MVVM

Poupart Valentin (FA)
06/12/2018

Résumé

Dans les développement de tous les jours, on peut facilement écrire beaucoup de lignes de codes dans un fichier. Mais au jour d'aujourd'hui il nous arrive beaucoup de travailler en équipe et pour la lisibilité et les bonnes pratiques il existe des « design pattern » pour nous guider.

Dans ce TP, nous verrons comment utiliser et mettre en place le design pattern MVVM. Ce dernier permet de simplifier et d'organiser l'écriture de vos interfaces graphiques.

Pré-requis

Les pré-requis nécessaire pour ce TP :

- *Savoir programmer une application Android*
- *Savoir créer un layout*
- *Savoir coder un minimum en Java*
- *Connaître les notions de data-binding*

Code source

Code source **initial** : https://github.com/ValentinPoupart/MVVM

Code source **final** : https://github.com/ValentinPoupart/MVVM_finish

Explications du TP

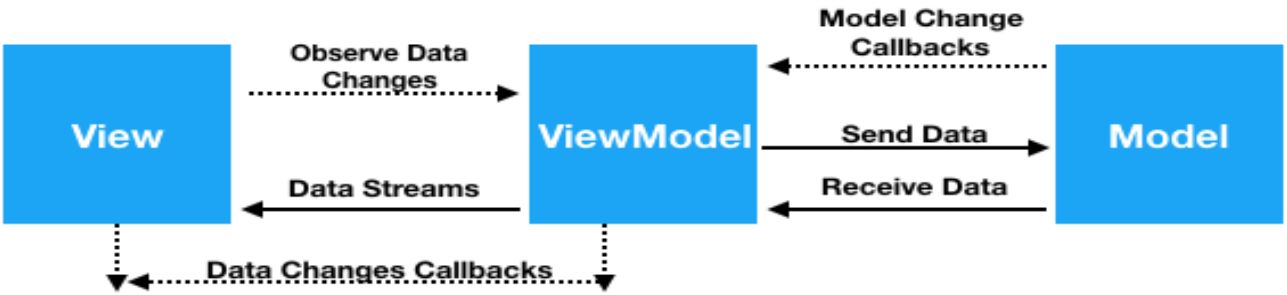
Dans ce tp nous allons simuler que l'on puisse envoyer notre commande depuis notre smartphone dans un restaurant. Nous allons uniquement travailler sur l'affichage d'un formulaire en utilisant le design pattern MVVM et data-binding.

Avant tout parlons un peu de ce design pattern.

- Model: Cela tient les données de l'application. Il ne peut pas directement parler à la Vue. Généralement, il est recommandé d'exposer les données au ViewModel par Observables.

- View: Il représente l'UI de l'application exempté de n'importe quelle logique d'application (calcul, ..). Il observe le ViewModel.

- ViewModel: Il agit comme un lien entre le Model et la View. Sa responsabilité est de transformer les données du Modèle. Il fournit des flux de données à la View. Il utilise aussi des des rappels de service (callbacks) pour mettre à jour la View.



Etape 1: paramétrer gradle pour le tp

Pour ce tp nous utilisons du data-binding, pour cela il nous faudra rajouter du code dans le fichier build.gradle.

Voici le contenu du fichier (vérifiez la version de votre SDK) :

```
apply plugin: 'com.android.application'

android {
    dataBinding {
        enabled = true
    }
    compileSdkVersion 28
    defaultConfig {
        applicationId "tuto.fr.mvvm"
        minSdkVersion 15
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-
rules.pro'
        }
    }
}

dependencies {
    implementation 'android.arch.lifecycle:extensions:1.1.0'
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-
core:3.0.2'
}
```

Etape 2: créer le model

Nous allons créer un model qui pourra correspondre à un repas, il y a une classe Repas dans le package « `tuto.fr.mvvm.model` » dans laquelle on travaillera pour cette étape.

En premier lieu il faudra ajouter les éléments que vous souhaiterez voir dans le formulaire. Pour simplifier je vous propose d’ajouter les attributs Entree, Plat et Dessert (tous des String). Cependant si vous voulez personnaliser votre model vous êtes libre de le faire.

Suite à cela on ajoutera la notation « @NonNull » à nos attributs pour dire que la valeur de retour ne sera jamais null. Ensuite il faudra créer les Getters/Setters et le constructor de la classe avec tous les attributs de la classe en paramètre.

Pour finir notre model, nous allons ajouter des méthodes de vérification, par exemple :

```
public boolean isPlatValid(){
    return !TextUtils.isEmpty(getNomPlat()) && getNomPlat().length() > 3;
}
```

Ces méthodes serviront ensuite pour vérifier notre formulaire.

Rappel vos constructeurs/getter/setter doivent ressembler à ça :

```
@NonNull
public String getNomEntree() {
    return NomEntree;
}

public void setNomEntree(@NonNull String nomEntree) {
    NomEntree = nomEntree;
}
public Repas(@NonNull String nomEntree) {
    NomEntree = nomEntree;
}
```

Etape 3: créer le layout

Le layout correspondant à l'affichage est important, l'utilisation de data-binding se retrouve ici, c'est pourquoi nous n'allons pas voir en détail la création du layout.

Vous pouvez retrouver ci dessous le « Text » de base pour le fichier activity_mvvm.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:bind="http://schemas.android.com/tools">

    <data>
        <variable
            name="viewModel"
            type="tuto.fr.mvvm.viewmodel.RepasViewModel" />
    </data>

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:layout_margin="8dp"
            android:orientation="vertical">

            <Button
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_marginTop="8dp"
                android:onClick="@{()-> viewModel.onValidationClicked()}"
                android:text="Valider"
                bind:toastMessage="@{viewModel.toastMessage}" />
        </LinearLayout>
    </ScrollView>
</layout>
```

Une fois ce texte de base renseigné il faut ajouter nos champs pour notre model.

Voici le schéma à suivre pour ces champs. Il faut bien penser à changer l'id et le hint pour chaque champs si on fait un copier coller. Il faudra également changer la méthode utilisé dans « afterTextChanged », les méthodes utilisés à cette emplacement seront écrites dans le view model.

```
<EditText
                android:id="@+id/in_entree"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:afterTextChanged="@{(editable)-
>viewModel.afterEntreeTextChanged(editable)}"
                android:hint="Entree"
                android:inputType="text"
                android:padding="8dp" />
```

Voilà notre layout est prêt !

Etape 4: créer le ViewModel

Comme vu dans le layout, il nous faudra des méthodes pour l'affichage des données quand elles sont modifiés. Avant tout il nous faudra ajouter un objet Repas dans notre classe.

Il faudra également ajouter un constructeur pour notre classe qui ne prends pas de paramètre et qui initialise notre objet Repas, à vous de voir si vous souhaitez y mettre des valeurs par défauts.

Voici le schéma à suivre pour chaque attributs de notre objet Repas :

```
public void afterEntreeTextChanged(CharSequence s) {
    repas.setNomEntree(s.toString());
}
```

Suite à cela nous avons encore à ajouter la fonction « onValidationClicked » que l'on a sur notre bouton de validation pour le formulaire dans le layout.

On va donc ajouter cette méthode mais juste avant il nous faudra ajouter ces lignes :

```
@Bindable
public String toastMessage = null;

public String getToastMessage() {
    return toastMessage;
}

private void setToastMessage(String toastMessage) {
    this.toastMessage = toastMessage;
    notifyPropertyChanged(BR.toastMessage);
}
```

Ces lignes nous permettront d'avoir un message lors de la validation.

Il ne nous reste plus qu'à ajouter notre fonction de validation qui utilise nos fonctions de validation du model et qui renvoie des messages, on peut l'écrire comme ceci :

```
private String successMessage = "Validation du repas";
private String entreeErrorMessage = "Une entrée vous ouvrirez l'appétit ..";

public void onValidationClicked() {
    if (repas.isRepasValid())
        setToastMessage(successMessage);
    else if (!repas.isEntreeValid())
        setToastMessage(entreeErrorMessage);
}
```

Dernière étape: créer la view

Ici notre view nous permettra d'initialiser le data-binding et le toastMessage et surtout de démarrer notre layout.

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ActivityMvvmBinding activityMainBinding =
            DataBindingUtil.setContentView(this, R.layout.activity_mvvm);
    activityMainBinding.setViewModel(new RepasViewModel());
    activityMainBinding.executePendingBindings();

}

@BindingAdapter({"toastMessage"})
public static void runMe(View view, String message) {
    if (message != null)
        Toast.makeText(view.getContext(), message, Toast.LENGTH_SHORT).show();
}
```

Informations complémentaires

https://www.journaldev.com/20292/android-mvvm-design-pattern#android-mvvm-example-project-structure le tutoriel ayant servi d'exemple pour l'écriture du tp

D'autres liens utilisés :

https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

https://www.youtube.com/watch?v=d1UuUTAKoi8

https://openclassrooms.com/fr/courses/26832-apprenez-a-programmer-en-java/25552-mieux-structurer-son-code-le-pattern-mvc