DAA - Laboratoire 03

Développement d'Applications Android

Martins Alexis, Saez Pablo

HEIG VD

12 novembre 2023

Table des matières

1 Introduction

Dans ce troisième laboratoire de DAA, le but est de nous faire pratiquer la manipulation des divers composants utilisables dans un formulaire. Nous devons refaire une interface d'un formulaire dont on nous a donné le modèle. Ce formulaire permet à une personne de saisir ses informations personnelles, ainsi que son occupation. Cette dernière saisie implique d'avoir un formulaire dynamique selon si la personne est un étudiant ou un travailleur.

2 Réponses aux questions

2.1 Pour le champ remark, destiné à accueillir un texte pouvant être plus long qu'une seule ligne, quelle configuration particulière faut-il faire dans le fichier XML pour que son comportement soit correct ? Nous pensons notamment à la possibilité de faire des retours à la ligne, d'activer le correcteur orthographique et de permettre au champ de prendre la taille nécessaire.

Voici le code utilisé pour ce champ dans notre layout.

```
<EditText
   android:id="@+id/input_edittext_remarks"
   android:layout_width="0dp"
   android:layout_height="wrap_content"
   android:inputType="textMultiLine|textAutoCorrect"
   android:layout_marginTop="@dimen/remarks_margin"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toBottomOf="@+id/additional_remarks_title"
   android:importantForAutofill="no"
   tools:ignore="LabelFor" />
```

Les modifications importantes afin que le comportement décrit soit respecté sont dans le paramètre `inputType`. On y retrouve notamment deux valeurs :

- textMultiLine : Il permet de répondre au besoin d'avoir un champ de saisie textuelle sur plusieurs lignes.
- textAutoCorrect : Il permet de répondre au besoin d'accès au correcteur orthographique.

Finalement pour que le champ prenne la taile du texte entré par l'utilisateur, on spécifie qu'il doit prendre toute la taile dans la largeur. De plus, il va s'étendre verticalement, car on a précisé qu'il devait s'adapter au contenu saisi avec `wrap_content`.

2.2 Pour afficher la date sélectionnée via le DatePicker nous pouvons utiliser un DateFormat permettant par exemple d'afficher 12 juin 1996 à partir d'une instance de Date. Le formatage des dates peut être relativement différent en fonction des langues, la traduction des mois par exemple, mais également des habitudes régionales différentes : la même date en anglais britannique serait 12th June 1996 et en anglais américain June 12, 1996. Comment peut-on gérer cela au mieux ?

Voici le code utilisé pour la gestion du `DatePickerDialog`.

```
    private fun showDatePickerDialog() {
        val currentDate = Calendar.getInstance()
        val year = currentDate.get(Calendar.YEAR)
        val month = currentDate.get(Calendar.MONTH)
        val day = currentDate.get(Calendar.DAY_OF_MONTH)

        val userLocale = Locale.getDefault()

        val datePickerDialog = DatePickerDialog(
            this,
            { _, selectedYear, selectedMonth, selectedDay ->
                // Update the birthdayEditText with the selected date
                val formattedDate = SimpleDateFormat("yyyy-MM-dd",
                    userLocale)
                    .format(Calendar.getInstance().apply {
                        set(Calendar.YEAR, selectedYear)
                        set(Calendar.MONTH, selectedMonth)
                        set(Calendar.DAY_OF_MONTH, selectedDay)
                    }.time)

                birthdayEditText.setText(formattedDate)
            },
            year,
            month,
            day
        )

        datePickerDialog.show()
    }
```

Dans ce code, c'est la méthode `Locale.getDefault()` qui s'occupe de récupérer les paramètres de l'utilisateur afin d'afficher la date dans le bon format. On le comprend assez bien si on se réfère à l'explication dela méthode.

> Gets the current value of the default locale for this instance of the Java Virtual Machine. The Java Virtual Machine sets the default locale during startup based on the host environment.

2.3 Si vous avez utilisé le DatePickerDialog du SDK. En cas de rotation de l'écran du smartphone lorsque le dialogue est ouvert, une exception android.view.WindowLeaked sera présente dans les logs, à quoi est-elle due ?

Comme vu durant le laboratoire précédent (et le cours), le changement d'orientation d'une activité va causer sa recréation. De ce fait, le DPD ne se ferme pas correctement et provoque une erreur.

La solution serait d'intercepter les évènements appelés par l'activité à ce moment-là afin de fermer le DPD avant qu'il n'ait le temps de générer une erreur.

```
1  private fun dismissDatePickerDialog() {
2      datePickerDialog?.let {
3          if (it.isShowing) {
4              it.dismiss()
5          }
6      }
7  }
8
9  override fun onSaveInstanceState(outState: Bundle) {
10     super.onSaveInstanceState(outState)
11     dismissDatePickerDialog()
12 }
13
14 override fun onDestroy() {
15     dismissDatePickerDialog()
16     super.onDestroy()
17 }
```

2.4 Lors du remplissage des champs textuels, vous pouvez constater que le bouton « suivant » présent sur le clavier virtuel permet de sauter automatiquement au prochain champ à saisir, cf. Fig. 2. Est-ce possible de spécifier son propre ordre de remplissage du questionnaire ? Arrivé sur le dernier champ, est-il possible de faire en sorte que ce bouton soit lié au bouton de validation du questionnaire ?

Premièrement, il est important de préciser que nativement c'est assez bien fait. Lorsque l'on souhaite passer au prochain champ du formulaire, Kotlin vérifie quel est le champ le plus proche auquel il peut passer. Ce qui explique pourquoi est-ce que notre code actuel fonctionne dans le bon ordre et pas dans un ordre aléatoire ou autre.

Ensuite, il est en effet possible de changer l'ordre dans lequel défilent les champs lorsque l'on fait suivant. On a trouvé plusieurs solutons, tel que `NextFocusRight`et ses variations par exemple, mais la plus polyvalentes reste `android:imeOptions="actionNext"`. Nous avons trouvé cette réponse sur un poste StackOverflow où on peut même y retrouver un exemple d'utilisation. Il suffit de mettre `ActionNext` dans les inputs qui doivent être modifiés et ensuite dans le code directement, il est possible d'établir l'ordre dans lequel se font les focus.

Finalement, par rapport au submit automatique lorsque l'on arrive en fin de champ, on peut utiliser `android:imeOptions="actionDone"` et le combiner avec le code ci-dessous.

```
emailEditText.setOnEditorActionListener { _, actionId, _ ->
    if (actionId == EditorInfo.IME_ACTION_DONE) {
          // If the condition is true, we can call the function to create a
             person
          createPerson()
          return@setOnEditorActionListener true
    }
    false
}
```

2.5 Pour les deux Spinners (nationalité et secteur d'activité), comment peut-on faire en sorte que le premier choix corresponde au choix null, affichant par exemple « Sélectionner » ? Comment peut-on gérer cette valeur pour ne pas qu'elle soit confondue avec une réponse ?

Nous sommes partis du principe que l'on allait aussi ajouter l'option `Sélectionner` dans les spinners. Cette option, c'est la première option de chaque Spinner (index 0).

Dans le code lorsque l'on doit travailler avec un Spinner, on va simplemenet vérifier que ça ne soit pas l'index 0 qui est actuellement l'élément sélectionné. Voici un exemple qui est utilisé pour vérifier qu'un `Worker` ait bien choisi son secteur. On voit à la deuxième ligne la vérification de la position apparaître.

```
if (companyEditText.text.isEmpty() ||
    sectorSpinner.selectedItemPosition == 0 ||
    experienceEditText.text.isEmpty()) {

            Toast.makeText(this, "Merci de remplir tous les champs", Toast.
               LENGTH_SHORT).show()
            return null
}
```

3 Explications de l'implémentation

Par rapport au layout, on a essayé comme conseiller d'utiliser le moins de layouts possible. On a donc le `ConstraintLayout` obligatoire et ensuite tous les champs sont éléments sont directement dans celui-ci. Pour les zones spécifiques aux occupations, ces parties sont des des groupes que l'on peut afficher ou cacher selon nos besoins.

Pour la partie contenant le code lui-même, il n'y a pas grand chose à dire. On a essayé de partionner le code en fonctions le plus possible comme on le ferait dans les bonnes pratiques.

Pour les éléments spécifiques, les Spinners ont déjà eu leur explication. Le DPD a été utilisé car c'est l'élément sur lequel nous sommes tombés en premier et qui répondait bien à nos besoins sans trop de complexité.

4 Conclusion

Ce laboratoire nous a permis de nous familiariser avec les différents éléments de formulaire que l'on peut utiliser dans une application Android. On a pu voir que la plupart des éléments sont assez simples à utiliser. Il faut juste bien comprendre comment ils fonctionnent et comment les utiliser ce qui peut être difficile étant donné le large éventail de possibilités pour chaque élément. Le mieux reste de s'en tenir à la documentation officielle ou aux divers posts sur Internet.