každý, kdo implementuje protokol *Publisher*. Abstraktní pojem.
- *AnyPublisher*<Output, Failure> wrapper.

Pipe, roura, handshake

- Bude nám to připomínat pojem *pipe* z operačních systémů.
- Handshake — příjemce si odběr může řídit:
  - jsem připraven přijímat,
  - pošli další data,
  - končím…

Subject

- *Subject* je *Publisher*, do kterého lze ”poslat hodnotu”.
  - metoda send(value: Output).
- *PassthroughSubject*
  - vstup kopíruje odběratelům. NotificationCenter.
- *CurrentValueSubject*
  - navíc, nově příchozímu odběrateli pošle i poslední známou hodnotu.

Řetězení publisherů

- Odběratelem publisheru A se může stát jiný publisher B.
  - Proč? Transformace hodnoty. Řešení "failures".
- Operátory — reaktivní programování.
  - P1 -> P2 -> P3 -> … -> S1
  - Libovolná síť napojených P-S. Logicky, případný Subject musí stát na počátku.
  - P, P.map { $0 }, encode/decode, univerzální transformace, **znovupoužitelnost operací**.

Subscribers

- Tady řetěz publisherů končí…
- Subscriber:
  - *Assign* — obdrženou hodnotu umístí na zadanou *keyPath* (opět KVC, modernizovaná syntaxe, *Trida.attr*, *.attr*).
  - reference[keyPath:zadanaKeyPath] = … { get set }
  - *Sink* — closure, (Output) -> ()

Vlastnictví…jako vždy esenciální

- Stanete se subscriber, vlastní vás ten publisher?
  - Ukládá subscription, drží referenci. Subscriber si drží svého P.
  - Jak se ukončí subscription. *cancel()*
  - *let p=publisher; p.map { $0*2}; let p2=p.map { $0*2}*
- AnyCancellable
  - Registrací Subscribera vzniká AnyCancellable.
  - publisher.sink { neco } -> AnyCancellable, uložit.
  - Nedává smysl registrovat P/S, ale nereferencovat ho.

uložit AnyCancellable

```
// AnyPublisher<Int,Never>
let base = Just(3)
let kopirka = CurrentValueSubject<Int, Never>(3)
//
class Muj : ObservableObject {
    //
    let kopirkaSink: AnyCancellable
    var storage = Set<AnyCancellable>()
    //
    var lastValueKopirka: Int = 0
    //
    init() {
        // vznika hodnota AnyCancellable reprezentujici
        // Sink subscription
        // neni ulozena, deinit, cancel()
        kopirka.sink { print($0) }
        // registruj sink a jeho ref uloz
        kopirkaSink = kopirka.sink { val in print("Hodnota \(val)") }
        // registruj self.lastValueKopirka=, ref uloz
        kopirka.assign(to: \.lastValueKopirka, on: self).store(in: &storage)
    }
}
```

Aplikace

- Combine na úrovni Modelu aplikace:
  - zpracování dat, procesů, síťových operací.
  - URLSession.
- … na úrovni ViewModelu:
  - události z View.

URLSession

```swift
// https://theswiftdev.com/urlsession-and-the-combine-framework/
struct Post: Codable {
    let id: Int
    let title: String
    let body: String
    let userId: Int
}
//
let url = URL(string: "https://jsonplaceholder.typicode.com/posts")!
//
let reqp = URLSession.shared.dataTaskPublisher(for: url)
    .map { $0.data }
    .decode(type: [Post].self, decoder: JSONDecoder())
    .replaceError(with: [])
    .eraseToAnyPublisher()
    .sink(receiveValue: { posts in
        print(posts.count)
    })
```

Znovupoužitelnost

```
func netLoader<T:Decodable>(Record: T.Type, url: URL) -> AnyCancellable
{
    // [return]
    URLSession.shared.dataTaskPublisher(for: url)
        .map { $0.data }
        .decode(type: [T].self, decoder: JSONDecoder())
        .replaceError(with: [])
        .eraseToAnyPublisher()
        .sink(receiveValue: { posts in print(posts.count) })
}
```

CaseStudy

- Inspirace: https://peterfriese.dev/swift-combine-love/
- Obrazovka se zadáním:
  - uživatelského jména
  - hesla poprvé, podruhé
  - testováním platnosti jména, hesel
  - podmíněné tlačítko ”pokračovat”

AnyPublisher<Output, Failure>

- Základem všeho je nějaký *publisher*.
  - Je to hodnota (struct). Potřebujeme ji nějak *vlastnit*. let/var.
  - struct — dává smysl, aby byla vlastněna pouze jednou.
- Napojení na publisher:
  - Za účelem odebírat jeho hodnotu (sink), ukládat (assign).
  - … transformovat jeho hodnotu — navázat vytvořením dalšího AnyPublisher. P1->P2->P3-> { odběratelé }
  - Všechny je třeba nějak referencovat. *AnyCancellable*.

@Published je publisher

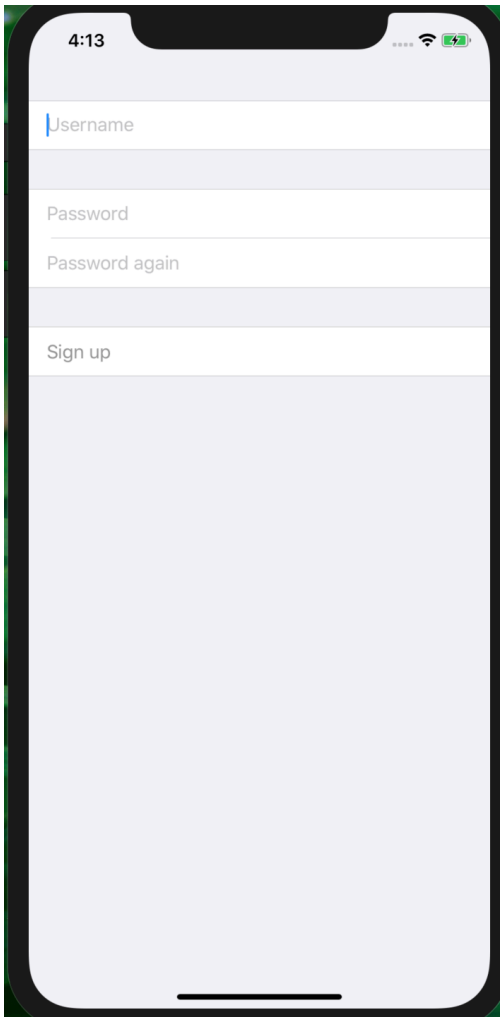
- @Published<Output> představuje AnyPublisher<Output, Never>
- Je vlastněn jako instanční proměnná objektu.
- Provázání publisherů musí být provedeno v inicializátoru objektu, tj. nelze ho zadat ”deklarativně”.
  - Jak publisher referencovat. Jak ho držet, vlastnit.
  - Jak naložit s jeho výsledkem.

Opakování: konstrukce objektu

```
// AnyPublisher<Int,Never>
let base = Just(3)
//
class Muj {
    // korektni, prava strana je mimo "self"
    let a = 3
    // toto nelze, prava strana pouziva "self"
    //let b = a + 1
    let b: Int
    // toto je funkce, neni to ulozena hodnota
    var c: Int { a + b }
    // toto je ulozena hodnota, ale musim ji "nastartovat"
    lazy var d = a + b
    // toto se provede pred vstupem do init()
    let e = base.map { $0 + 1}.sink { print($0) }
    //
    init() {
        // co muzu delat, nez dokompletuju ”self”
        b = a + 1
        // startuju lazy "d"
        let _  = d
    }
}
```

```
//
struct ContentView: View {
    //
    @ObservedObject private var userViewModel = UserViewModel()
    //
    var body: some View {
      Form {
        Section {
          TextField("Username", text: $userViewModel.username)
            .autocapitalization(.none)
          }
          Section {
            SecureField("Password", text: $userViewModel.password)
            SecureField("Password again", text:
$userViewModel.passwordAgain)
          }
          Section {
            Button(action: { }) {
              Text("Sign up")
            }.disabled(userViewModel.isValid == false)
          }
       }
    }
}
```

Observable Model




```
class UserViewModel: ObservableObject {
  // Input
    @Published var username = ""
    @Published var password = ""
    @Published var passwordAgain = ""

    // Output
    @Published var isValid = false
```

Elementátní publishery

```
//
    var userNameValid: AnyPublisher<Bool, Never> {
        // username je String
        // _username je @propertyWrapper, Published<String>
        // _username.projectedValue, Published<String>.Publisher
        // ted se obracime na Publishera, $
        $username
            // pockame si 0.8s
            .debounce(for: 0.8, scheduler: RunLoop.main)
            .removeDuplicates()
            // explicitne zobrazim typ, je to String
            // mapovani (String)->(Bool)
            .map { (unCandidate: String) in unCandidate.count >= 3}
            .eraseToAnyPublisher()
    }
```

znovup…

```swift
//
typealias OkPublisher<Output> = AnyPublisher<Output, Never>

//
func checkUserName(_ username: OkPublisher<String>) -> OkPublisher<Bool>
{
    username
        .debounce(for: 0.8, scheduler: RunLoop.main)
        .removeDuplicates()
        .map { (unCandidate: String) in unCandidate.count >= 3}
        .eraseToAnyPublisher()
}
```

```swift
var passwordValid: AnyPublisher<Bool, Never> {
    //
    $password
        .debounce(for: 0.8, scheduler: RunLoop.main)
        .removeDuplicates()
        // kontroluj kvalitu hesla....
        .map { $0.isEmpty == false }
        .eraseToAnyPublisher()
}
//
var comparePasswords: AnyPublisher<Bool, Never> {
    //
    Publishers.CombineLatest($password, $passwordAgain)
        .debounce(for: 0.2, scheduler: RunLoop.main)
        .map { $0 == $1 }
        .eraseToAnyPublisher()
}
```

```
//
    var checkPasswords: AnyPublisher<Bool, Never> {
        //
        Publishers.CombineLatest3(passwordValid, $password,
$passwordAgain)
            .map { pOK, p1, p2 in (p1==p2) && pOK }
            .eraseToAnyPublisher()
    }

    //
    var checkForm: AnyPublisher<Bool, Never> {
        //
        Publishers.CombineLatest(userNameValid, checkPasswords)
            .map { u, p in u && p }
            .eraseToAnyPublisher()
    }
```

Výsledek

```
class UserViewModel: ObservableObject {
    @Published var password = ""
    @Published var passwordAgain = ""
    @Published var isValid = false
    @Published var username = ""
    //
    var cancellableSet = Set<AnyCancellable>()
    //
    init() {
        //
        checkForm
            .assign(to: \.isValid, on: self)
            .store(in: &cancellableSet)
    }
```

CoreData

```
//
struct PrvniTAB : View {
    //
    @Environment(\.managedObjectContext) var managedObjectContext
    // tohle je vyznamem NSFetchedResultsController
    @FetchRequest(
        entity: NB.entity(),
        sortDescriptors: [NSSortDescriptor(keyPath: \NB.title,
ascending: true)]
    ) var sesity: FetchedResults<NB>
    //
    @State var addSheet = false
    //
    var body: some View {
        //
        NavigationView {
            //
            List(sesity, id: \.self) { itm in Text(itm.title!) }
```

SwiftUI na macOS

- Ambice Applu jsou sjednotit programování UI pro macOS a iOS.
  - tvOS, watchOS je trivialita.
- Implementace SwiftUI pro macOS je zatím velmi nepřesvědčivá.
  - Cocoa/macOS je velmi odlišný v VC, tabulkách, …
  - Jak asi bude vypadat zdroják XCode ve SwiftUI? ;)

Závěr

- SwiftUI+Combine.
  - Příliš rozsáhlé téma. Musíme se k němu ještě vracet.
  - CoreData. CloudKit. Dokumenty.
  - Příště: Kódování dat, UIDocument.
- Jisté je:
  - Formulace V-VM je úžasná.
  - Je třeba se probojovat přes Model a procesy. Nakonec, Model má být zcela izolován od -V-C, nebo -V-VM.