Tvorba mobilních aplikací

7. seminář

Radek Janoštík

Univerzita Palackého v Olomouci

29. 3. 2022

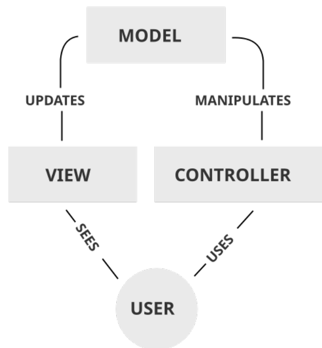
Architektura aplikací

- Struktura aplikace potřebuje nějaký řád
  - Chaotické řešení událostí, funkcionality, UI, práce na pozadí nikam nevede
  - Je dobré mít aplikaci rozdělenou na menší, ucelené celky
  - Rozdělení zodpovědnosti za funkcionalitu
- Není potřeba „znovu vynalézat kolo“
- Vedle návrhových vzorů (Singleton, Adapter, Factory, Builder, . . .) máme osvědčené vzory pro architekturu aplikací
- S některými jsme se již setkali nebo setkáme (DAO, DTO, Publish-subscribe)
- Vhodnost architektury podle typu aplikace
- Na platformě Android je velmi doporučená *Model-View-ViewModel (MVVM)*, případně *Model-View-Controller*, *Model-View-Presenter*
  - Mírné odlišnosti
  - Rozdělení zodpovědnosti

Model-View-Controller (MVC)

- Model – Aplikační data a veškerá logika (nezávislá na zobrazení)
  - Databázové entity
  - Interní stav aplikace
  - Veškerá funkcionalita
- View – Komponenty, které zobrazují data (která obdrží)
  - Logika pouze za účelem prezentace
  - „hloupé zobrazovače“
  - Zasílání uživatelských akcí (klik, přejetí myši, . . .) kontroleru
- Controller – vrstva mezi Modelem a View
  - Přijímá uživatelské vstupy (validace) a předává Modelu
  - Controller zná své View

MVC obrázkem

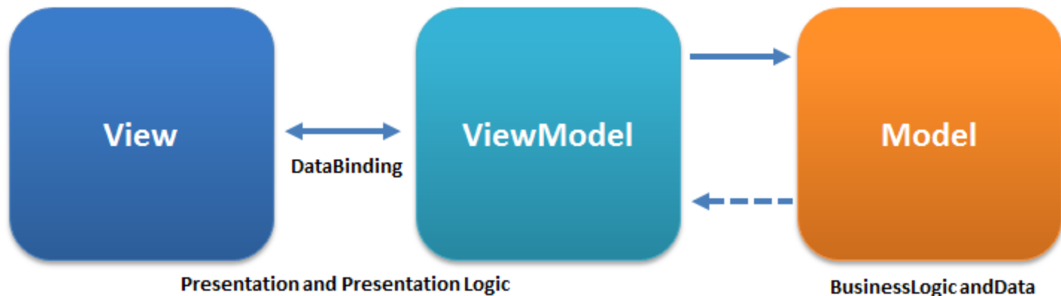


Obrázek: Zdroj: `https://en.wikipedia.org/wiki/File:MVC-Process.svg`

Model-View-ViewModel (MVVM)

- Model – Aplikační data a veškerá logika (nezávislá na zobrazení)
  - Databázové entity
  - Interní stav aplikace
  - Veškerá funkcionalita
- View – Pouze zobrazovací komponenty
  - Žádná aplikační logika
  - Žádná UI logika
- ModelView – Obsahuje data, která se mají zobrazit
  - Nezná svůj View – máme *binder*, který mapuje proměnné na prvky ve View
  - „Překlápí“ uživatelské akce do Modelu
- Rozdělení rolí ⇒ lepší testovatelnost
  - Můžeme logiku (Model) testovat např. Unit testy
  - Můžeme testovat UI bez ohledu na Model (změna proměnných ve ViewModelu)

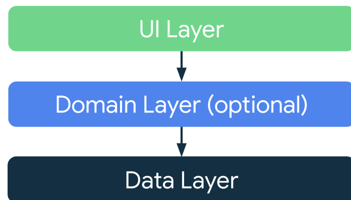
MVVM obrázkem



Obrázek: Zdroj: `https://en.wikipedia.org/wiki/File:MVVMPattern.png`

MVVM v androidu

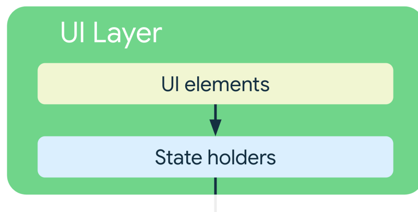
- Jistě cítíte, že „zjišťovat EditText podle ID, nastavovat mu text, případně získávat" ručně není správná cesta
- Uživatelské rozhraní by mělo být řízeno (persistentními) daty, které nejsou závislé na UI
  - Nedojde ke ztrátě dat, když OS „zabije" aplikaci
  - Nepřijdeme o data při otočení displeje, ...
- Je doporučováno rozdělit aplikaci na 2 vrstvy – UI vrstva a Datová vrstva



Obrázek: Zdroj: `https://developer.android.com/jetpack/guide`

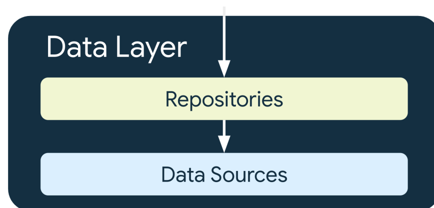
Vrstva UI – prezentační vrstva

- Zobrazuje aplikační data
- Interakce s uživatelem
- „Přepracování dat" – potřeba pouze části, více zdrojů, . . .
- Definice UI elementů
- Interní stav UI a jeho korespondující zobrazení



Datová vrstva

- Sdružuje aplikační data veškerou aplikační (business) logiku
- Řeší kam budou data uložena, jak budou transformována
- „Životnost“ je delší než „jedna obrazovka“
- Persistentní stav
- Poskytnutá data vyšším vrstvám by měly být pouze read-only
- Složena z *repositářů* (pro každou entitu zvlášť)
    - Obsahují datové zdroje
    - Sdružují funkcionalitu okolo jedné datové struktury
    - Změny dat na jednom místě
    - Aplikační logika



Doménová vrstva

- Nepovinná vrstva – koncepčně nic nemění
- Obsahuje společnou funkcionalitu pro více view modelů
- Bývá přítomna u komplexních aplikací
- Odbourává duplicitu v kódu
- Neměla by zacházet s mutovatelnými daty

Třída ViewModel

- Systém může zničit UI prvky (aktivity, fragmenty) kdy se mu zachce
  - Ztratíme uložený stav v těchto komponentách
- Časově náročnější operace bychom neměli volat z vlákna UI
- $\Rightarrow$ Třída `ViewModel` – součástí Android Jetpack
- Lifecycle-aware komponenta – přežívá životní cyklus aktivit
  - Navržená k uchování vnitřního stavu UI
  - Podpora *Kotlin coroutines* pro asynchronní operace
- Pomocí *bindingu* se bude automaticky mapovat na UI prvky
- Pojďme předělat naši aplikaci na MVVM architekturu

Třída pro aplikaci

- Prozatím jsme měli jako „hlavní třídu“ nějakou aktivitu, která se spustila
- Podědíme z `android.app.Application` – třída pro správu globálního stavu aplikace
- Bude obsahovat odkaz na databázi a později na repositáře

```
class MyApplication : Application() {
    val database by lazy { AppDb.getDatabase(this) }

}
```

- Do `AndroidManifest.xml` přidáme do elementu `application` odkaz na naši třídu

```
android:name=".MyApplication"
```

LiveData

- Při přidání nové poznámky jsme museli ručně obnovovat seznam
- „Překreslovat“ všechny komponenty pracující s tímto seznamem
- Funkcionalitu jde zjednodušit pomocí `LiveData` `https://developer.android.com/topic/libraries/architecture/livedata`
- $\Rightarrow$ „Pozorovatelný“ nosič dat
  - Přihlásíme se k odběru „změn“
  - Při změně zdrojových dat se provede definovaná operace
- LiveData berou v potaz životní cyklus komponent
  - $\Rightarrow$ nebudou notifikovat již mrtvé odběratele
- Paměťově bezpečné, šetří spoustu kódu a zvyšují přehlednost

Přidání LiveData a ViewModelu

- Do build.gradle přidáme nutné závislosti

```
def lcVersion = "2.4.1"
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lcVersion"
implementation "androidx.lifecycle:lifecycle-viewmodel-compose:$lcVersion"
implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lcVersion"
implementation "androidx.lifecycle:lifecycle-runtime-ktx:$lcVersion"
implementation "androidx.lifecycle:lifecycle-viewmodel-savedstate:$lcVersion"
```

- A upravíme návratovou hodnotu metody `getAll` v `noteDao`

```kotlin
fun getAll() : LiveData<List<Note>>
```

Vytvoření ViewModelu

- Nyní již můžeme vytvořit ViewModel pro hlavní aktivitu
- Bude obsahovat odkaz na aplikaci a (zatím jen) poskytovat všechny poznámky pro UI

```
class MainActivityViewModel(val app: MyApplication) : ViewModel() {
    val notes = app.database.noteDao().getAll()
}
```

- Ještě musíme vytvořit *továrnu*, která bude vytvářet instance

```
class MyModelFactory(private val app: MyApplication) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(MainActivityViewModel::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return MainActivityViewModel(app) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Přepsání Adaptéru

- Předěláme adaptér pro RecyclerView, aby nebyl závislý na aktivitě
  - Bude si udržovat odkaz na seznam poznámek

```
class NoteAdapter(notesParam: List<Note>) :
    RecyclerView.Adapter<NoteAdapter.NoteItemHolder> () {
    var notes = notesParam
    set(value) {
        field = value
        notifyDataSetChanged()
    }
```

- Do setteru přidáme „překreslení“ RecyclerView

Nastavení modelu

- Už jen zbývá zajistit vytvoření modelu pro aktivitu
- Do `build.gradle` přidáme

```
implementation "androidx.activity:activity-ktx:1.4.0"
```

- Přidáme vytvořený ViewModel jako *vlastnost* aktivitě

```
private val model: MainActivityViewModel by viewModels {
    MyModelFactory((application as MyApplication)) }
```

- Přihlásíme se k odběru změn v databázi poznámek

```
model.notes.observe(this) { adapter?.notes = it
                recyclerView?.scrollToPosition( it .size−1)}
```

- A pročistíme kód od zbytečných volání

Další funkcionalita – binding

- Chtěl bych přidat další funkcionalitu
  - Po přidání první poznámky se změní text tlačítka na „Přidat další“
  - Nahoře zobrazovat titulek naposledy přidané poznámky
  - Data z EditText získávat „lépe“ než pomoci `findViewById`
- To vše uděláme pomocí ViewModelu a *bindování*
- Do `build.gradle` přidáme do sekce `android{}`

```
buildFeatures { dataBinding true }
```

- Upravíme layout aktivity – přidáme „nad-element“

```
<layout xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
<data>
  <variable
   name="model"
   type="com.example.recyclerviewbase.viewModels.MainActivityViewModel"
        />
</data>
```

Další funkcionalita – binding

- Aby byl viewModel dostupný z XML, musíme do `onCreate()` přidat

```
val binding: ActivityMainBinding = DataBindingUtil.setContentView(
this, R.layout.activity_main)
binding.model = model
```

- Do ViewModelu přidáme *pozorovatelné* vlastnosti

```
var lastEnteredString = ObservableField<String>("Zatim nic nepridano")
var added = ObservableField<Int>(R.string.add)
var noteAddTitle = ObservableField<String>()
var noteAddBody = ObservableField<String>()
```

Bindování dat

- Pro každé XML s layoutem se nám automaticky generuje třída – např.: `ActivityMainBinding`
  - Každý view element s ID v ní máme dostupný „přes tečku“
- Budeme používat Data Binding Library
- V XML můžeme definovat, která *vlastnost* modelu se má zobrazovat v jakém prvku

```
android:text="@{model.added}"
```

- Obsah `ObservableField added` se bude promítat do textu tlačítka
- Podobně můžeme přidat binding pro zobrazení poslední přidané poznámky
- K dispozici máme „jazyk“ pro definování složitějších vazeb
- `https://developer.android.com/topic/libraries/data-binding/expressions`

Bindování dat

- Výše zmíněné funguje ve směru `ViewModel`⇒`Layout`
- Pro získávání dat z formulářů se hodí i opačný směr
- Pro obousměrné bindování místo `@` použijeme `@=`

  android:text=`"@={model.noteAddTitle}"`
- Události by měl také obsluhovat `ViewModel`

  android:onClick=`"@{model::saveClick}"`
- Ukázka

Dokončení MVVM

- V naší aplikace ještě stále mícháme zobrazovací logiku a aplikační logiku
- ViewModel by se měl starat pouze o zobrazovací logiku, nevytvářet nové entity
- Vytvoříme `NoteRepository`
  - Bude obsahovat veškerou aplikační logiku „kolem“ poznámek
  - ViewModel bude pouze volat jeho metody a „přemosťovat“ data a volání

```
fun createNote(title: String, text: String) : Note {
    val note = Note(0, title, text, false)
    val newId = database.noteDao().insert(note)
    return note.copy(id = newId)
}
```

- Nyní již máme čistou MVVM architekturu aplikace
  - Na první pohled jsou jasné „zodpovědnosti“ komponent

Rest z minula

- Minule i dnes jsme používali synchronní dotazy do DB
- Delší dotazy se vykonávaly ve vlákně UI
- Mohlo docházet ke zpomalení
- ViewModel umožňuje pracovat asynchronně pomocí *Kotlin coroutines*
  - Nástroj pro snadné psaní asynchronních metod
  - Běh korutiny může být *uspán* či *přerušen*
  - Každá korutina běží v nějakém *scope* omezující její „životnost“
- ViewModel poskytuje *viewModelScope*
  - Při zničení daného ViewModelu dojde k přerušení všech běžících korutin

Rest z minula

- Upravíme metodu `insert` z `noteDao`, aby mohla být přerušena

```
suspend fun insert (note: Note): Long
```

- Takto označené metody mohou být volány ze *scope* korutiny nebo z jiné suspend funkce
- Změníme volání vytváření nové poznámky

```
viewModelScope.launch(Dispatchers.IO) {
    app.noteRepo.createNote(title, noteAddBody.get()?:"")
}
```

- vytvoření nové poznámky bude provedeno ve vlákně pro IO operace
- Nyní již můžeme odmazat `allowMainThreadQueries()` z `AppDb`

Úkol

1. Předělejte svoji aplikaci na MVVM architekturu
   - Pro každou aktivitu definujte ViewModel
   - Data zobrazujte a získávejte pomocí bindování
2. Nastudujte si sami téma Kotlin coroutines (bez i s návazností na Android)
   - Vyzkoušejte si nějaké příklady (budu chtít vidět)