

Model-View-ViewModel LiveData Navigation



Peter Borovanský
KAI, I-18

MS-Teams: [2sf3ph4](#), [List](#), [github](#)

borovan 'at' ii.fmph.uniba.sk



Kap. 39 – 46 Modern Android Architecture with JetPack
Kap. 47 – 48 Navigation Architecture Component

Plán

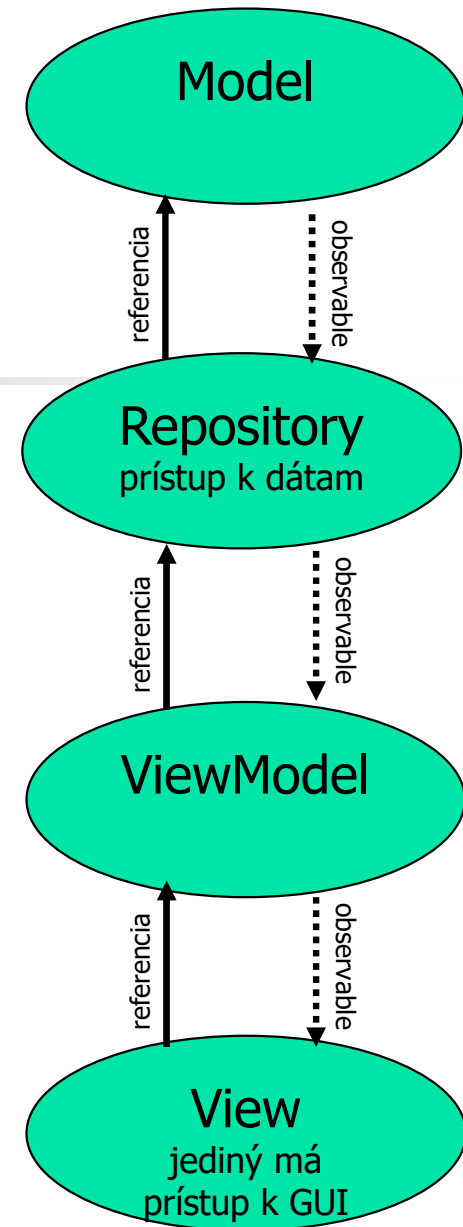
Návrhové vzory:

- Model View Presenter (MVP)
- Model View Controller (MVC)
- Model View ViewModel (MVVM)
 - LiveData
 - DataBindings
 - JetPack library
- Cvičenie - malé príklady:
 - konvertovacia kalkulačka
- Cvičenie - midterm

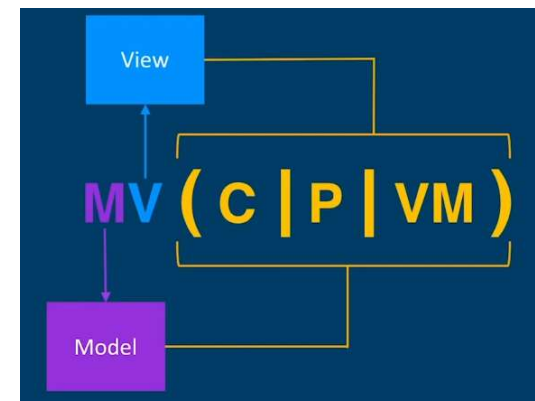
Alternatíva (dobře pokrýva MVVM):

kurz android basic compose

<https://codelabs.developers.google.com/codelabs/kotlin-android-training-view-model/>



MV [C | P | VM]



Atribúty dobrého kódu

- stabilný – k drobným zmenám
- robustný – prežije, ak sa zväčšuje projekt, resp. komplikuje sa, resp. inak sa vyvíja
- testovateľný nezávisle – GUI, Model (model – junit testami, GUI – mockovacími technikami)
- modulárny

3 bežne používané návrhové vzory majú v názve spoločné model&view a líšia sa:

- Controller
- Presenter
- ViewModel (o tomto bude väčšina prednášky)

Majú spoločné:

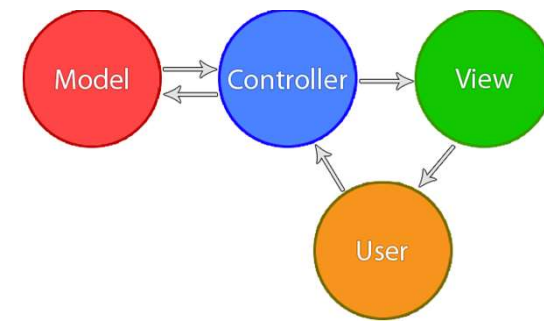
- **Model** – implementuje *business logic*, nevie **nič** o prezentácií dát, eventoch, ...
 - cez **Repository** komunikuje s databázou, internetom, resp. lokálnym zdrojmi (resources) ...
 - **vystavuje svoje dáta**, komukoľvek, kto ich potrebuje
 - nemá nič spoločné s Androidom (triedami androidx), môžete k nemu napísať sériu junit testov
- **View** – zobrazuje dáta
 - nevie nič o ich logike dát, ich pôvode, uložení, ...

Chýba im, líšia sa:

- v medzivrstva, ktorá prepája **Model** - ...???... - **View**

Architektonický *mess*

(MVC)



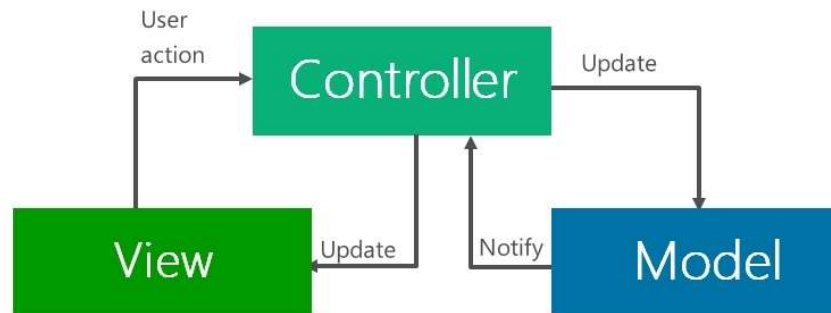
...vzniká, ak vizuálne komponenty (Views) sú v kóde zviazané s dátovými objektami a opačne, príklad:

```
prev.setOnClickListener(new OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        i++;  
        i %= imgs.length;  
        iv.setImageDrawable(imgs[i]);  
    }  
});
```



- preto sa pri návrhu GUI používajú návrhové vzory (design patterns)

3 Tier Architecture - iOS



napr. Model-View-Controller
alebo Model-View-ViewModel

- **motto:** the architecture of most Android-apps (*in the pass time*) is a mess.
- v Androide Activita často reprezentuje rolu View aj Controllera, aj Modelu

Model View Controller (MVC)

(Model sú len dáta netušiace nič o ich prezentácii)



```
class Model() : Observable() { ← vystavuje svoje dáta
    private var indx = 0
    private var list = mutableListOf<Drawable>()
```

```
    fun addDrawables(imgs: List<Drawable>) {
        list.addAll(imgs)
    }
```

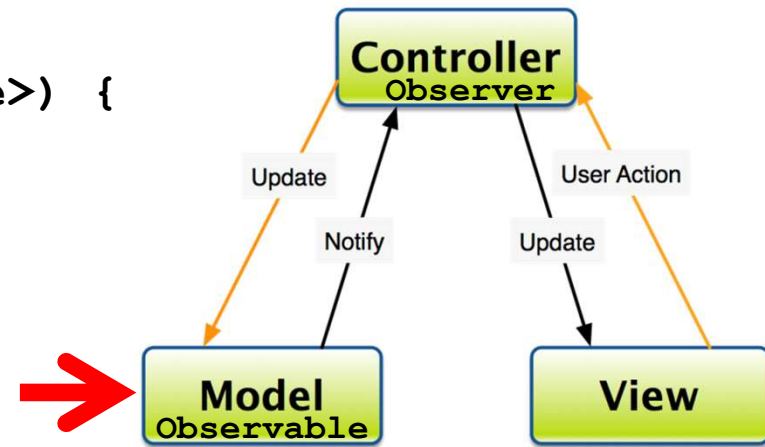
```
    val currentDrawable: Drawable
    get() = list[indx]
```

```
    fun nextValue() {
        indx++
        indx %= list.size
        setChanged()
        notifyObservers()
    }
```

promenáda
dát →

```
setChanged()
notifyObservers()
```

[java.util.Observable](#)
[setChanged\(\)](#) - marks this Observable object as having been changed
[notifyObservers\(\)](#)
[notifyObservers\(Object arg\)](#) - if hasChanged, then notify all of its observers and then call the clearChanged = no longer changed.



```
    fun prevValue() {
        indx--
        if (indx<0) indx = list.size-1
        setChanged()
        notifyObservers()
    }
```

Model View Controller (MVC)

(Controller – komunikuje medzi modelom a view)

```
class MainActivity : AppCompatActivity(), Observer {  
    lateinit var myModel: Model  
    lateinit var myView: MyView  
    lateinit var binding : ActivityMainBinding  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        binding = ActivityMainBinding.inflate(layoutInflater)  
        setContentView(binding.root)  
        myModel = Model() // inštancia business modelu  
        myModel.addObserver(this) // this-Controller je observerom modelu  
        myModel.addDrawables(Repository.allDrawables(this))  
        myView = MyView(binding, this) // views potrebujú context MainActivity  
    }  
}
```

Controller ako manažer nerozumie dátam, nevie ich zobrazit', ale dohodne to medzi nimi (Modelom a View)

tu sa model dozvie, že ho niekto sleduje

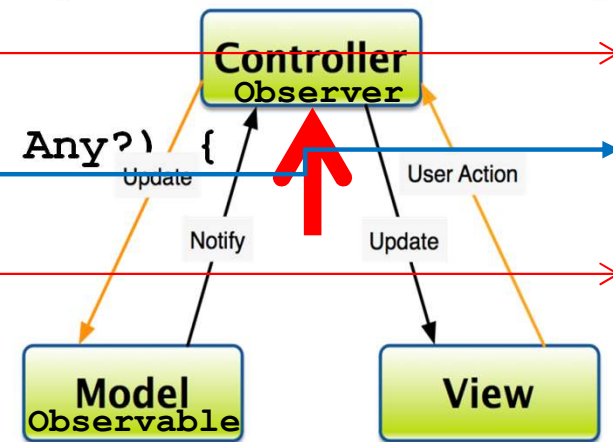
// interface Observer

```
override fun update(arg0: Observable, arg1: Any?) {  
    myView.myupdate(myModel.currentDrawable)  
}
```

tu sa controller dozvie, že sledovaný model zmenil dáta

[java.util.Observer](#)
[update](#)(o : [Observable](#), arg : Any?) - this method is called whenever the observed object is changed.

PikachuMVC.zip



Model View Controller (MVC)

(View je GUI, zobrazenie Views, eventy)

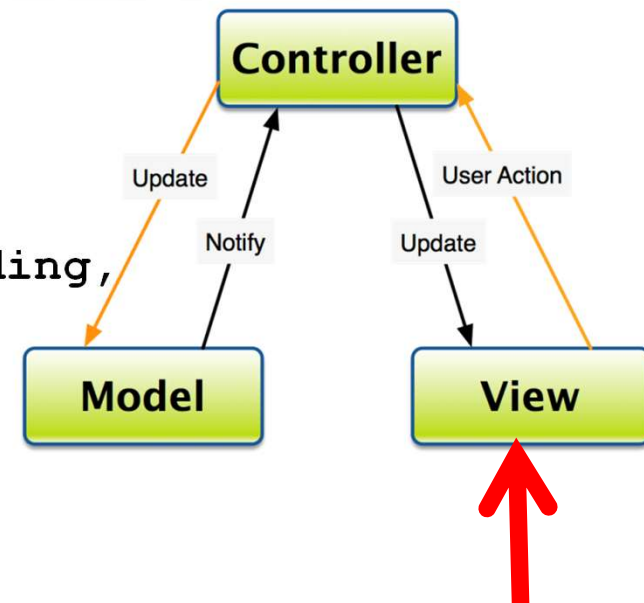


View

- prezentuje dáta vo Views
- odchyťava eventy
- musí vedieť, kto je jeho manažér = Controller

```
class MyView(val binding:ActivityMainBinding,  
             val main: MainActivity) {  
    // pointer na Controller
```

```
    init {  
        binding.prevBtn.setOnClickListener {  
            main.myModel.prevValue()  
        }  
        binding.nextBtn.setOnClickListener {  
            main.myModel.nextValue()  
        }  
        myupdate(main.myModel.currentDrawable)  
    }  
    fun myupdate(im:Drawable) {  
        binding.imageView1.setImageDrawable(im)  
    }  
}
```



keby nemal referenciu na controller nemá ako manažérovi oznámiť, že nastal GUI event

Controller je trieda zodpovedná za komunikáciu medzi View a Model

Controller prikázal prekresliť Views, tak prekresli

Model View Controller (MVC)

(Repository – sprístupňovač dát)



Repository jediné vie, či dáta

- sú lokálne
- sú z lokálnej databázy, napr. Room, resp. cloudovej databázy, napr. FireBase alebo
- sú z netu, cez nejakú webovú službu, REST API, či servis, ...

```
class Repository {  
    companion object {  
        fun allDrawables(context : Context) =  
            listOf(  
                R.drawable.butterfree, R.drawable.golbat, R.drawable.kakuna,  
                R.drawable.raichu, R.drawable.venomoth, R.drawable.venusaur,  
                R.drawable.pok0, R.drawable.pok1, R.drawable.pok2,  
                R.drawable.pok3, R.drawable.pok4, R.drawable.pok5,  
                R.drawable.pok6  
            ).map {  
                ContextCompat.getDrawable(  
                    context.applicationContext, it)!!  
                }  
            }  
    }  
}
```




Model View Controller

review

Čo je tu divné ?

```
class MyView(val binding: ActivityMainBinding,
              val main: MainActivity) { // pointer na Controller

    init {
        binding.prevBtn.setOnClickListener {
            main.myModel.prevValue()
        }
        myupdate(main.myModel.currentDrawable)
    }
    fun myupdate(im: Drawable) {
        binding.imageView1.setImageDrawable(im)
    }
}
```

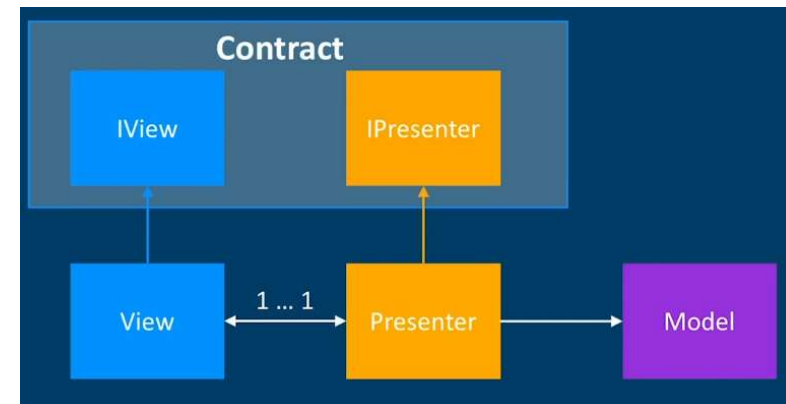
Nemal by mať controller svoj interface, a View svoj interface

```
interface MyViewInterface { fun myupdate(im: Drawable) }
interface ControllerInterface {
    fun prevValue()
    fun nextValue()
}
```

Cons: výsledkom používania návrhových vzorov je často zväčšenie objemu kódu ...

Pros: aj keď aplikácia prestane byť triviálna, scale-up, tak sa vám to nerozpadne

Model View Presenter



```
interface Login {  
    interface View {  
        fun setUsername(name : String)  
        fun setPassword(passs : String)  
        fun showValidationSuccessful()  
        fun showValidationFailed()  
        fun setPresenter(p : Login.Presenter)  
    }  
    interface Presenter {  
        fun loginUser(name:String, pass:String)  
    }  
}
```

interface pre View
... myupdate ..

interface pre Presenter

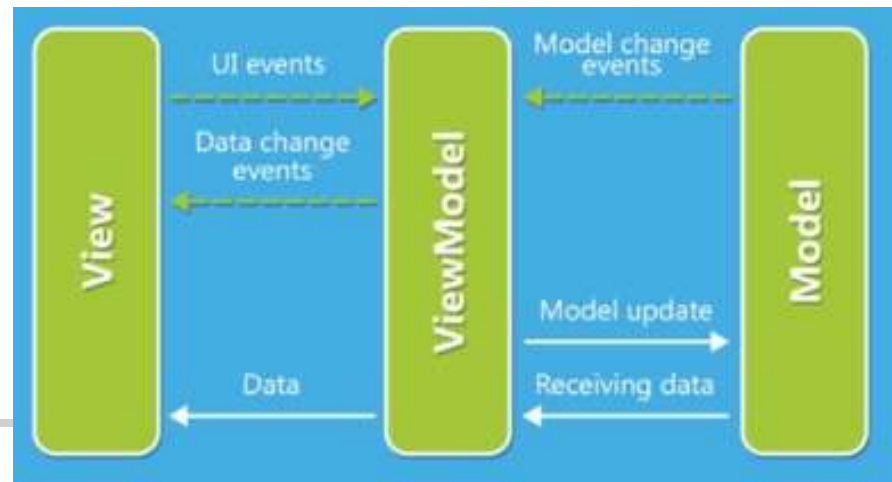
```
class LoginView : Login.View {  
    lateinit var mpresenter : Login.Presenter  
    override fun setUsername(name: String) { }  
    override fun setPassword(passs: String) { }  
    override fun showValidationSuccessful() { }  
    override fun showValidationFailed() { }  
    override fun setPresenter(p: Login.Presenter) {  
        mpresenter = p  
    }  
}
```

```
class LoginPresenter (view:Login.View):Login.Presenter {  
    var mView : Login.View  
    init {  
        mView = view  
        mView.setPresenter(this)  
    }  
    override fun loginUser(name: String, pass: String) { }  
}
```



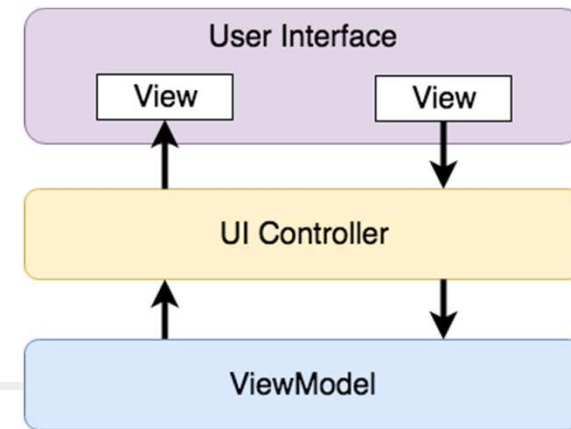
- MainActivity.kt: (8, 43): 'Observer' is deprecated. Deprecated in Java
- MainActivity.kt: (15, 17): 'addObserver(Observer!): Unit' is deprecated. Deprecated in Java
- MainActivity.kt: (22, 31): 'Observable' is deprecated. Deprecated in Java
- Model.kt: (6, 17): 'constructor Observable()' is deprecated. Deprecated in Java
- Model.kt: (6, 17): 'Observable' is deprecated. Deprecated in Java
- Model.kt: (18, 9): 'setChanged(): Unit' is deprecated. Deprecated in Java
- Model.kt: (19, 9): 'notifyObservers(): Unit' is deprecated. Deprecated in Java
- Model.kt: (25, 9): 'setChanged(): Unit' is deprecated. Deprecated in Java
- Model.kt: (26, 9): 'notifyObservers(): Unit' is deprecated. Deprecated in Java

Čo je JetPack



- celý moderný vývoj iOS postavený na jazyku Swift je striktne založený na Model-View-Controller vzore (MVC)
- Model-View-Controller je založený na triedach Observable a Observer
- na mnohých príkladoch single activity apps sme videli, že sa mieša kód pre GUI s **business** logikou aplikácie
- Google si to uvedomil 2017 a navrhol JetPack pre multi-activity apps
- cieľom:
 - je oddeliť kód pre GUI od kódu s logikou
 - vyriešiť problémy so životným cyklom, napr. pri rotácii displaya
 - zabezpečiť perzistenciu dát (inak ako sme to robili cez SharedPreferences)
- architektúra separácie GUI a logiky kódu založená na ViewModel, nie MVC
- MVVM pochádza od Microsoft, 2005
- ViewModel je analógia k Controlleru (MVC), či k Presenteru (MVP)
- ViewModel je také *lepidlo*, čo nejako inak spája View a Model

Model View ViewModel



- **ViewModel** je jediný, čo vie o dátach a ich logike
- keď zmeníme GUI, **ViewModel** zostáva nezmenený
- ak sa zmení napr. orientácia, tak **ViewModel** stále drží pôvodné dáta
- **View** oznamuje **ViewModelu**, čo sa zmenilo, napr. UI events, zadaný login,..
- **View** má referenciu na **ViewModel**, cez ktorú mu to oznamuje
- **ViewModel** nemá žiadnu predstavu o **View**, len ponúka dáta (producer)
- **View** je v roli prijímateľa dát (consumer) a **ViewModel** sa nestará o to, kto dáta konzumuje, a či...
- dáta sa ale môžu meniť nezávisle na GUI, a aj často, napr. realtime data
- ako často/resp. kedy sa má GUI dopytovať, či nemá už dáta prekresliť, či sa náhodou nezmenili ?
- agresívne „spojité“ pool-ovanie dát je náročné, tak sa to nerobí...

View ViewModel

(consumer producer)

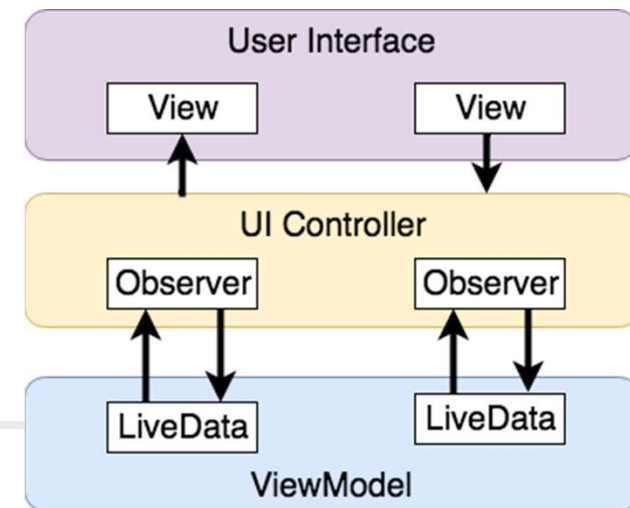
Preto je na to generická trieda LiveData (JetPack):

LiveData – Observable/Observer

- **ViewModel** vystavuje premennú LiveData<T>, resp. MutableLiveData<T>
- ktokoľvek kto sa stane observerom pre ViewModel sa dozvie o zmenách tejto premennej, teda observer dostane info, ak sa dáta zmenia
- ak aktivita-fragment prestane byť aktívna-y a opäť sa prebudí, dostane rovnaké data ako mala pred deaktiváciou
- ak aktivita-fragment zmení orientáciu, tak po zmene sa opäť obnovia jej pôvodné dáta v premennej typu LiveData

Výhody:

- nepíšeme množstvo interface-ov, ako pri MVP
- vzťah medzi **View** a **ViewModel** nie je silne zviazaný (párom pointrov)
- **ViewModel** ani netuší, či a aké **Views** ho observujú (počúvajú)
- ergo, **ViewModel** sa ani nemusí zaujímať, či **View** ešte existuje, žije...



Projekt Fragment+ViewModel

(verzia 1 – dostanete zadarmo)

```
class MainFragment : Fragment() { // má rolu View
    companion object { // statická metoda
        fun newInstance() = MainFragment()
    }
    private lateinit var viewModel: MainViewModel // referencia na ViewModel
    private lateinit var binding : MainFragmentBinding

    override fun onCreateView(inflater: LayoutInflater,
                              container: ViewGroup?,
                              savedInstanceState: Bundle?): View {
        binding = MainFragmentBinding.inflate(inflater, container, false)
        return binding.root
    }
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
        // TODO: Use the ViewModel
    }
}
```

```
import androidx.lifecycle.ViewModel

class MainViewModel : ViewModel() {
    // TODO: Implement the ViewModel
}
```

Projekt Fragment+ViewModel

(verzia 1 – ViewModel, ViewModelProvider)

```
class MainFragment : Fragment() { // má rolu View
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
        binding.apply {
            convertBtn.setOnClickListener {
                if (inputAmount.text.isNotEmpty()) {
                    viewModel.convertUSD2EURO = usd2euro.isChecked
                    viewModel.setInputCurrencyAmount(inputAmount.text.toString())
                    outputAmount.setText("%.2f".format(viewModel.outputCurrencyAmount))
                }
            }
        }
    }
}
```

```
class MainViewModel : ViewModel() {
    val dolar2euroRate = 0.95f
    var convertUSD2EURO = true
    var inputCurrencyAmount = 0f
    var outputCurrencyAmount = 0f

    fun setInputCurrencyAmount(value : String) {
        inputCurrencyAmount = value.toFloat()
        outputCurrencyAmount =
            if (convertUSD2EURO) inputCurrencyAmount * dolar2euroRate
            else inputCurrencyAmount / dolar2euroRate
    }
}
```

Pros:
máme oddelené views a dáta
Cons:
o GUI refresh sa staráme my

LiveData

(verzia 2 – Observer, MutableLiveData<T>)

Pros:

observer sa automaticky dozvie o zmene premennej LiveData, na ktorú je priviazaný

Cons:

do GUI to musím explicitne zapísať my

```
class MainFragment : Fragment() {
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
        val resultObserver = Observer<Float> {
            result -> outputAmount.setText("%.2f".format(result))
        }
        viewModel.outputCurrencyAmount.observe(viewLifecycleOwner, resultObserver)
        convertBtn.setOnClickListener {
            if (inputAmount.text.isNotEmpty()) {
                viewModel.convertUSD2EURO = usd2euro.isChecked
                viewModel.setInputCurrencyAmount(inputAmount.text.toString())
            }
        }
    }
}
```

```
class MainViewModel : ViewModel() {
    val dolar2euroRate = 0.95f
    var convertUSD2EURO = true
    var inputCurrencyAmount = 0f // ViewModel vystaví dáta pomocou LiveData
    var outputCurrencyAmount : MutableLiveData<Float> = MutableLiveData()
    fun setInputCurrencyAmount(value : String) {
        inputCurrencyAmount = value.toFloat()
        outputCurrencyAmount.value =
            if (convertUSD2EURO) inputCurrencyAmount * dolar2euroRate
            else inputCurrencyAmount / dolar2euroRate
    }
}
```

LiveData

(verzia 2++ viac kotlinish)

Pros:

MutableLiveData premenná je private vo ViewModel

Observerovi sa vystavuje len non-mutable LiveData premenná

Kotlin na to ponúka getter property

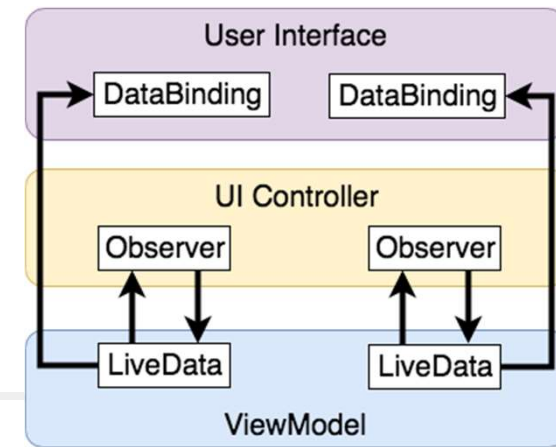
```
class MainFragment : Fragment() {
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        convertBtn.setOnClickListener {
            if (inputAmount.text.isNotEmpty()) {
                viewModel.convertUSD2EURO = usd2euro.isChecked
                viewModel.inputCurrencyAmount = inputAmount.text.toString().toFloat()
            }
        }
    }
}
```

```
class MainViewModel : ViewModel() {
    val dolar2euroRate = 0.95f
    var convertUSD2EURO = true
    set(value) { field = value }
}
```

```
private val _outputCurrencyAmount:MutableLiveData<Float> = MutableLiveData()
val outputCurrencyAmount : LiveData<Float>
    get() = _outputCurrencyAmount
```

```
var inputCurrencyAmount = 0f
    set (value : Float) { field = value
        _outputCurrencyAmount.value = if (convertUSD2EURO)
            inputCurrencyAmount * dolar2euroRate
        else inputCurrencyAmount / dolar2euroRate
    }
}
```

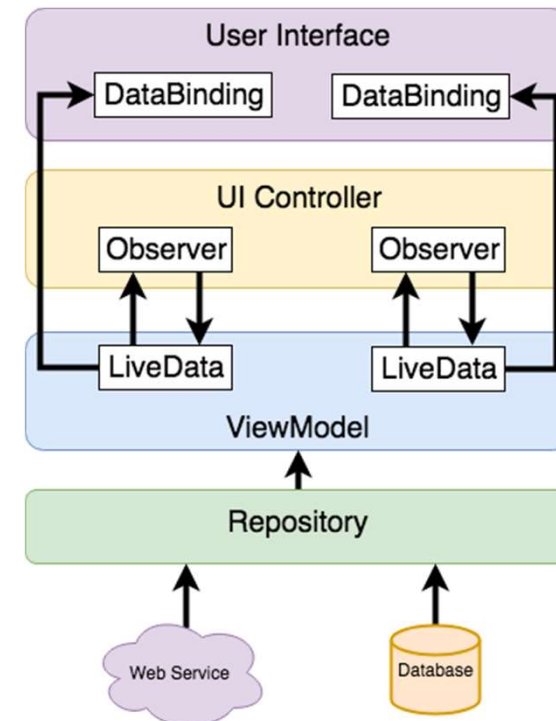
Data Binding



- ako zabezpečiť, aby sa dáta v observeri správne zobrazili v GUI
- LiveData - ViewModel má priamo informáciu o konkrétnom view v .xml layout file, kde sa majú dáta zobrazit' a refreshovať

Externé data:

- Repository slúži na dáta externých zdrojov
- je to vrstva, ktorá zakrýva pôvod, protokol dát



DataBinding

(build.gradle)

Neobjavuje, skopírujte do build.gradle

```
plugins {  
    . . .  
    id 'kotlin-android-extensions'  
    id 'kotlin-kapt' ←  
}  
android {  
    buildFeatures {  
        viewBinding = true  
        dataBinding = true  
    }  
}  
dependencies {  
    annotationProcessor  
        "com.android.databinding:compiler:$kotlin_version"  
    . . .  
}  
kapt {  
    generateStubs = true  
}
```

Kotlin annotation processor

Vysvetlenie: .xml súbor bude obsahovať nové anotácie, ktoré musí niekto prepojiť s kódom

DataBinding

(fragment.xml)

```
<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>
        <variable
            name="myViewModel"
            type="com.example.jetpack3.ui.main.MainViewModel" />
        </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/main"
        tools:context=".ui.main.MainFragment">

        <EditText
            android:text="@={myViewModel.inputCurrencyAmount}"
            android:hint="@string/input_currency_amount"/>

        <EditText
            android:id="@+id/outputAmount"
            android:text="@{String.valueOf(myViewModel.outputCurrencyAmount)}"
            android:text="@{safeUnbox(myViewModel.outputCurrencyAmount) == 0.0 ? "" :
                String.valueOf(safeUnbox(myViewModel.outputCurrencyAmount))}" />

        <Button
            android:id="@+id/convertBtn"
            android:onClick="@{() -> myViewModel.convertValue()}" />

        <RadioGroup">
            <RadioButton
                android:id="@+id/usd2euro"
                android:checked="@={myViewModel.usd2euroChecked}" />
            <RadioButton
                android:id="@+id/euro2usd"
                android:checked="@={myViewModel.euro2usdChecked}" />
        </RadioGroup>

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```



DataBinding

(fragment.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:android="http://schemas.android.com/apk/res/android">
    <androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:id="@+id/main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.main.MainFragment">
        ...
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```



Data Binding

previazanie .xml komponentu s LiveData premennou

Binding Expression má tvar `"@{ ... }"`

Jednosmerná väzba `@{ .. }`

- napr. Button, má zavolať zodpovedajúcu metódu pre `onClick` listener
`android:onClick="@{ () -> myViewModel.convertValue() }"`

- hodnota z LiveData premennej sa má automaticky zobrazit' vo View
`android:text="@{String.valueOf(myViewModel.outputCurrency) }"`

warning:

- `myViewModel.outputCurrency.getValue()` is a boxed field but needs to be un-boxed to execute `String.valueOf(viewModel.outputCurrency.getValue())`.

```
android:text='@{safeUnbox(myViewModel.outputCurrencyAmount) == 0.0 ?"":  
String.valueOf(safeUnbox(myViewModel.outputCurrencyAmount)) }'
```

Dvojsmerná väzba `@={ .. }`

napr. EditText môže zmeniť `MutableLiveData<>`, a tiež naopak

```
android:text="@={myViewModel.inputCurrencyAmount}"
```

DataBinding

(verzia 3 – databindings)

```
class MainFragment : Fragment() { // v roli View previaže .xml a ViewModel
    private lateinit var viewModel: MainViewModel
    lateinit var binding : FragmentMainBinding
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?): View {

        binding = DataBindingUtil.inflate(inflater,
            R.layout.fragment_main, container, false)
        binding.lifecycleOwner = this
        return binding.root
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
        binding.setVariable(myViewModel, viewModel)
    }
}
```

```
<data>
    <variable
        name="myViewModel"
        type="com.example.jetpack3.ui.main.MainViewModel" />
</data>
```


Pros:
Neriešite observera ani observables
LiveData to porieši za vás

DataBinding

(verzia 3 – databindings)

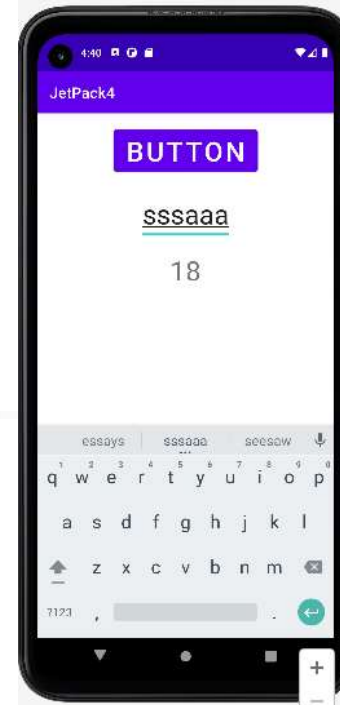
```
class MainViewModel : ViewModel() {
    val dolar2euroRate = 0.95f
    var usd2euroChecked : MutableLiveData<Boolean> = MutableLiveData()
    var euro2usdChecked : MutableLiveData<Boolean> = MutableLiveData()
    var inputCurrencyAmount : MutableLiveData<String> = MutableLiveData()
    var outputCurrencyAmount : MutableLiveData<Float> = MutableLiveData()

    fun convertValue() {
        inputCurrencyAmount.let {
            if ((it.value?:"").isEmpty()) {
                if (usd2euroChecked.value?:false)
                    //outputCurrencyAmount.value=it.value?.toFloat()?.times(dolar2euroRate)
                    outputCurrencyAmount.value = (it.value?:"0").toFloat() *
                                                    dolar2euroRate
                else
                    //outputCurrencyAmount.value=it.value?.toFloat()?.div(dolar2euroRate)
                    outputCurrencyAmount.value = (it.value?:"0").toFloat() / dolar2euroRate
            } else {
                outputCurrencyAmount.value = 0f
            }
        }
    }
}
```

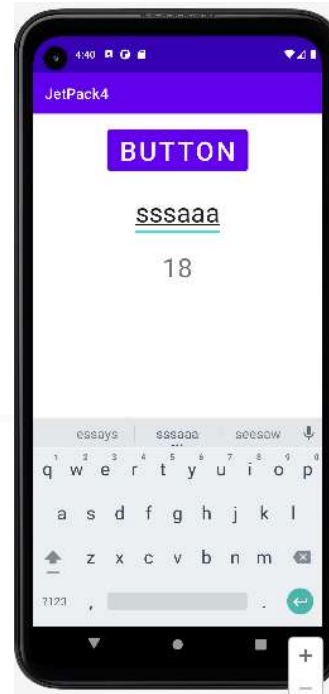
Lifecycle a LiveData

Hodne zjednodušený fragment_main.xml

```
<layout>
    <data>
        <variable
            name = "myViewModel"
            type = "com.example.jetpack4.ui.main.MainViewModel" />
    </data>
    <androidx.constraintlayout.widget.ConstraintLayout>
        <Button
            android:onClick="@{() -> myViewModel.buttonClicked()}">
        <EditText
            android:text="@={myViewModel.edittext}"
        />
        <TextView
            android:text="@{myViewModel.elapsedTime}"
        />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```



Lifecycle a LiveData



Synchrónna a asynchrónna zmena

```
class MainViewModel : ViewModel() {
    var editText : MutableLiveData<String> = MutableLiveData("")
    var _elapsedTime = 0
    var elapsedTime:MutableLiveData<String> = MutableLiveData()
    init {
        object : CountDownTimer(100*1000, 1000) {
            override fun onTick(p0: Long) {
                _elapsedTime++
                elapsedTime.value = _elapsedTime.toString()
            }
        }.start()
    }
    fun buttonClicked() {
        Log.d(TAG, "button clicked")
        editText.value += "a"
    }
}
```

Pros:

Dáta nevlastní View ale ViewModel v premennej LiveData, tá nepodlieha životnému cyklu, napr. pri zmene orientácie sa zachová jej hodnota

ViewModel SavedState



```
import androidx.lifecycle.SavedStateHandle

const val ELAPSEDTIMEKEY = "elapstime"
const val EDITBOXKEY = "editboxkey"

//class MainViewModel : ViewModel() {
class MainViewModel(private val savedStateHandle: SavedStateHandle)
    : ViewModel() {
    var edittext : MutableLiveData<String> =
        savedStateHandle.getLiveData(EDITBOXKEY)
    var elapsedTime : MutableLiveData<Int> =
        savedStateHandle.getLiveData(ELAPSEDTIMEKEY)
    object : CountdownTimer(100*1000, 1000) {
        override fun onTick(p0: Long) {
            elapsedTime.value = (elapsedTime.value?:0)+1
            savedStateHandle.set(ELAPSEDTIMEKEY, elapsedTime.value)
        }
    }.start()

    fun buttonClicked() {
        edittext.value += "a"
        savedStateHandle.set(EDITBOXKEY, edittext.value)
    }
}
```

Pros:

SaveDataHandle vám zachová dáta,
je to key-value mapa a pracuje sa s
ňou podobne ako s Bundle



SavedStateHandle

SavedStateHandle je wrapper pre `onSaveInstanceState`, a má aj rovnakú životnosť

- kým je aplikácia na obrazovke, `SavedStateHandle` si drží hodnoty,
- ak je aplikácia odstránená z obrazovky, `SavedStateHandle` si nepamätá hodnoty
 - ak ich treba, `SharedPreferences`, databáza, ...

// vo fragmente

```
val factory = SavedStateViewModelFactory(it, this)
viewModel = ViewModelProvider(this, factory)
    .get(MainViewModel::class.java)
```

// v aplikácii

```
viewModel = ViewModelProvider
    .AndroidViewModelFactory
    .getInstance(mainactivity.application)
    .create(MainViewModel::class.java)
```

```
dependencies {
    implementation "androidx.savedstate:savedstate:1.2.1"
    implementation "androidx.lifecycle:lifecycle-viewmodel-savedstate:2.6.2"
```



Pikas MVVM

ViewModel

```
class PikaViewModel: ViewModel() {  
    val index : MutableLiveData<Int> = MutableLiveData()  
    val time : MutableLiveData<Int> = MutableLiveData()  
    val currentImg: MutableLiveData<Drawable> = MutableLiveData()  
    val finish: MutableLiveData<Boolean> = MutableLiveData()  
    var list = mutableListOf<Drawable>()  
    init {  
        index.value = 0  
        time.value = 0  
        list = mutableListOf<Drawable>()  
        object : CountdownTimer(20000,1000) {  
            override fun onTick(p0: Long) {  
                time.value =(time.value?:0)+1  
            }  
            override fun onFinish() {  
                finish.value = true  
            }  
        }.start()  
    }  
}
```



Pikas MVVM

```
class MainFragment : Fragment() {
    private lateinit var viewModel: PikaViewModel
    lateinit var binding : FragmentMainBinding
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View {
        binding = DataBindingUtil.inflate(inflater,
                                           R.layout.fragment_main, container, false)
        binding.setLifecycleOwner(this)
        return binding.root
    }
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = ViewModelProvider(this).get(PikaViewModel::class.java)
        binding.setVariable(pikaViewModel, viewModel)
        viewModel.addDrawables(Repository.allDrawables(requireContext()))
        val finishObserver = Observer<Boolean> {
            result -> activity?.finish()
        }
        viewModel.finish.observe(viewLifecycleOwner, finishObserver)
    }
}
```



Pikas MVVM

```
<?xml version="1.0" encoding="utf-8"?>

<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable name="pikaViewModel"
            type="com.example.pikatchumvvm.PikaViewModel" />
    </data>
    ...
    <Button
        android:onClick="@{() -> pikaViewModel.prevValue()}"
    ...
    <Button
        android:onClick="@{() -> pikaViewModel.nextValue()}"
    ...
    <TextView
        android:text="@{pikaViewModel.time.toString()}"
    <ImageView
        android:drawable="@{pikaViewModel.currentImg}"
</layout>
```


Navigácia

Navigácia

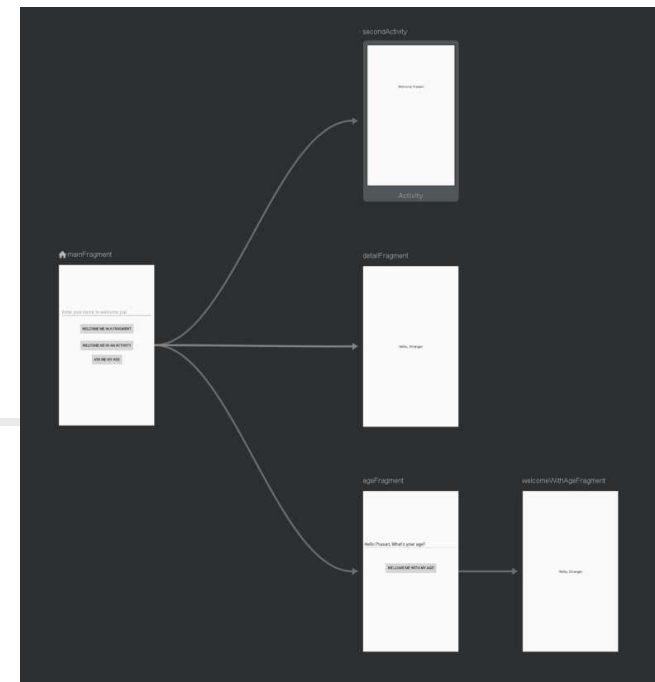
- uľahčuje programovanie prechodov-prepínanie medzi fragmentami
- používa navigačný zásobník: ak opúšťame fragment, tak sa uloží na zásobník, a vrátime sa k nemu jednoducho pomocou Back tlačidla
- jeden fragment je koreňom/host-om 🏠 navigácie **NavHostFragment**,
- Back v ňom znamená koniec aplikácie
- **NavHostFragment** popisuje prechodový graf aplikácie

<FrameLayout

...

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/fragmentContainerView"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:defaultNavHost="true"
    app:navGraph="@navigation/navig_graph" />
```

</FrameLayout>



Navigačný graf

design

host → mainFragment

atribúty akcie →

animácie →

argumenty →

akcia →

Attributes

- id: action_mainFragment_to_blankF...
- destination: blankFragment
- Animations
 - enterAnim: android:anim/slide_in_left
 - exitAnim: android:anim/slide_out_right
 - popEnterAnim: _default_pop_enter_animation
 - popExitAnim: iv_default_pop_exit_animation
- Argument Default Values
 - message: string, default value
 - size: integer, default value
- Pop Behavior
 - popUpTo: [dropdown]
 - popUpToInclusive: [checkbox]
- Launch Options
 - launchSingleTop: [checkbox]

Component Tree

- navig_graph - navigation
 - mainFragment - fragment
 - action_mainFragment_to_blankF...
 - blankFragment - fragment

Navigačný graf

res/navigation/navig_graph.xml

```
<navigation
    android:id="@+id/navig_graph"
    app:startDestination="@id/mainFragment">

    <fragment
        android:id="@+id/mainFragment"
        android:name="com.example.jetpacknav.ui.main.MainFragment"
        android:label="fragment_main"
        tools:layout="@layout/fragment_main" >
        <action
            android:id="@+id/action_mainFragment_to_blankFragment"
            app:destination="@id/blankFragment"
            app:enterAnim="@android:anim/slide_in_left"
            app:exitAnim="@android:anim/slide_out_right"
            app:popEnterAnim="@anim/nav_default_pop_enter_anim"
            app:popExitAnim="@anim/nav_default_pop_exit_anim" />
        </fragment>
        <fragment
            android:id="@+id/blankFragment"
            android:name="com.example.jetpacknav.BlankFragment"
            android:label="fragment_blank"
            tools:layout="@layout/fragment_blank" >
            <argument
                android:name="message"
                app:argType="string"
                android:defaultValue="empty" />
            <argument
                android:name="size"
                app:argType="integer" />
            </fragment>
        </navigation>
```

host

akcia

atribúty akcie

animácie

argumenty

Navigácia

kód

■ source

```
button.setOnClickListener {  
    val action = MainFragmentDirections.  
        actionMainFragmentToBlankFragment(  
            R.id.action_mainFragment_to_blankFragment)  
    action.setMessage(sourceText.text.toString())  
    action.setSize(44)  
    Navigation.findNavController(it).navigate(action)  
}
```

akcia

argumenty

■ destination

```
override fun onStart() {  
    super.onStart()  
    arguments?.let {  
        var args = BlankFragmentArgs.fromBundle(it)  
        textView.text = args.message  
        textView.textSize = args.size.toFloat()  
    }  
}
```

argumenty

Navigation Controller

- v Listeneri nejakého View

```
button.setOnClickListener {  
    val action = ...  
    Navigation.findNavController(it).navigate(action)  
}
```

← it je button: View

- vo Fragmente

```
override fun onActivityCreated(savedInstanceState: Bundle?) {  
    super.onActivityCreated(savedInstanceState)  
    viewModel = ViewModelProvider(this).get(MainViewModel::class.java)  
    val action = ...  
    val controller = Navigation.findNavController(button) ← View  
    ... alebo  
    val controller = NavHostFragment.findNavController(this) ← Fragment  
    controller.navigate(action)  
}
```

- v Aktivite

```
val action = ...  
val controller = Navigation.findNavController(activity,  
    R.id.fragmentContainerView)  
controller.navigate(R.id.action_mainFragment_to_blankFragment)
```



Navigation setup

graf navigácie

- použite Fragment+ViewModel template

- do build.gradle(Module) pridajte

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
    id 'androidx.navigation.safeargs'  
}  
  
dependencies {  
    implementation 'androidx.navigation:navigation-fragment-ktx:2.5.3'  
    implementation 'androidx.navigation:navigation-ui-ktx:2.5.3'}
```

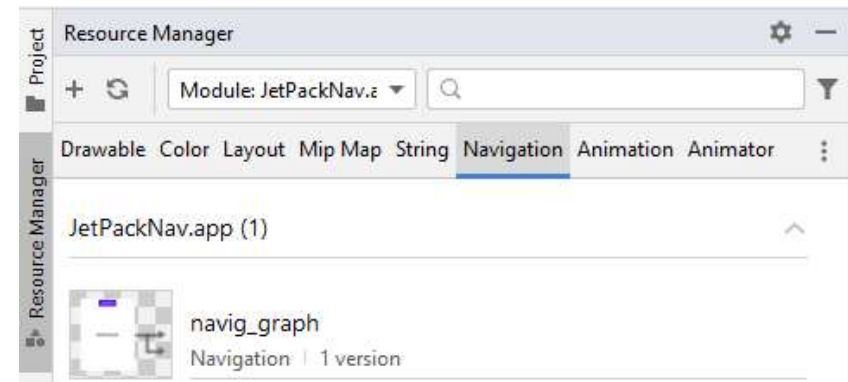
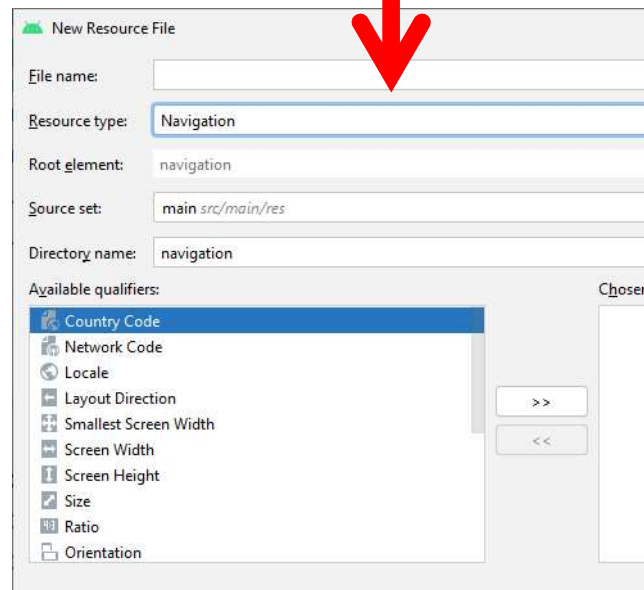
- do build.gradle(Project) pridajte

```
dependencies {  
    classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.5.3"}
```

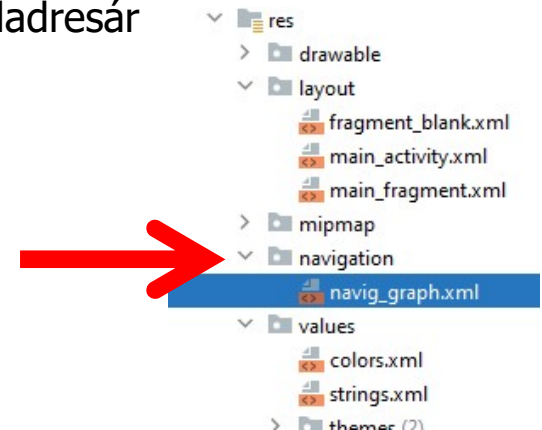
Navigation setup

graf navigácie

- vytvorte navigačný graf v resource adresári
 - resource manageri
- New/Add Android Resource File/ →
Resource Type = Navigation



vytvorí sa vám podadresár
navigation



Navigation setup

navigačný host

- do res/layout/main_activity.xml umiestnite component **androidx.navigation.fragment.NavHostFragment**
- a previažte ho na váš navigačný graf
- **app:defaultNavHost="true"**

```
<FrameLayout
```

```
...
```

```
tools:context=".MainActivity" >
```

```
<androidx.fragment.app.FragmentContainerView
```

```
    android:id="@+id/fragmentContainerView"
```

```
    android:name="androidx.navigation.fragment.NavHostFragment"
```

```
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
```

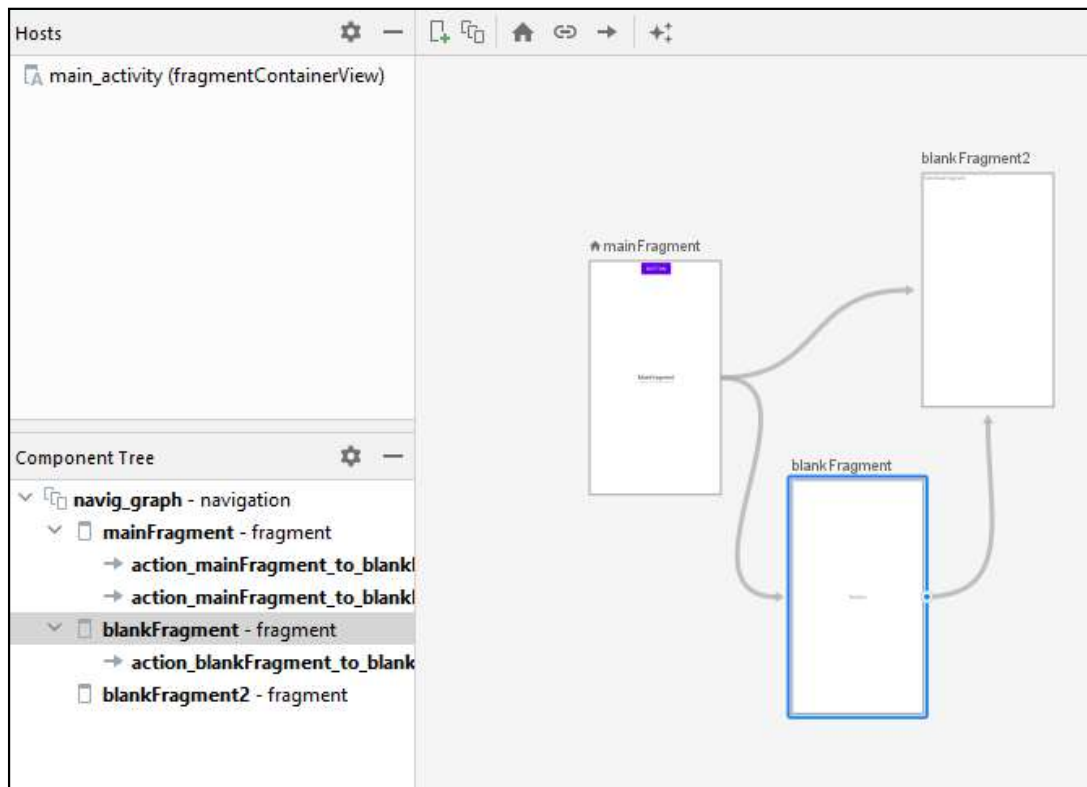
```
    app:defaultNavHost="true"
```

```
    app:navGraph="@navigation/navig_graph" />
```

```
</FrameLayout>
```


Navigation setup

vytvorenie navigačného grafu, pridanie akcií (prechody)



Attributes

→ action action_mainFragment_to_blankFragment

id 1Fragment_to_blankFragment

destination blankFragment

▼ Animations

enterAnim @android:anim/slide_in_left

exitAnim @android:anim/slide_out_right

popEnterAnim /nav_default_pop_enter_anim

popExitAnim n/nav_default_pop_exit_anim

▼ Argument Default Values

message	string	default value
size	integer	default value

▼ Pop Behavior

popUpTo

popUpToInclusive

▼ Launch Options

launchSingleTop

```
Navigation.findNavController(it)
    .navigate(R.id.action_mainFragment_to_blankFragment)
```

Bottom Navigation Menu

(asi na cvičení)

- Bottom NavigationView je MD component, ktorý potrebuje menu
`<com.google.android.material.bottomnavigation.BottomNavigationView`

```
...  
app:menu="@menu/bottom_nav_menu" />
```

- previazať s fragmentami, ktoré sa zobrazia

```
val navController = findNavController(R.id.nav_host_fragment_activity_main)  
val appBarConfiguration = AppBarConfiguration(  
    setOf(  
        R.id.navigation_home,  
        R.id.navigation_dashboard,  
        R.id.navigation_notifications  
    )  
)  
setupActionBarWithNavController(navController, appBarConfiguration)  
navView.setupWithNavController(navController)
```

