Développement Android

Laboratoire n°2: Interactions avec l'utilisateur - Approche MVC

Friedli Jonathan, Marengo Stéphane, Silvestri Géraud

15.11.2022

Introduction

Le but de ce laboratoire est de réaliser une activité proposant un formulaire permettant d'éditer les informations d'une personne. Ledit formulaire est décomposé en plusieurs parties et devra s'adapter à la saisie d'un étudiant ou d'un travailleur en affichant uniquement les parties pertinentes.

1. Détails d'implémentation

1.1. Layout

Nous avons réussi à faire toute notre interface graphique à l'aide d'un seul `constraint layout`. Nous avons regroupé les différents champs dans plusieurs groupes: `group_base` contient toutes les informations principales comme le nom, la date de naissance et les `radio button` permettant de choisir entre un étudiant et un employé.

Le groupe `group_student` contient toutes les informations relatives à un étudiant comme l'université et l'année de la filière. Ce groupe n'est visible que si le `radio button` "étudiant" est coché.

Le groupe `group_worker` contient toutes les informations relatives à un employé comme son entreprise, son secteur et son expérience. Comme précédemment, ce groupe n'est visible que si le `radio button` "employé" est coché.

Finalement, nous avons un dernier groupe qui est tout le temps affiché: `group_additional` qui contient les informations supplémentaires comme l'adresse mail et un champ pour faire des commentaires.

Il est important de noter que les groupes `group_base` et `group_additional` ne sont pas utiles, mais nous les avons mis afin d'être cohérent.

Voici le code qui permet de les masquer/afficher:

```
radGroup.setOnCheckedChangeListener { _, checkedId ->
    when (checkedId) {
        R.id.main_occupation_student -> {
            workerGroup.visibility = View.GONE
            studentGroup.visibility = View.VISIBLE
        }
        R.id.main_occupation_worker -> {
            workerGroup.visibility = View.VISIBLE
            studentGroup.visibility = View.GONE
        }
        else -> {
            workerGroup.visibility = View.GONE
            studentGroup.visibility = View.GONE
        }
    }
}
```

1.2. ScrollView

Nous avons mis tous nos layout dans une scrollview afin de voir toutes les informations lorsque le clavier est ouvert.

```
<ScrollView 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"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">


     <!-- layout -->

</ScrollView>
```

1.3. Barrière

Nous avons utilisé une barrière afin de faire en sorte que toutes les `EditText` soit alignées. Pour ce faire, nous avons rajouté une balise `androidx.constraintlayout.widget.Barrier`

```
<androidx.constraintlayout.widget.Barrier
            android:id="@+id/main_barrier"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:barrierDirection="end"
            app:barrierMargin="@dimen/barrier_margin"

app:constraint_referenced_ids="main_lastName_title,main_firstName_title,main_birth
Date_title,
        main_nationality_title, main_occupation_title,student_school_name_title,

student_diploma_year_title,worker_company_title,worker_experience_title,worker_sec
tor_title" />
```

A l'intérieur de cette dernière, nous avons rajouté tous les id des `TextView` qui nous intéressent. Ensuite, nous avons utilisé cette barrière comme référence pour toutes les `EditText` afin qu'elles soient alignées.

1.4. MaterialDatePicker

Nous avons eu le choix entre utiliser `DatePickerDialog` et `MaterialDatePicker` afin de choisir une date. Nous avons choisi d'utiliser `MaterialDatePicker`, car il est plus moderne. Pour le détails d'implémentations, voir la question 2.3.

1.5. Ouverture du DatePicker

Nous avons gérer l'ouverture du `DatePicker` grace à cette méthode:

```
txtBirthday.setOnFocusChangeListener { _, hasFocus ->
    if (hasFocus) {
        showDatePicker()
    }
}
txtBirthday.keyListener = null // Désactive le clavier
```

1.6. Bouton ok/cancel

Les listeners sont définis comme suit:

```
btnSubmit.setOnClickListener {
    createPerson()
}

btnReset.setOnClickListener {
    resetFields()
}
```

La méthode `resetFields()` permet de clear tous les `EditText` et `RadioGroup` et de remettre les séléctions des `Spinner` à zéro.

1.7. Création d'une personne

Lors de la soumission du formulaire, nous vérifions en premier lieu que tous les champs en commun sont remplis, à l'exception du champ commentaire. Si ce n'est pas le cas, un message d'erreur s'affiche à l'aide d'un `Toast`. Nous vérifions ensuite que le champ email contienne bien un email valide grâce à une Regex fournie par Android.

```
if (!Patterns.EMAIL_ADDRESS.matcher(email).matches()) {
    showError(resources.getString(R.string.error_invalid_email))
    return
}
```

Ensuite la personne est créée en fonction de son type (étudiant ou employé):

```
when (radGroup.checkedRadioButtonId) {
    R.id.main_occupation_student -> {
        // Création d'un étudiant
    }
    R.id.main_occupation_worker -> {
        // Création d'un employé
    }
    else -> {
        showError(resources.getString(R.string.error_undefined_occupation))
        return
    }
}
Log.println(Log.INFO, "Person", person.toString())
Toast.makeText(this, getString(R.string.user_created), Toast.LENGTH_LONG).show()
resetFields()
```

Les champs spécifiques à un étudiant ou un employé sont vérifiés dans le when. Si un champ n'est pas rempli, un message d'erreur s'affiche à l'aide d'un `Toast`.

Voici l'implémentations de la fonction `showError`

```
private fun showError(message: String) {
    Toast.makeText(this, message, Toast.LENGTH_SHORT).show()
}
```

1.8. Charger une personne existante

La méthode suivante, qui est publique, permet de charger une personne existante et d'afficher ses informations dans le formulaire:

```
fun loadPerson(person: Person) {
    txtLastName.setText(person.name)
    txtFirstName.setText(person.firstName)

    val formattedDate = dateFormatter.format(
        LocalDateTime.ofInstant(
            person.birthDay.toInstant(),
            ZoneId.systemDefault()
        )
    )
    txtBirthday.setText(formattedDate)

    val nationalityPosition = nationalityAdapter.getPosition(person.nationality)
    spnNationality.setSelection(nationalityPosition)

    txtEmail.setText(person.email)
    txtRemark.setText(person.remark)

    when (person) {
        is Student -> {
            radStudent.isChecked = true
            txtUniversity.setText(person.university)
            txtGraduationYear.setText(person.graduationYear.toString())
        }
        is Worker -> {
            radWorker.isChecked = true
            txtCompany.setText(person.company)

            val sectorPosition = sectorAdapter.getPosition(person.sector)
            spnSector.setSelection(sectorPosition)

            txtExperienceYear.setText(person.experienceYear.toString())
        }
    }
}
```

2. Question complémentaire

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 la balise `EditText` dans laquelle on peut voir l'attribut `android:inputType` qui permet d'activer le correcteur orthographique et le mode multilignes.

```
<EditText
    android:id="@+id/additional_remarks_editText"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:autofillHints="remarks"
    android:inputType="text|textMultiLine|textCapSentences|textAutoCorrect"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/additional_remarks_title" />
```

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 ?

Pour gérer cela au mieux, il est recommandé d'utiliser les classes `DateTimeFormatter` et `LocalDate`. Un formateur localisé peut être obtenu via la méthode `DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM)` qui permet de prendre en compte le format actuel du système.

Cette ligne permet de parser une date depuis une chaîne de caractères:

```
// dateFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM)
val date = LocalDate.parse(txtBirthday.text, dateFormatter)
```

2.3

Est-il possible de limiter les dates sélectionnables dans le dialogue, en particulier pour une date de naissance il est peu probable d'avoir une personne née il y a plus de 110 ans ou à une date dans le futur. Comment pouvons-nous mettre cela en place ?

Oui, c'est possible. Voici l'implémentation de notre date picker:

```
private fun showDatePicker() {
    val selectedDate = Calendar.getInstance()
    if (txtBirthday.text.isNotEmpty()) {
        val date = LocalDate.parse(txtBirthday.text, dateFormatter)
        selectedDate.set(date.year, date.monthValue - 1, date.dayOfMonth)
    }

    // Calcule de l'année minimale en utilisant l'année courante
    val minDateInMs =

OffsetDateTime.now(ZoneOffset.UTC).minusYears(DATEPICKER_RELATIVE_LOWER_YEAR)
            .toInstant().toEpochMilli()

    // Définition des validateurs de date pour désactiver les dates invalides
    val calendarValidators = CompositeDateValidator.allOf(
        listOf(
            DateValidatorPointBackward.now(), // pas de date dans le futur
            DateValidatorPointForward.from(minDateInMs)
        )
    )

    val calendarConstraints = CalendarConstraints.Builder()
        .setOpenAt(selectedDate.timeInMillis) // ouverture du calendrier à la date
sélectionnée
        .setEnd(MaterialDatePicker.thisMonthInUtcMilliseconds()) // Empêche
d'aller sur les mois suivants
        .setValidator(calendarValidators)

    val datePicker = MaterialDatePicker.Builder.datePicker()
        .setCalendarConstraints(calendarConstraints.build())
        .setSelection(selectedDate.timeInMillis)
        .build()

    datePicker.show(supportFragmentManager, null)

    datePicker.addOnPositiveButtonClickListener {
        // Conversion de la date sélectionnée en string
        val date = LocalDateTime.ofInstant(Instant.ofEpochMilli(it),
ZoneOffset.UTC)
        txtBirthday.setText(date.format(dateFormatter))
    }
}
```

Comme indiqué dans la documentation du `DatePicker`, ce dernier ne travaille qu'en UTC. Ce qui nous force à spécifier la timezone à certaines lignes.

Nous faisons deux `validators` qui permettent de limiter les dates sélectionnables dans le dialogue. Dans notre cas, les dates sont limitées dans l'intervalle [110 ans en arrière, aujourd'hui].

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 ?

Oui c'est possible. Pour spécifier son propre ordre de remplissage du questionnaire, il faut ajouter l'attribut `android:nextFocusForward` et `android:imeOptions="actionNext"` sur les champs en questions.

Pour faire en sorte que le bouton « suivant » soit lié au bouton de validation du questionnaire, il faut ajouter l'attribut `android:imeOptions="actionDone"` sur le champ en question.

Ensuite, il faut ajouter un listener dans l'activité et simuler un clic du bouton de validation du questionnaire.

```
additional_remarks_editText.setOnEditorActionListener { _, actionId, _ ->
    if (actionId == EditorInfo.IME_ACTION_DONE) {
        ok_button.performClick()
        true
    } else {
        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 avons fait un adapteur personnalisé qui prend une chaîne de caractères en paramètre. Cette chaîne de caractères est utilisée comme affichage lorsque aucun élément n'est sélectionné.

Pour gérer l'affichage, nous avons réécrit les méthodes suivantes:

```
override fun getDropDownView(position: Int, convertView: View?, parent:
ViewGroup): View {
    if (position == 0) {
        return View(context).apply {
            visibility = View.GONE
            tag = 1 // indique que cette vue est un "hint" et qu'il ne faut pas la
réutiliser
        }
    }

    return if (convertView?.tag == 1) { // si la vue précédente est un "hint", il
faut en créer une nouvelle
        super.getDropDownView(position, null, parent)
    } else {
        super.getDropDownView(position, convertView, parent)
    }
}

override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
    if (position == 0) {
        return getHintView(parent)
    }
    return super.getView(position, convertView, parent)
}

private fun getHintView(parent: ViewGroup): View {
    val view = inflater.inflate(resource, parent, false)

    view.findViewById<TextView>(android.R.id.text1).also {
        it.hint = this.hintText
    }

    return view
}
```

Et voici comment nous l'utilisons dans le code:

```
sectorAdapter = ArrayAdapterWithDefaultValue(
    this,
    android.R.layout.simple_list_item_1,
    resources.getStringArray(R.array.sectors).toList(),
    resources.getString(R.string.sectors_empty)
)

spnSector.adapter = sectorAdapter

spnSector.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
    override fun onItemSelected(
        parent: AdapterView<*>?,
        view: View?,
        position: Int,
        id: Long
    ) {
        selectedSector = sectorAdapter.getItem(position) // On garde une référence
sur l'élément sélectionné lors d'un changement
    }

    override fun onNothingSelected(parent: AdapterView<*>?) {
        selectedSector = null
    }
}
```

Pour ne pas confondre la première valeur avec une réponse, nous avons inséré `null` au début de la liste des valeurs.

Ces deux lignes permettent de sélectionner le bon élément dans le spinner en fonction de la valeur:

```
val nationalityPosition = nationalityAdapter.getPosition(person.nationality)
spnNationality.setSelection(nationalityPosition)
```