TP3 Architecture d'application

Il est temps de mettre un peu d'ordre et de découvrir plus en détail les composants d'architecture d'Android. Nous allons retrouver la séparation des préoccupations dans ce TP afin que chaque élément joue le rôle auquel il doit être affecté.

1 Les ViewModels

1.1 Rappel

Le ViewModel est la partie qui contient les données utilisées par la vue (activité ou fragment) auquel il est associé. C'est lui qui joue le rôle de la prise de décision. On y retrouvera donc les données à afficher mais également les méthodes de comportements de nos vues.

1.2 Nos premiers ViewModels

On reprend notre TP 2 pour poursuivre nos améliorations et comme à chaque fois, il faut commencer par ajouter des dépendances :

```
    implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
```

On va maintenant changer un peu l'arborescence en créant un package `viewmodel` dans lequel on va créer la classe `IdentityViewModel`.

```
class IdentityViewModel : ViewModel()
{
    init {
        Log.i("IdentityViewModel", "created")

    }

    override fun onCleared() {
        super.onCleared()
        Log.i("IdentityViewModel", "destroyed")
    }
}
```

Puis déclarer la variable qui le lie à notre fragment `IdentityFragment`.

```
private lateinit var viewModel: IdentityViewModel
```

Quelque chose d'assez important : les fragments sont de ces éléments qui sont recréés lorsque, par exemple, on effectue une rotation de l'écran. Mais pour rappel, le ViewModel survie à cela. Pour ce faire, il faut toujours utiliser `ViewModelProvider` pour initialiser notre ViewModel !!

Tant que le fragment ne sera pas détâché, le ViewModel existera. Il est créé la première fois, lorsqu'on l'attache au fragment. On ajoutera alors dans `onCreateView()` :

```
// viewModel = ViewModelProviders.of(this).get(IdentityViewModel::class.java) //
depreciate
viewModel =
ViewModelProvider(this,viewModelFactory).get(IdentityViewModel::class.java)
```

Avec les logs insérés, on verra lorsque notre ViewModel est créé ou détruit.

Désormais, on se doit de remplir notre ViewModel. Il ne doit contenir AUCUNE dépendance vers nos vues !!

Donc on se doit de déplacer une partie de notre code actuel du Fragment au sein du ViewModel et notamment l'initialisation du user.

```
var user: User = User("Doe", "John") // déplacé depuis le fragment
```

Néanmoins, si on souhaite transmettre des données au ViewModel, on est un peu bloqué, car c'est le constructeur vide qui est appelé à chaque fois. Alors comment faire ? Utiliser le `ViewModelFactory`.

Créons alors notre `PersonalDataViewModel` pour le second fragment et son ViewModelFactory dans un package `viewmodelfactory`.

```
class PersonalDataViewModel(userParam: User) : ViewModel()
{
    var user = userParam

    init {
        Log.i("PersonalDataViewModel", "created")
    }

    fun onGender(gender: String) {
        user?.gender = gender
    }
}
```

```
class PersonalDataViewModelFactory(private val user: User) :
ViewModelProvider.Factory {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(PersonalDataViewModel::class.java)) {
            return PersonalDataViewModel(user) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }

}
```

Bien évidemment, il faut changer notre fragment avec la déclaration du ViewModel et du ViewModelFacory

```
class PersonalDataFragment : Fragment(), PersonalDateEventListener {
    private lateinit var binding: FragmentPersonalDataBinding
    private lateinit var viewModel: PersonalDataViewModel // Variable du ViewModel
    private lateinit var viewModelFactory: PersonalDataViewModelFactory //
Variable du ViewModelFactory

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = DataBindingUtil.inflate(inflater,
R.layout.fragment_personal_data, container, false)
        binding.eventListener = this

        val args = PersonalDataFragmentArgs.fromBundle(requireArguments())
        viewModelFactory = PersonalDataViewModelFactory(args.user) //
initialisation du Factory
        viewModel =
ViewModelProvider(this,viewModelFactory).get(PersonalDataViewModel::class.java)//
Initialisation du ViewModel

       // viewModel = ViewModelProviders.of(this, viewModelFactory)
         //   .get(PersonalDataViewModel::class.java)
         // depreciate

        binding.user = viewModel.user // Modification de l'initialisation du
binding

        binding.apply {
            tvTitle.text = user?.firstname.plus(" ").plus(user?.lastname)
            evBirthday.hint = getString(R.string.birthdayDate)
            btValidate.text = getString(R.string.validate)
        }
        binding.btValidate.setOnClickListener {
            validate(it)
        }

        return binding.root
    }

    override fun onGender(gender: String) {
        viewModel.onGender(gender) // Désormais on appelle le ViewModel
    }

    private fun validate(view: View) {
        val message = viewModel.user.gender + " " +
LongConverter.dateToString(viewModel.user.birthdayDate) // Ici également
        Toast.makeText(this.context, message, Toast.LENGTH_SHORT).show()
    }
}
```

Voilà, nous avons déjà séparé notre code métier de notre code de vue.

En résumé :

- Les vues gèrent uniquement les intéractions entre l'utilisateur et l'interface. Les données sont gérées par le `ViewModel`
- L'interface `ViewModelProvider.Factory` permet de créer les `ViewModel`

1.3 S'observer avec LiveData

`LiveData` est un wrapper qui permet de mettre en place la méthode Observer-Observable. Qui plus est, cette classe suit le cycle de vues de l'application (activité, fragment...). Dans ce cas, seulement les composants actifs (Started ou Resumed) à l'écran seront mis à jour.

On va commencer par changer notre code dans notre classe `IdentityViewModel`. On va également jouer sur l'encapsulation des données. C'est à dire qu'on va limiter l'accès à nos objets seulement en lecture depuis l'extérieure de notre ViewModel. On utilisera alors :

- `MutableLiveData` pour les objets modifiables qui sont dans le ViewModel
- `LiveData` pour les objets accessibles seulement en lecture

Dans notre ViewModel, on utilisera alors la valeur privée et mutable.

```
private val _user = MutableLiveData<User>()
val user: LiveData<User>
    get() = _user

init {
    Log.i("IdentityViewModel", "created")

    _user.value = User("Doe", "John")
}
```

Et côté fragment, on va attacher notre variable avec un `Observer`. Automatiquement, notre variable va être mise à jour.

```
viewModel.user.observe(this, Observer { user ->
    binding.user = user
})
```

On va aussi devoir modifier la variable dans la fonction `validate()`. On utilise le `.value` pour accéder à la valeur. Néanmoins cette valeur peut-être `null` et il faut donc effectuer un test (sachant qu'on a choisi de ne pas pouvoir passer de valeur nullable au fragment suivant).

```
view.findNavController().navigate(IdentityFragmentDirections.actionIdentityFragmen
tToPersonalDataFragment(viewModel.user.value?:User()))
```

A vous désormais de faire les mises à jour dans le second fragment.

Ah oui, mais en fait il serait plus judicieux de réaliser le binding avec le `ViewModel` et non plus avec le `User`. Cela nous permettrait de supprimer l'`EventListner` de notre deuxième fragment en liant directement les événements aux fonctions. Pas mal non ?

On en revient donc à ceci :

```
<?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="viewModel"
                type="com.example.tp2.viewmodel.IdentityViewModel" />
    </data>


    <androidx.cardview.widget.CardView
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

    <androidx.constraintlayout.widget.ConstraintLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_margin="6sp">

        <TextView
                android:id="@+id/tv_title"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textAppearance="@style/TextAppearance.AppCompat.Body1"
                android:textSize="32sp"
                android:layout_marginBottom="16sp"
                app:layout_constraintTop_toTopOf="parent"
                app:layout_constraintRight_toRightOf="parent"
                app:layout_constraintLeft_toLeftOf="parent"/>

        <com.google.android.material.textfield.TextInputLayout
                android:id="@+id/ev_lastname"
                android:layout_width="0dp"
                android:layout_height="wrap_content"
                app:layout_constraintTop_toBottomOf="@id/tv_title"
                app:layout_constraintLeft_toLeftOf="parent"
                app:layout_constraintRight_toRightOf="@id/tv_guideline">

            <com.google.android.material.textfield.TextInputEditText
                    android:id="@+id/ti_lastname"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:text="@={viewModel.user.lastname}"
                    android:inputType="text"/>

        </com.google.android.material.textfield.TextInputLayout>

        <com.google.android.material.textfield.TextInputLayout
```

```
                android:id="@+id/ev_firstname"
                android:layout_width="0dp"
                android:layout_height="wrap_content"
                app:layout_constraintTop_toBottomOf="@id/tv_title"
                app:layout_constraintRight_toRightOf="parent"
                app:layout_constraintLeft_toLeftOf="@id/tv_guideline">

            <com.google.android.material.textfield.TextInputEditText
                    android:id="@+id/ti_firstname"
                    android:layout_width="match_parent"
                    android:layout_height="wrap_content"
                    android:text="@={viewModel.user.firstname}"
                    android:inputType="text"/>

        </com.google.android.material.textfield.TextInputLayout>

        <com.google.android.material.button.MaterialButton
                android:id="@+id/bt_validate"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                app:layout_constraintTop_toBottomOf="@id/ev_firstname"
                app:layout_constraintRight_toRightOf="parent"/>

        <androidx.constraintlayout.widget.Guideline
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:id="@+id/tv_guideline"
                android:orientation="vertical"
                android:layout_margin="2sp"
                app:layout_constraintGuide_percent="0.5"/>

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

```
class IdentityFragment : Fragment() {

    private lateinit var binding: FragmentIdentityBinding
    private lateinit var viewModel: IdentityViewModel

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = DataBindingUtil.inflate(inflater, R.layout.fragment_identity,
container, false)

        viewModel = ViewModelProviders.of(this).get(IdentityViewModel::class.java)
        binding.viewModel = viewModel

//        viewModel.user.observe(this, Observer { user ->
//            binding.user = user
//        })

        binding.apply {
            tvTitle.text = getString(R.string.title)
            tiFirstname.hint = getString(R.string.firstname)
            tiLastname.hint = getString(R.string.lastname)
            btValidate.text = getString(R.string.validate)
        }

        binding.btValidate.setOnClickListener {
            validate(it)
        }

        return binding.root
    }

    private fun validate(view: View) {
        var t = 0

view.findNavController().navigate(IdentityFragmentDirections.actionIdentityFragmen
tToPersonalDataFragment(viewModel.user.value?:User()))
    }

}
```

```
<?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>
        <import type="com.example.tp2.LongConverter"/>
        <variable
                name="viewModel"
                type="com.example.tp2.viewmodel.PersonalDataViewModel" />
    </data>

    <androidx.cardview.widget.CardView
            android:layout_width="match_parent"
            android:layout_height="wrap_content">

        <androidx.constraintlayout.widget.ConstraintLayout
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_margin="6sp">

            <TextView
                    android:id="@+id/tv_title"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_marginBottom="16sp"
                    android:textAppearance="@style/TextAppearance.AppCompat.Body1"
                    android:textSize="32sp"
                    app:layout_constraintLeft_toLeftOf="parent"
                    app:layout_constraintRight_toRightOf="parent"
                    app:layout_constraintTop_toTopOf="parent" />

            <RadioGroup
                    android:id="@+id/rg_gender"
                    android:layout_width="0dp"
                    android:layout_height="wrap_content"
                    app:layout_constraintLeft_toLeftOf="parent"
                    app:layout_constraintRight_toRightOf="@id/tv_guideline"
                    app:layout_constraintTop_toBottomOf="@id/tv_title">

                <RadioButton
                        android:id="@+id/rb_woman"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content"

android:checked="@{viewModel.user.gender.equals(@string/woman)}"
                        android:onClick="@{() ->
viewModel.onGender(@string/woman)}"
                        android:text="@string/woman" />

                <RadioButton
                        android:id="@+id/rb_man"
```

```
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content"

android:checked="@{viewModel.user.gender.equals(@string/man)}"
                        android:onClick="@{() -> viewModel.onGender(@string/man)}"
                        android:text="@string/man" />
            </RadioGroup>

            <com.google.android.material.textfield.TextInputLayout
                    android:id="@+id/ev_birthday"
                    android:layout_width="0dp"
                    android:layout_height="wrap_content"
                    app:layout_constraintLeft_toLeftOf="@id/tv_guideline"
                    app:layout_constraintRight_toRightOf="parent"
                    app:layout_constraintTop_toBottomOf="@id/tv_title">

                <com.google.android.material.textfield.TextInputEditText
                        android:id="@+id/ti_birthday"
                        android:layout_width="match_parent"
                        android:layout_height="wrap_content"
                        android:inputType="date"
                        android:text="@=
{LongConverter.dateToString(viewModel.user.birthdayDate)}"
                    />
            </com.google.android.material.textfield.TextInputLayout>

            <com.google.android.material.button.MaterialButton
                    android:id="@+id/bt_validate"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    app:layout_constraintRight_toRightOf="parent"
                    app:layout_constraintTop_toBottomOf="@id/ev_birthday" />

            <androidx.constraintlayout.widget.Guideline
                    android:id="@+id/tv_guideline"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_margin="2sp"
                    android:orientation="vertical"
                    app:layout_constraintGuide_percent="0.5" />

        </androidx.constraintlayout.widget.ConstraintLayout>
    </androidx.cardview.widget.CardView>

</layout>
```

```
class PersonalDataFragment : Fragment() { //, PersonalDateEventListener {
    private lateinit var binding: FragmentPersonalDataBinding
    private lateinit var viewModel: PersonalDataViewModel
    private lateinit var viewModelFactory: PersonalDataViewModelFactory
```

```
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = DataBindingUtil.inflate(inflater,
R.layout.fragment_personal_data, container, false)
//        binding.eventListener = this

        val args = PersonalDataFragmentArgs.fromBundle(requireArguments())
        viewModelFactory = PersonalDataViewModelFactory(args.user)
        viewModel =
ViewModelProvider(this,viewModelFactory).get(PersonalDataViewModel::class.java)

        //viewModel = ViewModelProviders.of(this, viewModelFactory)
        //    .get(PersonalDataViewModel::class.java)
        // depreciate

        binding.viewModel = viewModel

        viewModel.user.observe(this, Observer { user ->
            binding.tvTitle.text = user.firstname.plus(" ").plus(user.lastname)
        })

        binding.apply {
            evBirthday.hint = getString(R.string.birthdayDate)
            btValidate.text = getString(R.string.validate)
        }
        binding.btValidate.setOnClickListener {
            validate(it)
        }

        return binding.root
    }

//    override fun onGender(gender: String) {
//        viewModel.onGender(gender)
//    }

    private fun validate(view: View) {
        val message = viewModel.user.value?.gender + " " +
LongConverter.dateToString(viewModel.user.value?.birthdayDate?:0)
        Toast.makeText(this.context, message, Toast.LENGTH_SHORT).show()
    }
}
```

On gagne quand même un petit de code et on se facilite la vie.

Ajoutez `binding.lifecycleOwner = this` au sein des deux `onCreateView()` des fragments pour que le binding fonctionne avec les `LiveData`. Cela permettra d'effectuer notamment les mises à jour visuelles sans que vous ayez à passer par une fonction par le `observe`. Dans notre cas, on verrait seulement l'utilité pour la mise à jour du nom et prénom dans le second fragment.

Vous pouvez donc désormais supprimer ce code dans le second fragment :

```
viewModel.user.observe(this, Observer { user ->
    binding.tvTitle.text = user.firstname.plus(" ").plus(user.lastname)
})
```

Et ajouter une propriété sur le TextView de titre

```
<TextView
    android:id="@+id/tv_title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16sp"
    android:textAppearance="@style/TextAppearance.AppCompat.Body1"
    android:textSize="32sp"
    android:text="@{viewModel.user.firstname + ` ` +viewModel.user.lastname}"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

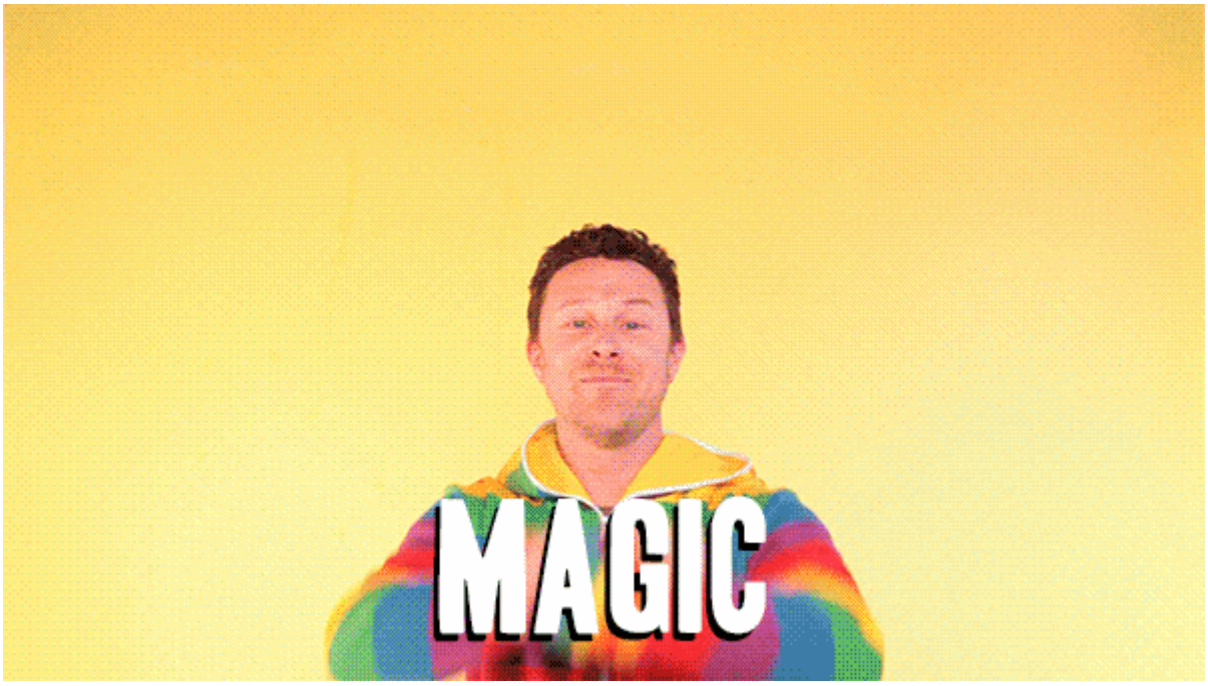
Mais essayez d'ajouter ceci également la propriété `android:text="@{viewModel.user.firstname + ` ` +viewModel.user.lastname}"` pour le TextView du premier fragment. Vous verrez que si vous modifiez les données du nom, le titre ne change pas. En fait, il faut aussi rendre l'objet observable.

Ainsi, si vous modifiez votre modèle en lui ajoutant la notification des modificiations de propriétés, vous aurez un affichage des modifications en temps réel. Si des erreurs apparaissent notamment avec les variables associées à `BR`, cliquez sur le petit marteau qui va rafraichir les données. **Attention, pour l'import de la classe de data binding BR, il faudra inscrire le nom de votre package puis BR - exemple : `import com.example.tp2.BR`**

```
import androidx.databinding.BaseObservable
import androidx.databinding.Bindable
import com.example.tp2.BR


@Keep
data class User(private var _lastname: String? = "", private var _firstname:
String? = "", private var _birthdayDate: Long = 0, private var _gender: String? =
"") : Parcelable,
    BaseObservable() {

    var lastname: String?
        @Bindable get() = _lastname
        set(value) {
            _lastname = value
            notifyPropertyChanged(BR.lastname)
        }

    var firstname: String?
        @Bindable get() = _firstname
        set(value) {
            _firstname = value
            notifyPropertyChanged(BR.firstname)
        }

    var birthdayDate: Long
        @Bindable get() = _birthdayDate
        set(value) {
            _birthdayDate = value
            notifyPropertyChanged(BR.birthdayDate)
        }

    var gender: String?
        @Bindable get() = _gender
        set(value) {
            _gender = value
            notifyPropertyChanged(BR.gender)
        }

    constructor(parcel: Parcel) : this(
        parcel.readString(),
        parcel.readString(),
        parcel.readLong(),
        parcel.readString()
    )

    override fun writeToParcel(parcel: Parcel, flags: Int) {
        parcel.writeString(lastname)
        parcel.writeString(firstname)
        parcel.writeLong(birthdayDate)
        parcel.writeString(gender)
    }
```

```
    override fun describeContents(): Int {
        return 0
    }

    companion object CREATOR : Parcelable.Creator<User> {
        override fun createFromParcel(parcel: Parcel): User {
            return User(parcel)
        }

        override fun newArray(size: Int): Array<User?> {
            return arrayOfNulls(size)
        }
    }
}
```

Comme par magie, si vous tapez votre nom et votre prénom, il sera mis à jour en temps réel ! Et les plus heureux pourront s'écrier JAVASCRIIIPT !



En résumé :

- `LiveData` est une classe de détenteur de données observables et sensible au cycle de vie de l'application
- `LiveData` est observable, ce qui signifie qu'un observateur, comme une activité ou un fragment, peut être averti lorsque les données changent
- `LiveData` met à jour les valeurs que quand les observateur sont actifs
- `MutableLiveData` est un objet `LiveData` dont la valeur peut être modifiée
- Il est préférable d'encapsuler le `LiveData` et donc de rendre privée la variable de type `MutableLiveData` qui est modifiable
- Un `ViewModel` peut être associé à la liaison de données

2 Les données

Pour terminer ce TP, nous allons voir comment créer et utiliser une base de données ainsi que comment appeler un Web Service.

2.1 Room

Sous Android, les données sont représentées dans des classes de données et les données sont accessibles et modifiées à l'aide d'appels de fonction. Les requêtes n'existent donc pas et ceci c'est grâce à `Room`. Cette librairie fait le travail à notre place et permet une écriture simplifiée.

Pour ce faire, nous allons modifier notre modèle `User` et comme d'habitude, quelques modifications du fichier gradle sont necessaires :

```
implementation "androidx.room:room-runtime:2.3.0"
kapt "androidx.room:room-compiler:2.3.0"
implementation "androidx.room:room-ktx:2.3.0"
```

Dans notre modèle `User` nous allons déjà déclarer le nom de notre table grâce aux annotations :

```
@Entity(tableName = "user")
data class User(private var _lastname: String? = "", private var _firstname:
String? = "", private var _birthdayDate: Long = 0, private var _gender: String? =
"") : Parcelable,
{
    //...
}
```

Il nous faut ensuite annoter une clé primaire et les différentes colonnes :

```
@Keep
@Entity(tableName = "user")
data class User(@ColumnInfo(name = "lastname")
                private var _lastname: String? = "",

                @ColumnInfo(name = "firstname")
                private var _firstname: String? = "",

                @ColumnInfo(name = "birthday_date")
                private var _birthdayDate: Long = 0,

                @ColumnInfo(name = "gender")
                private var _gender: String? = ""): Parcelable,
    BaseObservable() {
```

```
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    private var _id: Long = 0L
    var id: Long
        @Bindable get() = _id
        set(value) {
            _id = value
            notifyPropertyChanged(BR.id)
        }

    var lastname: String?
        @Bindable get() = _lastname
        set(value) {
            _lastname = value
            notifyPropertyChanged(BR.lastname)
        }


    var firstname: String?
        @Bindable get() = _firstname
        set(value) {
            _firstname = value
            notifyPropertyChanged(BR.firstname)
        }


    var birthdayDate: Long
        @Bindable get() = _birthdayDate
        set(value) {
            _birthdayDate = value
            notifyPropertyChanged(BR.birthdayDate)
        }

    var gender: String?
        @Bindable get() = _gender
        set(value) {
            _gender = value
            notifyPropertyChanged(BR.gender)
        }

    constructor(parcel: Parcel) : this(
        parcel.readLong(),
        parcel.readString(),
        parcel.readString(),
        parcel.readLong(),
        parcel.readString()
    )

    override fun writeToParcel(parcel: Parcel, flags: Int) {
        parcel.writeLong(id)
        parcel.writeString(lastname)
        parcel.writeString(firstname)
        parcel.writeLong(birthdayDate)
```

```
        parcel.writeString(gender)
    }

    override fun describeContents(): Int {
        return 0
    }

    companion object CREATOR : Parcelable.Creator<User> {
        override fun createFromParcel(parcel: Parcel): User {
            return User(parcel)
        }

        override fun newArray(size: Int): Array<User?> {
            return arrayOfNulls(size)
        }
    }
}
```

Vous pouvez executer le code désormais mais pensez à ajouter une valeur (0) lors de la création de votre utilisateur de base.

Pour continuer, nous devons créer un objet d'accès aux données (DAO). Il va nous permettre d'avoir des méthodes pour insérer, supprimer et mettre à jour notre base de données. Comme on le disait un peu plus haut, nous allons appeler des fonctions Kotlin pour accéder aux données qui sont en réalité mappées avec des requêtes SQL.
Des annotations simples existent pour insérer `@Insert`, supprimer `@Delete` et mettre à jour `@Update` les données. Si on doit créer des requêtes plus complexes, on utilisera l'annotation `@Query` où on devra par contre créer notre requête manuellement.

Un DAO n'est pas une classe mais une interface, car elle représente juste une couche d'abstraction et il n'y a donc pas de code à écrire. Créez-là dans un nouveau package `database`.

```
@Dao
interface UserDao
{
    @Insert
    fun insert(user: User): Long

    @Delete
    fun delete(user: User)

    @Update
    fun update(user: User)

    @Query("SELECT * from user WHERE id = :key")
    fun get(key: Long): User?
}
```

Aussi simple que cela 😎

Il ne nous reste plus qu'à créer notre base de données qui sera une classe abstraite.

```
@Database(entities = [User::class], version = 1, exportSchema = false)
abstract class Database : RoomDatabase() {}
```

Grâce à l'annotation `Database()`, on va définir les arguments permettant d'initialiser notre base de données. En premier lieu les `entities` (ici seulement la classe `User`), la `version` de notre base (on devra l'augmenter à chaque modification de la base), et l'`exportSchema` à `false` pour ne pas avoir de sauvegarde des différentes versions de base.

On doit ensuite ajouter notre DAO dans le corps de notre classe afin de lier l'accès code-base de données.

```
abstract val userDao: UserDao
```

Puis on retrouve l'objet `companion` du TP1. Au sein de celui-ci, on aura la création de l'accès à la base sous forme de singleton. Ce companion est à déclarer au sein de votre nouvelle classe abstraite `Database`

```
companion object {

        @Volatile
        private var INSTANCE: MyDatabase? = null

        fun getInstance(context: Context): MyDatabase {
            synchronized(this) {
                var instance = INSTANCE

                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        MyDatabase::class.java,
                        "my_database"
                    )
                        .fallbackToDestructiveMigration()
                        .build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
```

Quelques petites informations sur ce code :

- L'annotation `@Volatile` permet d'éviter la mise en cache de la variable. On évite le problème de multithread qui pourrait arriver et on s'assure qu'il n'y ait bien qu'une instance de notre base pour l'ensemble des threads. Surtout, on évite la mise à jour d'un tuple par deux threads simultanément.
- Avec l'encapsulation dans `synchronized(this)`, on ne donnera accès à la base de données qu'à un thread à la fois donc il ne peut y avoir qu'une initialisation de la base de données.
- On utilisera une migration simple avec `fallbackToDestructiveMigration()`. Ici, on recréera la base de données à chaque fois, donc on perdra les données à chaque exécution.

En résumé :

- Pour définir une table, on utilise l'annotation `@Entity`. Les colonnes seront elles déclarées avec l'annotation `@ColumnInfocomme`.
- Pour accéder aux données, on utilise une interface DAO qui mappe le code kotlin en requête SQLite. Des annotations basiques existent pour ce mappage : `@Insert`, `@Delete` et `@Update`. Sinon, on choisira `@Query` qui sera complété par une chaîne de requête SQLite.
- Une classe abstraite est nécessaire pour récupérer l'accès à la base de données. On y utilise un singleton synchronisé pour éviter les problèmes multithreads.

Désormais il faut lier notre base avec le reste de notre code.

2.2 Liaison des données

Auparavant, nous avions créé deux ViewModel mais en réalité, on n'en aurait besoin que d'un seul puisque nos vues ne sont pas si spécifiques que cela. On va donc commenter le code de notre `PersonalDataViewModel` et du `PersonalDataViewModelFactory`. Par contre, on va recréer un `IdentityViewModelFactory`.

```
class IdentityViewModelFactory (
    private val dataSource: UserDao,
    private val application: Application
) : ViewModelProvider.Factory {
    @Suppress("unchecked_cast")
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(IdentityViewModel::class.java)) {
            return IdentityViewModel(dataSource, application) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Ce code fonctionnera si on pense à changer le constructeur de `IdentityViewModel` :

```
class IdentityViewModel(
    val database: UserDao,
    application: Application
) : AndroidViewModel(application){

    //...
    // Pensez à recopier la fonction suivante

    fun onGender(gender: String) {
        _user.value?.gender = gender
    }
}
```

Il y a également le fragment `personal_data_fragment`.

```
<variable
    name="viewModel"
    type="com.example.tp2.viewmodel.IdentityViewModel" />
```

Notre code ne compile toujours pas car nous devons initialiser toutes nos classes. Ceci sera à réaliser au sein des deux classes Fragment.

Dans un premier temps on doit récupérer le contexte de notre application (en évitant les exceptions).

```
val application = requireNotNull(this.activity).application
```

Dans un deuxième temps, c'est le DAO qu'il faut récupérer. Si vous vous souvenez, il est lié à notre base de données :

```
val dataSource = MyDatabase.getInstance(application).userDao
```

On peut donc enfin instancier notre ViewModel (on aurait pu l'appeller UserViewModel mais on ne va pas s'amuser à changer chaque détail) :

```
val viewModelFactory = IdentityViewModelFactory(dataSource, application)
// viewModel =
//    ViewModelProviders.of(
//        this, viewModelFactory).get(IdentityViewModel::class.java)
// depreciate
viewModel =
ViewModelProvider(this,viewModelFactory).get(IdentityViewModel::class.java)
```

En fait, on n'a plus vraiment besoin de se passer l'utilisateur de fragment en fragment. Découvrons maintenant un concept spécifique : les Coroutines.

2.3 Utilisation des données & coroutines

Afin de limiter les blocages de l'écran lorsque nous allons récupérer les données, il faut utiliser le système de routines de Kotlin. Un moyen efficace donc pour les tâches qui peuvent être de longue durée. Normalement on appelle donc une routine qui exécute la tâche puis se termine avec un callbacks.

Kotlin propose un système de coroutine qui permet donc d'effectuer des tâches longues durées asynchrones et non bloquantes : Documentation coroutines. En somme, une coroutine suspend le code où il en était et reprend lorsqu'il obtient le résultat.

Comme à chaque fois, on doit déclarer quelques petites librairies :

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.8"
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.0"
```

Pour commencer, nous devons déclarer un `Job`. Celui-ci nous permettra d'annuler toutes les routines lancées par le ViewModel lorsque celui-ci ne sera plus utilisé et sera détruit. On annulera les coroutines lorsque le ViewModel sera nettoyé. Enfin, on définit un scope qui nous permettra de savoir sur quel thread on execute nos coroutines. Ici, `Dispatchers.Main` signifie que les coroutines seront exécutées sur le thread principal.

```
class IdentityViewModel(
    val database: UserDao,
    application: Application
) : AndroidViewModel(application)
{
    private var viewModelJob = Job()
    private val uiScope = CoroutineScope(Dispatchers.Main + viewModelJob)

    //...

    override fun onCleared() {
        super.onCleared()
        Log.i("IdentityViewModel", "destroyed")
        viewModelJob.cancel()
    }
}
```

Nous allons désormais modifier un peu l'initialisation de notre utilisateur. On va soit récupérer le dernier utilisateur de la base soit en créer un :

```
init {
    Log.i("IdentityViewModel", "created")
    initializeUser()
}

private fun initializeUser() {
    uiScope.launch {
        _user.value = getUserFromDatabase()
    }
}

private suspend fun getUserFromDatabase(): User? {
    return withContext(Dispatchers.IO) {

        var user = database.getLastUser()
        if (user == null) {
            user = User()
            user.id = insert(user)
        }
        user
    }
}

private suspend fun insert(user: User): Long {
    var id = 0L
    withContext(Dispatchers.IO) {
        id = database.insert(user)
    }
```

```
    return id
}
```

On lance toujours une coroutine dans le scope, d'où la récuparation de la valeur à l'intérieur du bloc `uiScope.launch`.
La coroutine `getUserFromDatabase()` retour un objet `User` qui peut être `null`. On utilise le mot clé `suspend` évite de bloquer le code jusqu'à ce que le résultat soit retourné. Elle suspend donc le code et reprend ensuite là où elle s'était arrêtée, avec le résultat. Pendant que la coroutine est suspendue et attend un résultat, elle débloque le fil sur lequel elle est exécutée. De cette façon, d'autres fonctions ou coroutines peuvent s'exécuter.
On exécute la coroutine sur le thread `Dispatchers.IO` qui gère les entrées/sorties. Ce thread n'est pas utilisé pour la gestion de l'interface utilisateur.

Au sein du DAO, il faudra ajouter cette fonction pour récupérer le dernier utilisateur.

```
@Query("SELECT * FROM user ORDER BY id DESC LIMIT 1")
fun getLastUser(): User?
```

`Notez bien que, généralement, on sauvegarde l'utilisateur à la validation. Mais dans cet exemple, on essaie d'utiliser le plus les fonctions disponibles.`

Ensuite, on va déclarer quelques autres fonctions, dont celle qui sauvegardera les modifications utilisateurs en base de données.

```
fun onValidate() {
    uiScope.launch {
        val user = user.value ?: return@launch
        update(user)
    }
}

private suspend fun update(user: User) {
    withContext(Dispatchers.IO) {
        database.update(user)
    }
}

private suspend fun get(id: Long) {
    withContext(Dispatchers.IO) {
        database.get(id)
    }
}
```

La fonction `onValidate()` sera appelée à la fin de notre formulaire. Il faut donc ajouter un lien au sein de notre `personal_data_fragment` :

```
android:onClick="@{() -> viewModel.onValidate()}"
```

Alors qu'au sein de notre `IdentityFragment` on fera ainsi :

```
private fun validate(view: View) {
    viewModel.onValidate()

view.findNavController().navigate(IdentityFragmentDirections.actionIdentityFragmen
tToPersonalDataFragment(viewModel.user.value?:User()))
}
```

En résumé :

- La liaison des données entre la base et l'IHM se fait grâce aux `ViewModel` et `ViewModelFactory`.
- Les coroutines sont asynchrones et non bloquantes. Elles utilisent des fonctions `suspend` pour rendre le code asynchrone séquentiel (c'est à dire, on attend le résultat sans bloquer l'éxécution mais en le suspendant).
- La différence entre bloquer et suspendre est que si un thread est bloqué, aucun autre travail ne sera exécuté. Si le thread est suspendu, d'autres tâches sont exécutées jusqu'à ce que le résultat soit disponible.

2.4 Navigation et sécurité IHM

On se doit de sécuriser un peu notre IHM en bloquant les boutons lorsque les données ne sont pas remplies dans le formulaire. Mais surtout, notre code fonctionne mais l'utilisateur n'est plus récupéré entre chaque fragment. On doit donc encore faire quelques modifications.

On doit d'abord introduire un `userID` en paramètre de notre `ViewModel` et `ViewModelFactory` et modifier la récupération en base.

```
class IdentityViewModel(
    val database: UserDao,
    application: Application,
    private val userID: Long = 0L // userID
) : AndroidViewModel(application)
{
    //...

     private suspend fun getUserFromDatabase(): User? {
        return withContext(Dispatchers.IO) {

            var user = database.get(userID) // userID
            if (user == null) {
                user = User()
                user.id = insert(user)
            }
            user
        }
    }
}
```

```
class IdentityViewModelFactory (
    private val dataSource: UserDao,
    private val application: Application,
    private val userID: Long = 0L // userID
) : ViewModelProvider.Factory {
    @Suppress("unchecked_cast")
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(IdentityViewModel::class.java)) {
            return IdentityViewModel(dataSource, application,userID) as T //
userID
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Puis, on améliore un peu le passage de données entre les deux fragments. Cela va nous servir à bloquer le bouton avant que les données ne soit complétées. Pour notre cas, on va continuer à passer l'utilisateur entre les deux fragments, mais en réalité, on passe uniquement l'id de l'objet.

Pour ce qui est du ViewModel, on déclare donc une nouvelle variable seulement disponible en lecture et une fonction qui réinitialise la valeur. Enfin, on a une fonction qui va mettre à jour l'utilisateur juste pour le cas du premier fragment puis déclancher le fait que l'utilisateur est bien rempli.

```
private val _navigateToPersonalDataFragment = MutableLiveData<User>()

val navigateToPersonalDataFragment: LiveData<User>
    get() = _navigateToPersonalDataFragment

fun doneNavigating() {
    _navigateToPersonalDataFragment.value = null
}

fun onValidateIdentity() {
    uiScope.launch {
        val user = user.value ?: return@launch

        if(user.firstname.isNullOrEmpty())
            return@launch

        if(user.lastname.isNullOrEmpty())
            return@launch

        update(user)

        _navigateToPersonalDataFragment.value = user
    }
}
```

```
android:onClick="@{() -> viewModel.onValidateIdentity()}"
```

Au niveau du Fragment, on supprime le code du `onValidate`. On va observer notre nouvelle variable qu'on déclarera dans `onCreateView()`

```
// Code qui remplace la fonction onValidate()
viewModel.navigateToPersonalDataFragment.observe(viewLifecycleOwner, Observer {
user ->
    user?.let {
        this.findNavController().navigate(
            IdentityFragmentDirections
            .actionIdentityFragmentToPersonalDataFragment(user))

        viewModel.doneNavigating()
    }
})
```

Ces changements sont effectués car le code métier doit être uniquement dans le ViewModel.

On peut faire la même modification pour le second fragment pour la fonction `onValidate()` :

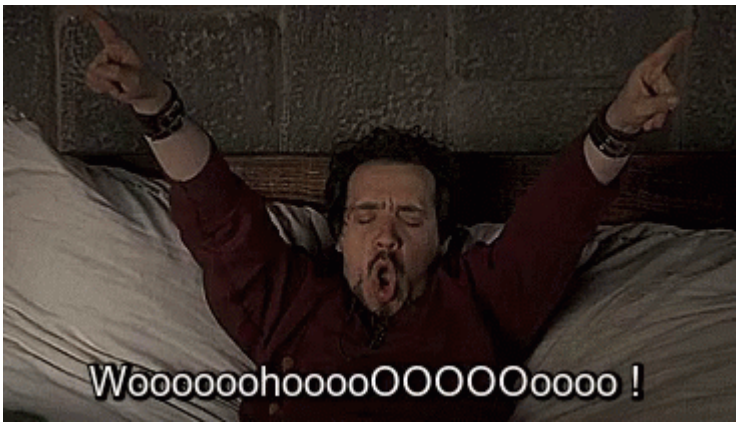
```
private val _navigateToOtherActivity = MutableLiveData<User>()

val navigateToOtherActivity: LiveData<User>
    get() = _navigateToOtherActivity

fun doneValidateNavigating() {
    _navigateToOtherActivity.value = null
}

fun onValidate() {
    uiScope.launch {
        val user = user.value ?: return@launch

        if(user.gender.isNullOrEmpty())
            return@launch

        update(user)

        _navigateToOtherActivity.value = user
    }
}
```

```
// Code qui remplace la fonction onValidate()
viewModel.navigateToOtherActivity.observe(viewLifecycleOwner, Observer { user ->
    user?.let {
        val message = viewModel.user.value?.gender + " " +
LongConverter.dateToString(viewModel.user.value?.birthdayDate?:0)
        Toast.makeText(this.context, message, Toast.LENGTH_SHORT).show()

        viewModel.doneValidateNavigating()
    }
 })
```

Nous avons désormais terminé la partie base de données.

Pour en terminer rendez-vous ici : https://codelabs.developers.google.com/codelabs/kotlin-android-training-room-database/index.html?index=..%2F..android-kotlin-fundamentals#0

Le but est de découvrir un autre aspect de la base de données en réalisant les parties 6.1, 6.2 et 6.3.


Wooooooh oooo OOOOO oooo !