TP3 : Les fragments
Programmation mobile

Mohamad Satea Almallouhi - Tony Nguyen
M1 Génie Logiciel
Faculté des Sciences
Université de Montpellier.

7 avril 2024

Résumé

Durant cet exercice, nous avons vue l'utilisation des Fragments, ModelView et MutableLiveData.

Table des matières

Introduction

Dans le cadre de l'Unité d'Enseignement Programmation Mobile, nous allons voir comment utiliser des fragments et la persistance des données.

Les sections du rapport suit les exercices.

Démonstration vidéo

En ligne sur Youtube, à l'adresse URL `https://youtu.be/OuzuGsB9IC4` une démonstration vidéo de notre travail.

Installation

Vous trouvez les instructions dans le README.md

1 Fragment

Un fragment est une portion réutilisable de l'interface utilisateur de notre application. Ils sont utiles car ils sont modulaires et réutilisables.

Dans cette section, nous allons voir comment les créer, les afficher et comment les faire communiquer entre eux.

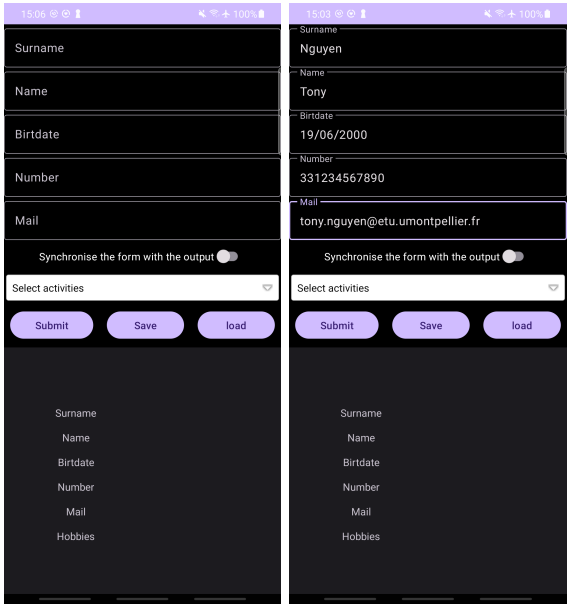
1.1 Création

Afin de créer un nouveau fragment, il nous suffit d'hériter de la classe de base Android : "*Fragment*".

Pour spécifier le comportement du nouveau fragment, il nous suffira de masquer la méthode **onCreateView()**.

Dans ce TP, nous avons créer 2 fragments, **UserInputFragment** et **DisplayFragment**.

1.2 Ajout



Pour pouvoir les afficher, nous avons créer une activity et son layout vide.

Nous avons ensuite, ajouté une balise **fragment** pour chaque fragment crée. Il faut remarquer 2 attributs important.

L'attribut **android :name** associe cette balise .xml à aux classe fragment crée dans la sous-section précédente.

L'attribut **tools :layout** associe cette balise au layout indiqué.

Nous savons à présent comment créer un fragment, et l'afficher.

```
<fragment
    android:id="@+id/userInputFragment"
    android:name="fr.umontpellier.etu.tp3_devmob.UserInputFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toTopOf="@id/guideline1"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.0"
    tools:layout="@layout/fragment_user_input" />
```

1.3 Communication entre fragment

Avant la communication, il est nécessaire de transmettre notre *ModelView* contenant les différentes MutableLiveData.

Une fois le model placé dans le *Bundle*, nous pouvons envoyer le message. Pour faire cela, nous utilisons la méthode **setFragmentResult(String,Bundle)** pour envoyer les données (contenus dans le bundle avec requestKey une chaine de charactères fixés à l'avance par le programmeur, le fragment receveur devra utiliser la même requestKey). Le message est maintenant envoyé, nous allons voir comment le recevoir.

```
model = new ViewModelProvider( owner: this).get(UserInputViewModel.class);
// data transmission
// bundle myMessage will hold the data
Bundle myMessage = new Bundle();
// setting up to transmit ViewModel with LiveData for synchronisation form/output
myMessage.putSerializable("the_model",model);
// send
// notifying fragment manager of change
getParentFragmentManager().setFragmentResult( requestKey: "Hello", myMessage);
```

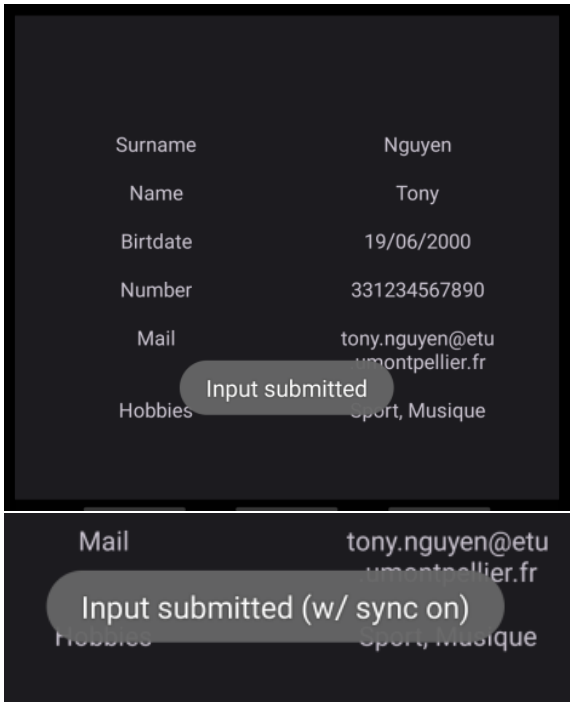
Nous allons maintenant voir comment recevoir le message.

Il sera nécessaire d'utiliser la méthode **setFragmentResultListener()** de la classe *FragmentManager*.

L'argument listener sera une classe anonyme qui hérite de *FragmentResultListener* dans laquelle on masque la méthode **onFragmentResult()**. Si nous avons bien utilisé les même requestKey, alors l'argument bundle contiendra les bonnes données. À partir de là, nos fragments possède la même instance du model.

```
getParentFragmentManager().setFragmentResultListener( requestKey: "Hello", lifecycleOwner: this,
    4 usages  tony-nguyen1
    @RequiresApi(api = Build.VERSION_CODES.TIRAMISU)
    @Override
    public void onFragmentResult(@NonNull String requestKey, @NonNull Bundle bundle) {
        Log.v( tag: "debug display", msg: "received model");

        // so both fragment source and target have same UserInputViewModel
        model = (UserInputViewModel) bundle.getSerializable( key: "the_model");
```

1.4 La synchronisation et mise à jour



Lors de la détection d'un changements, nous souhaitons publié le changement et mettre à jour la vue. Pour cela, nous allons accéder aux différents *MutableLiveData* dans notre *Model-View* : *UserInputViewModel*. Une fois fait, nous publions simplement la nouvelle valeur avec la méthode **postValue()**. Nous envoyons les valeurs lorsque l'utilisateur appuie sur "Submit" ou après qu'il ait selectioné l'option "Synchronise" puis modifie le texte.

```
// button SUBMIT
myView.findViewById(R.id.submit_button).setOnClickListener(v -> {
    Log.v( tag: "debug", msg: "posting change");
    model.get("surname").postValue(surname.getText().toString());
    model.get("name").postValue(name.getText().toString());
    model.get("birthdate").postValue(birthdate.getText().toString());
    model.get("number").postValue(number.getText().toString());
    model.get("mail").postValue(mail.getText().toString());
    model.get("hobby").postValue(hobby.getText().toString());

    this.isSynchronousWithOutput = ((SwitchMaterial) myView.findViewBy

    Toast.makeText(getContext(), text: "Input submitted" + (this.isSynch
});
```

Afin d'être notifié des plublications, il ne faut pas oublier de s'abboner aux MutableLiveData et les **observer**.

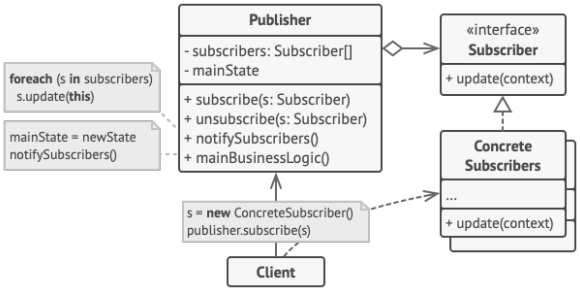
```
// Observe the LiveData, passing in this activity
// as the LifecycleOwner and the observer.
model.getCurrentSurname().observe(
        getViewLifecycleOwner(),
        DisplayFragment.this.createCustomObserver(R.id.surnameText));
model.getCurrentName().observe(
        getViewLifecycleOwner(),
        DisplayFragment.this.createCustomObserver(R.id.nameText));
model.getCurrentBirthdate().observe(
        getViewLifecycleOwner(),
        DisplayFragment.this.createCustomObserver(R.id.birthdateText));
model.getCurrentNumber().observe(
        getViewLifecycleOwner(),
        DisplayFragment.this.createCustomObserver(R.id.numberText));
model.getCurrentMail().observe(
        getViewLifecycleOwner(),
        DisplayFragment.this.createCustomObserver(R.id.mailText));
model.getCurrentHobby().observe(
        getViewLifecycleOwner(),
        DisplayFragment.this.createCustomObserver(R.id.hobbiesText));
```

Quand l'observateur est notifié, la méthode **onChanged()** est appelé et la nouvelle valeur est placé dans une $<TextView>$.

```
private Observer<String> createCustomObserver(int id) {
    tony-nguyen1
    return new Observer<String>() {
        tony-nguyen1
        @Override
        public void onChanged(@Nullable final String newName) {

            //Log.v("debug","display has observed a change");
            // Update the UI, in this case, a TextView.
            TextView myText = myView.findViewById(id);

            //Log.v("debug current display surname",myText.getTe
            //Log.v("debug display wanted surname ",myText.getTe
            myText.setText(newName);
        }
    };
}
```

1.5 Observer

Voici un rappel du diagramme de classe du patron de conception *Observateur*



Il faut remarquer que ce design pattern est essentiel dans ce tp pour implémenter la communication entre les fragments. On y retrouve les méthodes de souscriptions, notifications et mises à jour.

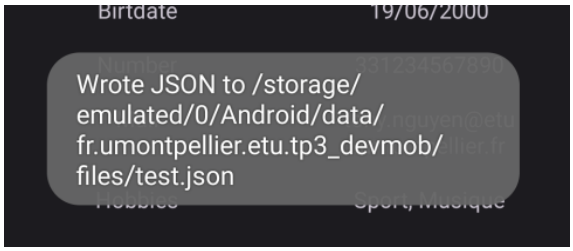
On identifie les méthodes de notifications comme **setFragmentResult()** et **postValue()**.

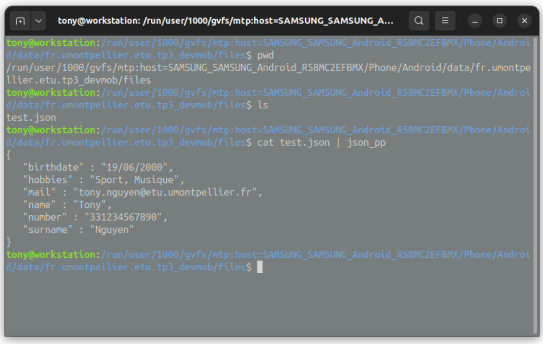
Les méthodes de mises à jour correspondent à **setFragmentResultListener()** et **observe()**.

2 Persistance

Nous allons maintenant nous intéressé à la conservation des données.

2.1 Sauvegarde





Lors d'un clique sur le boutton *"Save"*, une écriture sur le stockage local du smartphone est enclenché.

La méthode **saveJsonToFile()** se chargera de récupérer les informations intéressantes, les convertire en format JSON et d'écrire dans le fichier *test.json*.

```
// Construct your JSON object
JSONObject jsonObject = constructJsonFromData();

// Get the external files directory
File  basePath = context.getExternalFilesDir( type: null);
File file = new File(basePath , child: "test.json");
Log.d( tag: "YourFragment", msg: "Starting to write JSON to " + file.getAbsolutePath());
Toast.makeText(getContext(), text: "Wrote JSON to " + file.getAbsolutePath(), Toast.LENGTH_SHORT).show();
```

2.2 (Re-)Saisie automatique

Imaginons un petit scénario.

Un utilisateur commence à saisir ses informations et fait une mauvaise manipulation. Il a appuyé sur le boutton de retour à l'écran d'accueil (Android) du smartphone.

Quand il revient sur notre application, on aimerai qu'il ne recommence pas du début.

Échec Malheuresement, nous n'avons pas réussi à implémenter cette fonctionnalités.

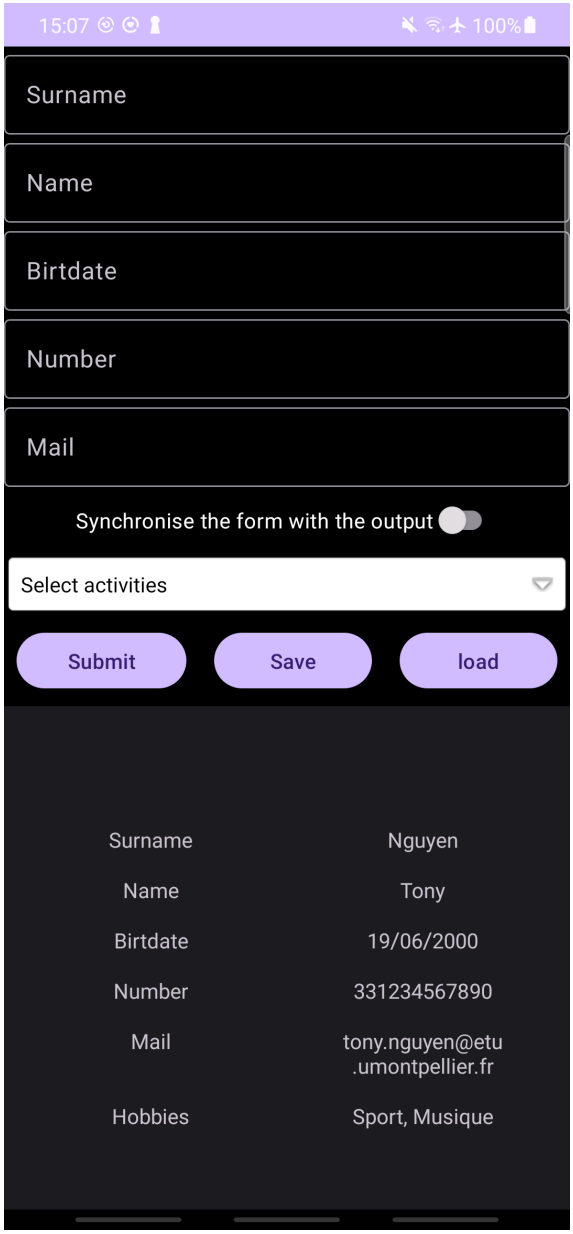
Dans le figure ci-dessous le code ne produisant *pas* une sauvegarde de l'état du formulaire quand l'utilisateur quitte temporairement l'application.

```
/********** Saving data for presestance (when we leave the app) *******/
SateaMall +1 *
@Override
public void onSaveInstanceState(@NonNull Bundle outState) {
    Log.v( tag: "debug presistancy", msg: "Starting to save...");
    super.onSaveInstanceState(outState);
    // Save the current state of your form or other data you need to preserve
    outState.putString("KEY_SURNAME", surname.getText().toString());
    outState.putString("KEY_NAME", name.getText().toString());
    outState.putString("KEY_BIRTHDATE", birthdate.getText().toString());
    outState.putString("KEY_NUMBER", number.getText().toString());
    outState.putString("KEY_MAIL", mail.getText().toString());
    outState.putString("KEY_HOBBIES", hobby.getText().toString());
    Log.v( tag: "debug presistancy", msg: "Values are saved...");
}

9 usages  SateaMall +1 *
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    // Initialize your views here if not already done
    //Log.v("debug presistancy","Recovering the values...");
    if (savedInstanceState != null) {
        // Restore state here

        surname.setText(savedInstanceState.getString( key: "KEY_SURNAME"));
        name.setText(savedInstanceState.getString( key: "KEY_NAME"));
        birthdate.setText(savedInstanceState.getString( key: "KEY_BIRTHDATE"));
        number.setText(savedInstanceState.getString( key: "KEY_NUMBER"));
        mail.setText(savedInstanceState.getString( key: "KEY_MAIL"));
        hobby.setText(savedInstanceState.getString( key: "KEY_HOBBIES"));
    }
    //Log.v("debug presistancy","Values recoverd...");
}
```

3 Réseau

Cet exerice n'a pas été réalisé.

4 Service



Après qu'une écriture sur le stockage du téléphone a été faite, il nous est possible de récupérer le contenu de ce fichier et de l'afficher à l'utilisateur.

À l'appuie du boutton "load", le service est démaré. *FileDownloadService* va simplement lire le fichier de sauvegarde prédéfini *test.json*.

Il va ensuite, diffuser les valeurs à travers un le *LocalBroadcastManager*.

Dans le DisplayFragment, Nous nous mettons à l'écoute du broadcast avec la fonction **registerReceiver()**.

À la reception du broadcast, le text est écrasé par les différentes valeurs contenus dans le fichier grâce à la fonction **onReceive()**.

On remarquera que l'*Intent* et l'*IntentFilter* ont le même nom.

```
@Override
protected void onHandleIntent(@Nullable Intent intent) {
    String relativePath = "test.json";
    parseFile(relativePath);
}


1 usage   SateaMall
private void parseFile(String relativePath) {
    // Obtain the absolute path to the external files directory
    File basePath = getExternalFilesDir( type: null);
    if (basePath != null) {
        // Correctly append the relative file path
        File file = new File(basePath, relativePath);
        if (file.exists()) {
            try (FileInputStream fis = new FileInputStream(file)) {
                byte[] data = new byte[(int) file.length()];
                fis.read(data);

                String jsonString = new String(data, charsetName: "UTF-8");
                JSONObject jsonObject = new JSONObject(jsonString);

                // Log and broadcast parsed JSON data
                Log.d( tag: "FileDownloadService",  msg: "Parsed JSON data: " + jsonObject.toStr
                Intent intent = new Intent( action: "action.UPDATE_DATA");
                intent.putExtra( name: "surname", jsonObject.getString( name: "surname"));
                intent.putExtra( name: "name", jsonObject.getString( name: "name"));
                intent.putExtra( name: "birthdate", jsonObject.getString( name: "birthdate"));
                intent.putExtra( name: "number", jsonObject.getString( name: "number"));
                intent.putExtra( name: "mail", jsonObject.getString( name: "mail"));
                intent.putExtra( name: "hobbies", jsonObject.getString( name: "hobbies"));
                LocalBroadcastManager.getInstance( context: this).sendBroadcast(intent);
```

```
private final BroadcastReceiver mMessageReceiver = (context, intent) -> {
        Log.d( tag: "Broadcast",  msg: "Received broadcast...");
        String surname = intent.getStringExtra( name: "surname");
        String name = intent.getStringExtra( name: "name");
        String birthdate = intent.getStringExtra( name: "birthdate");
        String number = intent.getStringExtra( name: "number");
        String mail = intent.getStringExtra( name: "mail");
        String hobbies = intent.getStringExtra( name: "hobbies");

        TextView_surname.setText(surname);
        TextView_name.setText(name);
        TextView_birthdate.setText(birthdate);
        TextView_number.setText(number);
        TextView_mail.setText(mail);
        TextView_hobbies.setText(hobbies);
};

 SateaMall +1 *
@Override
public void onStart() {
    Log.v( tag: "debug whenDoesItWork",  msg: "onStart()");
    super.onStart();
    IntentFilter intentFilter = new IntentFilter( action: "action.UPDATE_DATA");
    LocalBroadcastManager
            .getInstance(requireContext())
            .registerReceiver(mMessageReceiver, intentFilter);
}

 SateaMall +1
@Override
public void onStop() {
    Log.v( tag: "debug whenDoesItWork",  msg: "onStop()");
    super.onStop();
    LocalBroadcastManager.getInstance(requireContext()).unregisterReceiver(mMessageReceiver);
}
```

5 Conclusion

Pour conclure cet exercice, nous avons réaliser une petite application formulaire avec des fragments communiquant entre eux.

Malgré un sujet assez obscure, nous avons interpréter aux mieux de nos capacités. Nous avons réussi à implémenter, nous pensons, la majorité des fonctionnalités attendus : l'utilisateur peut afficher et sauvegarder ces données (même s'il n'a encore rien écrit), modifier l'affichage à la volée mais aussi les charger depuis un fichier.

FIGURE 1 – Diagrame d'utilisation de notre application mobile

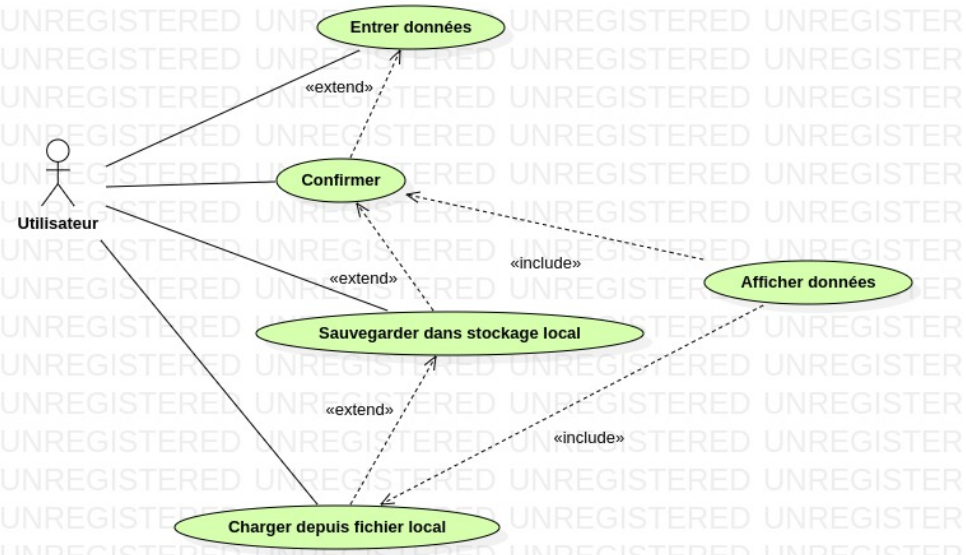


FIGURE 2 – Diagrame de séquence *possible* représentant la mise en place de la communication inter-fragment