ViewModel et LiveData

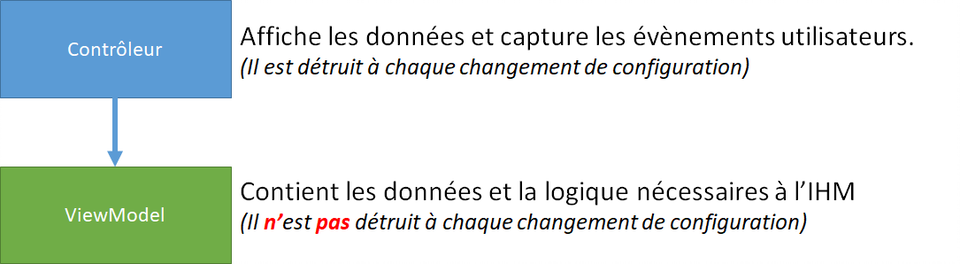
1. Objectifs du chapitre

* Comprendre ViewModel
* Comprendre LiveData

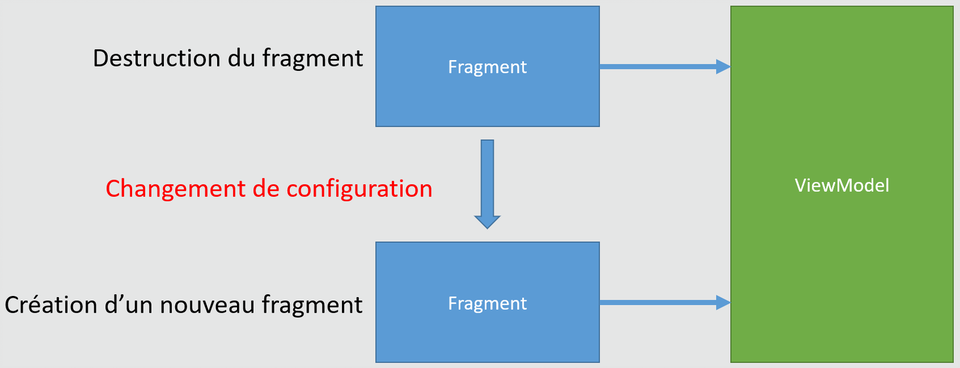
1. ViewModel
   1. Présentation

Il est recommandé de créer des applications respectant le modèle MVC. Le modèle d’une application MVC représente l’état de l’application, ainsi que la logique métier ou les opérations à effectuer. La logique métier doit être encapsulée dans le modèle, ainsi que toute autre logique d’implémentation pour la persistance de l’état de l’application. En général, les vues fortement typées utilisent des types ViewModel conçus pour contenir les données à afficher sur une vue. Le contrôleur crée et remplit ces instances de ViewModel à partir du modèle.

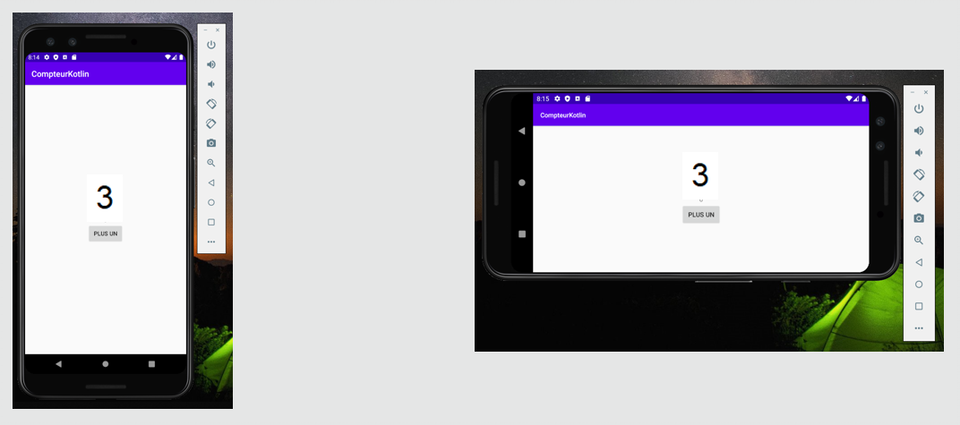
Donc un objet ViewModel fournit les données d'un composant d'interface utilisateur spécifique, tel qu'un fragment ou une activité, et contient une logique métier de gestion des données pour communiquer avec le modèle. Par exemple, le ViewModel peut appeler d'autres composants pour charger les données et il peut transmettre les demandes des utilisateurs pour modifier les données. Le ViewModel ne connaît pas les composants de l'interface utilisateur, il n'est donc pas affecté par les modifications de configuration, telles que la recréation d'une activité lors de la rotation de l'appareil.



En effet, à chaque fois qu’il y a un changement de configuration, le fragment qui a le focus est détruit et un nouveau est créé cela signifie que l’on perd toutes nos informations c’est pourquoi la classe ViewModel est conçue pour stocker et gérer les données liées à l'interface utilisateur en tenant compte du cycle de vie. La classe ViewModel permet aux données de survivre aux changements de configuration tels que les rotations d'écran.



Une des plus-values de cette architecture est de ne pas perdre d'information lors d'un changement de configuration.



1. Démonstration simple d'utilisation du ViewModel avec une activité (non chargé dans la vue)

ViewModel

Le ViewModel contient les données à afficher ainsi que la logique métier.

package fr.eni.viewmodellivedatademonstration.view\_model

​

import android.util.Log

import androidx.lifecycle.MutableLiveData

import androidx.lifecycle.ViewModel

​

class CompteurViewModel : ViewModel() {

​

//Données devant être affichées dans la vue

var compteur = 0;

//Logique métier

fun plusUn() {

compteur++

}

}

Contrôleur

Le contrôleur créé le ViewModel et réceptionne les actions utilisateurs afin de mettre à jour la vue.

Pour utiliser ViewModelProviders les deux dépendances ci-dessous doivent être utilisées :

implementation "android.arch.lifecycle:extensions:1.1.1"

implementation "android.arch.lifecycle:viewmodel:1.1.1"

Sources du contrôleur :

class MainActivity : AppCompatActivity() {

​

var compteur = 0;

​

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

​

var vb = ActivityMainBinding.inflate(layoutInflater)

setContentView(vb.root)

​

var vm = ViewModelProviders.of(this).get(MainActivityViewModel::class.java)

​

vb.tvValeur.text = vm.compteur.toString()

​

vb.btnCompteur.setOnClickListener {

vm.plusUn()

vb.tvValeur.text = vm.compteur.toString()

}

}

}

1. Démonstration simple d'utilisation du ViewModel avec un fragment (non chargé dans la vue)

ViewModel

Le ViewModel contient les données à afficher ainsi que la logique métier.

package fr.eni.viewmodellivedatademonstration.view\_model

​

import android.util.Log

import androidx.lifecycle.MutableLiveData

import androidx.lifecycle.ViewModel

​

class CompteurViewModel : ViewModel() {

​

//Données devant être affichées dans la vue

var compteur = 0;

//Logique métier

fun plusUn() {

compteur++

}

}

Contrôleur

Le contrôleur créé le ViewModel et réceptionne les actions utilisateurs afin de mettre à jour la vue.

package fr.eni.viewmodellivedatademonstration.fragments

​

import android.os.Bundle

import androidx.fragment.app.Fragment

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import androidx.databinding.DataBindingUtil

import androidx.lifecycle.ViewModelProvider

import fr.eni.viewmodellivedatademonstration.R

import fr.eni.viewmodellivedatademonstration.databinding.FragmentCompteurBinding

import fr.eni.viewmodellivedatademonstration.view\_model.CompteurViewModel

​

/\*\*

\* Contrôleur du compteur

\*/

class CompteurFragment : Fragment() {

//Permet d'accéder à tout les éléments de la vue.

private lateinit var binding: FragmentCompteurBinding

//Permet d'accéder à toutes les actions et informations necessaire à l'ihm.

private lateinit var viewModel: CompteurViewModel

​

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

//Instance de la classe de liaison(binding) permettant de mettre à jour n'importe quel élément de la vue.

binding = DataBindingUtil.inflate<FragmentCompteurBinding>(inflater, R.layout.fragment\_compteur, container, false)

​

//Instance de la classe viewModel permettant de fournir toutes les infos pour la vue.

//Si il y a une nouvelle configuration de l'affichage, le fragment sera détruit mais l'instance du viewModel persistera.

//Cette ligne est ici la clé de compréhension de l'amelioration apportait par les viewModel.

viewModel = ViewModelProvider(this).get(CompteurViewModel::class.java)

​

//On prend des infos du view model pour les mettre dans la vue via l'instance de binding.

binding.tvValeur.text = viewModel.compteur.toString()

//On lit des actions du viewModel pour les mettre dans la vue via l'instance de binding.

binding.btnCompteur.setOnClickListener{

viewModel.plusUn()

binding.tvValeur.text = viewModel.compteur.toString()

}

return binding.root

}

}

Vue

<?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">

<androidx.constraintlayout.widget.ConstraintLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView

android:id="@+id/tv\_valeur"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="0"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent" />

​

<Button

android:id="@+id/btn\_compteur"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Plus un"

app:layout\_constraintLeft\_toLeftOf="@id/tv\_valeur"

app:layout\_constraintRight\_toRightOf="@id/tv\_valeur"

app:layout\_constraintTop\_toBottomOf="@id/tv\_valeur" />

</androidx.constraintlayout.widget.ConstraintLayout>

</layout>

1. Démonstration simple d'utilisation du ViewModel (Chargé dans la vue)

ViewModel

package fr.eni.viewmodellivedatademonstration.view\_model

​

import android.util.Log

import androidx.lifecycle.MutableLiveData

import androidx.lifecycle.ViewModel

​

class CompteurViewModel : ViewModel() {

​

//Données devant être affichées dans la vue

var compteur = 0;

//Logique métier

fun plusUn() {

compteur++

}

}

Contrôleur

package fr.eni.viewmodellivedatademonstration.fragments

import android.os.Bundle

import androidx.fragment.app.Fragment

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import androidx.databinding.DataBindingUtil

import androidx.lifecycle.ViewModelProvider

import fr.eni.viewmodellivedatademonstration.R

import fr.eni.viewmodellivedatademonstration.databinding.FragmentCompteurBinding

import fr.eni.viewmodellivedatademonstration.view\_model.CompteurViewModel

•

/\*\*

\* Contrôleur du compteur

\*/

class CompteurFragment : Fragment() {

//Permet d'accéder à tout les éléments de la vue.

private lateinit var binding: FragmentCompteurBinding

//Permet d'accéder à toutes les actions et informations necessaire à l'ihm.

private lateinit var viewModel: CompteurViewModel

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

//Instance de la classe de liaison(binding) permettant de mettre à jour n'importe quel élément de la vue.

binding = DataBindingUtil.inflate<FragmentCompteurBinding>(inflater, R.layout.fragment\_compteur, container, false)

//Instance de la classe viewModel permettant de fournir toutes les infos pour la vue.

//Si il y a une nouvelle configuration de l'affichage, le fragment sera détruit mais l'instance du viewModel persistera.

//Cette ligne est ici la clé de compréhension de l'amelioration apportait par les viewModel.

viewModel = ViewModelProvider(this).get(CompteurViewModel::class.java)

binding.vm = viewModel

binding.btnCompteur.setOnClickListener{

viewModel.plusUn()

binding.vm = viewModel

}

return binding.root

}

}

Vue

<?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">

​ <data>

<variable

name="vm"

type="fr.eni.viewmodellivedatademonstration.view\_model.CompteurViewModel"

/>

</data>

​

<androidx.constraintlayout.widget.ConstraintLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView

android:id="@+id/tv\_valeur"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="@{String.valueOf(vm.compteur)}"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent" />

​

<Button

android:id="@+id/btn\_compteur"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Plus un"

app:layout\_constraintLeft\_toLeftOf="@id/tv\_valeur"

app:layout\_constraintRight\_toRightOf="@id/tv\_valeur"

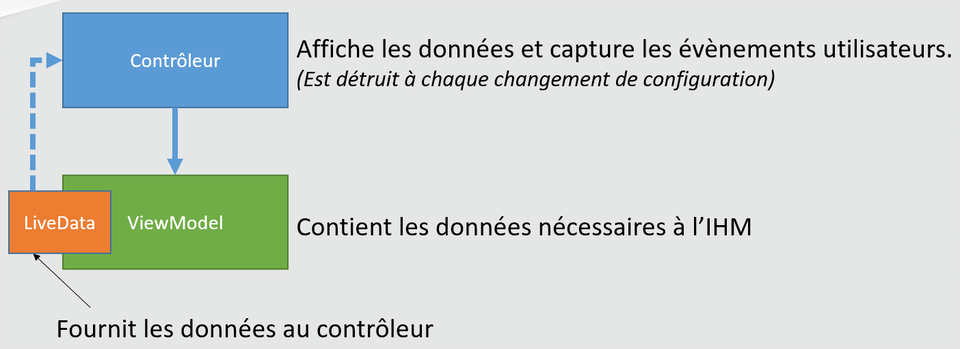
app:layout\_constraintTop\_toBottomOf="@id/tv\_valeur" />

</androidx.constraintlayout.widget.ConstraintLayout>

</layout>

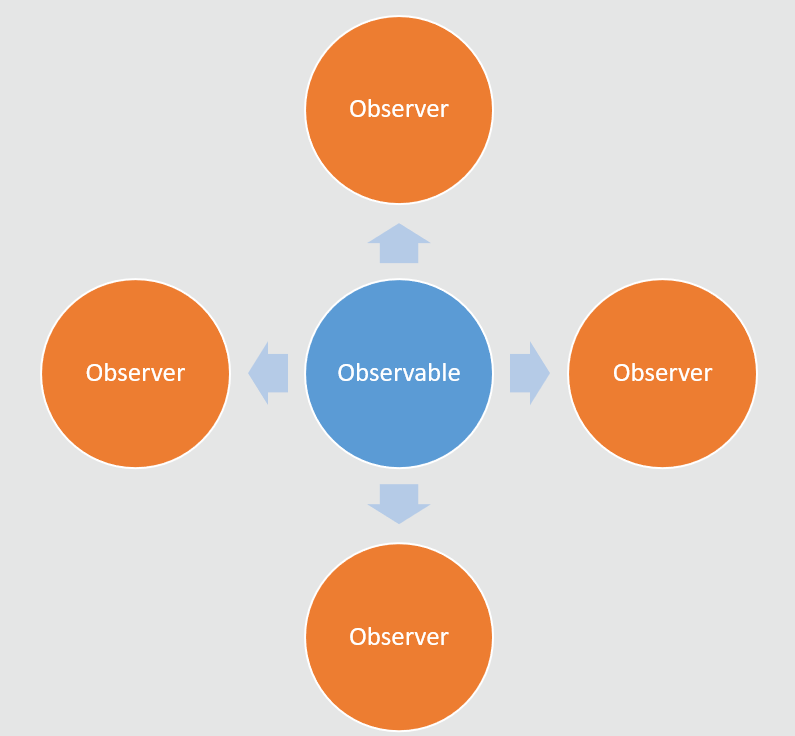
1. LiveData
   1. Présentation

LiveData est une classe permettant de mettre le design pattern observer/observable en place dans une application Android. Contrairement à un observable classique, LiveData est sensible au cycle de vie, ce qui signifie qu'il respecte le cycle de vie des autres composants de l'application, tels que les activités, les fragments ou les services. Cette prise de conscience garantit que LiveData ne met à jour que les observateurs de composants d'application qui sont dans un état de cycle de vie actif. ​​​​​​​



Observer / Observable

En gros quand certaines données sont mise à jour dans le modèle cela enclenche l’execution de certaines fonctionnalités dans le contrôleur dans le but de mettre à jour l’IHM automatiquement.



1. Démonstration simple du ViewModel avec LiveData

Dans le code ci-dessous on observe que l'IHM est étroitement liée au ViewModel au niveau de l'affichage des données et des actions liées aux boutons.

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

<layout>

<data>

<variable

name="Model"

type="fr.eni.compteurkotlin" />

</data>

​

<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

xmlns:app="http://schemas.android.com/apk/res-auto"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

tools:context=".fragments.CompteurFragment">

​

<TextView

android:id="@+id/tv\_compteur"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="@{String.valueOf(Model.compteur)}"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent" />

​

<Button

android:id="@+id/btn\_plus\_un"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Plus un"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintEnd\_toEndOf="parent"

app:layout\_constraintStart\_toStartOf="parent"

app:layout\_constraintTop\_toBottomOf="@+id/tv\_compteur"

android:onClick="@{() -> Model.plusUn()}"

/>

​

</androidx.constraintlayout.widget.ConstraintLayout>

</layout>

Dans le code ci-dessous, on observe qu'une donnée est observé.

package fr.eni.compteurkotlin

•

import android.util.Log

import androidx.lifecycle.MutableLiveData

import androidx.lifecycle.ViewModel

•

class CompteurViewModel : ViewModel() {

•

//Donnée observée

val compteur = MutableLiveData<Int>()

•

init {

compteur.value = 0

}

•

fun plusUn() {

compteur.value = (compteur.value)?.plus(1)

}

}

Dans le code ci-dessous, on observe que lorsque la donnée observée est mise à jour alors on renvoie le ViewModel à jour à l'IHM.

package fr.eni.compteurkotlin

import android.os.Bundle

import android.util.Log

import androidx.fragment.app.Fragment

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import androidx.databinding.DataBindingUtil

import androidx.lifecycle.Observer

import androidx.lifecycle.ViewModelProvider

import fr.eni.compteurkotlin.databinding.FragmentCompteurBinding

class CompteurFragment : Fragment() {

private lateinit var viewModel:CompteurViewModel

private lateinit var binding: FragmentCompteurBinding

override fun onCreateView(

inflater: LayoutInflater, container: ViewGroup?,

savedInstanceState: Bundle?

): View? {

binding = DataBindingUtil.inflate<FragmentCompteurBinding>(inflater, R.layout.fragment\_compteur, container, false)

viewModel = ViewModelProvider(this).get(CompteurViewModel::class.java)

//Fonction executée quand la donnée observée est mise à jour

viewModel.compteur.observe(viewLifecycleOwner, Observer { valeur ->

binding.model = viewModel

})

return binding.root

}

}

1. Conclusion

Vous avez saisi l’utilité des ViewModel (On persiste les données entre deux configurations)

Vous avez saisi l’utilité de LiveData (On met à jour l'IHM automatiquement)

Vous savez mettre en place une architecture avec ViewModel et LiveData