

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 vu l'utilisation des Fragments, ModelAndView et MutableLiveData.

Table des matières

Introduction	3
Démonstration	3
Démonstration	3
1 Fragment	3
1.1 Création	3
1.2 Ajout	3
1.3 Communication entre fragment	4
1.4 La synchronisation et mise à jour	4
1.5 Observer	5
2 Persistance	5
2.1 Sauvegarde	5
2.2 (Re-)Saisie automatique	5
3 Réseau	6
4 Service	6

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/NjIdJrN-Rm8>!!TODO!! une démonstration vidéo de notre travail.

Installation

Vous trouvez les instructions dans le README.md

To Do

- diagram use case DONE
- diagram sequence for synchronisation DONE
- diagram state machine w/ tikz library to describe save function NO
- add code picture DONE
- add smartphone picture DONE
- !!! make an .apk file for easy install, no compilation!!! NO NEED, ALREADY DONE
- diagram class observer DONE
- diagram class observer specific (Fragment Manager) KO
- → !!! → vidéo ← !!! ←

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.

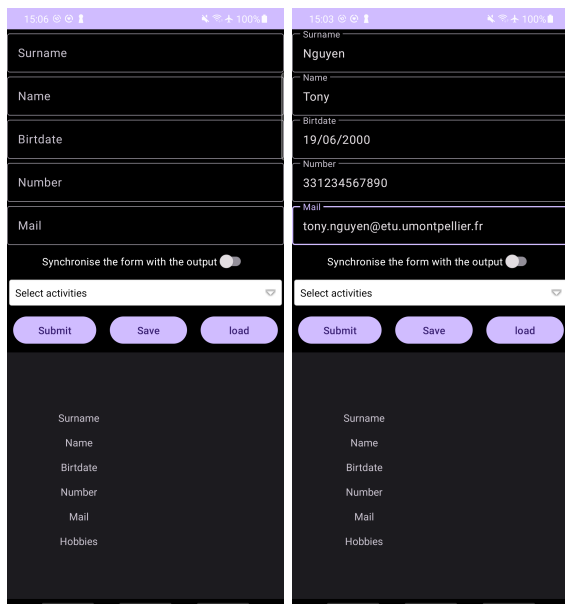
1.1 Création

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 créé 2 fragments, `UserInputFragment` et `DisplayFragment`.

1.2 Ajout



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

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

L'attribut **android:name** associe cette balise .xml à la classe fragment créée 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 chaîne de caractères 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 **setFragmentManagerListener()** de la classe *FragmentManager*.

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

```
getFragmentManager().setFragmentManagerListener( requestKey: "Hello", lifecycleOwner: this,
4 usages 1 tony-nguyen1
@RequiresApi(api = Build.VERSION_CODES.TIRAMISU)
@Override
public void onFragmentManagerResult(@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 changement, nous souhaitons publier 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 sélectionné 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.findViewById(R.id.switch_sync)).isChecked();

    Toast.makeText(getContext(), "Input submitted" + (this.isSynchronousWithOutput ? " (w/ sync on)" : ""), Toast.LENGTH_SHORT).show();
});
```

Afin d'être notifié des publications, il ne faut pas oublier de s'abonner 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));
```

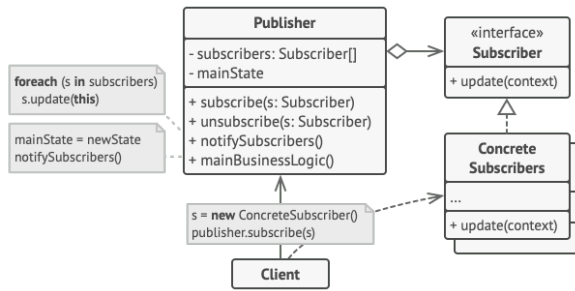
Quand l'observateur est notifié, la méthode **onChanged()** est appelé et la nouvelle valeur est placée dans une *<TextView>*.

```
private Observer<String> createCustomObserver(int id) {
    1 tony-nguyen1
    return new Observer<String>() {
        1 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.getText());
            //Log.v("debug display wanted surname ", myText.getText());
            myText.setText(newName);
        }
    };
}
```

1.5 Observer

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



```

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

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

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()`.

2.2 (Re-)Saisie automatique

Imaginons un petit scénario.

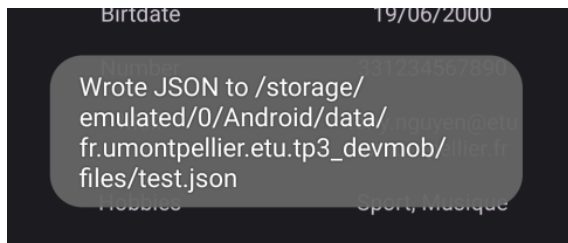
Un utilisateur commence à saisir ses informations et fait une mauvaise manipulation. Il a appuyé sur le bouton de retour à l'écran d'accueil (Android) du smartphone.

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

2 Persistance

Nous allons maintenant nous intéresser à la conservation des données.

2.1 Sauvegarde



```

tony@workstation: /run/user/1000/gvfs/ftp:host=SAMSUNG_SAMSUNG_A...
tony@workstation: /run/user/1000/gvfs/ftp:host=SAMSUNG_SAMSUNG_Android_R58NC2EFBXX/Phone/Android/data/fr.umontpellier.etu.tp3_devmob/files$ pwd
/run/user/1000/gvfs/ftp:host=SAMSUNG_SAMSUNG_Android_R58NC2EFBXX/Phone/Android/data/fr.umontpellier.etu.tp3_devmob/files
tony@workstation: /run/user/1000/gvfs/ftp:host=SAMSUNG_SAMSUNG_Android_R58NC2EFBXX/Phone/Android/data/fr.umontpellier.etu.tp3_devmob/files$ ls
test.json
tony@workstation: /run/user/1000/gvfs/ftp:host=SAMSUNG_SAMSUNG_Android_R58NC2EFBXX/Phone/Android/data/fr.umontpellier.etu.tp3_devmob/files$ cat test.json | json_pp
{
  "birthdate": "19/06/2000",
  "hobbies": "Sport, Musique",
  "mail": "tony.nguyen@etu.umontpellier.fr",
  "name": "Tony",
  "number": "331234567890",
  "surname": "Nguyen"
}
tony@workstation: /run/user/1000/gvfs/ftp:host=SAMSUNG_SAMSUNG_Android_R58NC2EFBXX/Phone/Android/data/fr.umontpellier.etu.tp3_devmob/files$
  
```

Lors d'un clic sur le bouton "Save", une écriture sur le stockage local du smartphone est enclenchée.

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

Échec

3 Réseau

Cet exercice n'a pas été réalisé.

4 Service

Surname	Nguyen
Name	Tony
Birtdate	19/06/2000
Number	331234567890
Mail	tony.nguyen@etu.umontpellier.fr
Hobbies	Sport, Musique

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 bouton "load", le service est démarré. *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);
}

// Usage
private void parseFile(String relativePath) {
    // Obtain the absolute path to the external files directory
    File basePath = getExternalFilesDir(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, Charset.forName("UTF-8"));
                JSONObject jsonObject = new JSONObject(jsonString);

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

private final BroadcastReceiver mMessageReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d("Broadcast", "Received broadcast...");
        String surname = intent.getStringExtra("surname");
        String name = intent.getStringExtra("name");
        String birthdate = intent.getStringExtra("birthdate");
        String number = intent.getStringExtra("number");
        String mail = intent.getStringExtra("mail");
        String hobbies = intent.getStringExtra("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);
    }
};

// SateMail
@Override
public void onStart() {
    Log.v("debug whenDoesItWork", "onStart()");
    super.onStart();
    IntentFilter intentFilter = new IntentFilter("action.UPDATE_DATA");
    LocalBroadcastManager
        .getInstance(requireContext())
        .registerReceiver(mMessageReceiver, intentFilter);
}

// SateMail
@Override
public void onStop() {
    Log.v("debug whenDoesItWork", "onStop()");
    super.onStop();
    LocalBroadcastManager.getInstance(requireContext()).unregisterReceiver(mMessageReceiver);
}
```

FIGURE 1 – Diagramme d'utilisation de notre application mobile

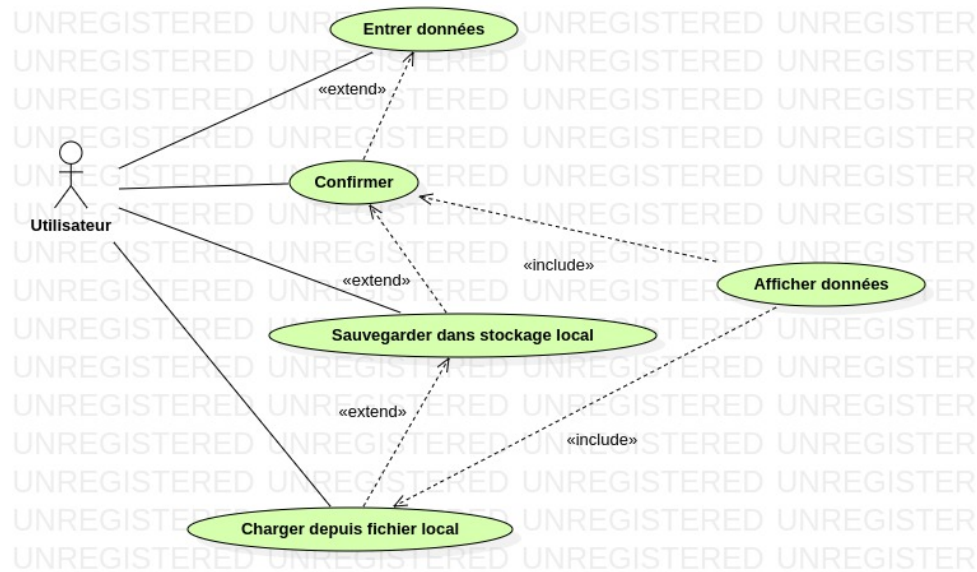


FIGURE 2 – Diagramme de séquence *possible* représentant la mise en place de la communication inter-fragment

