# Développement Android

## Laboratoire n°5: Application communicante

Friedli Jonathan, Marengo Stéphane, Silvestri Géraud

29.01.2023

## Introduction

Le but de ce laboratoire est de développer une application android servant de gestionnaire de contacts. L'application doit permettre de créer, modifier et supprimer des contacts. Les contacts sont stockés dans une base de données locale. Il faut également implémenter un algorithme de synchronisation des contacts avec un serveur distant.

## Détails d'implémentation

### UI

**AppContact:**

```
@Composable
fun AppContact(/* ... */) {
    val contacts: List<Contact> by
contactsViewModel.allContacts.observeAsState(initial = emptyList())
    var editionMode by remember { mutableStateOf(false) }
    var selectedContact: Contact? by remember { mutableStateOf(null) }

    fun closeEdition() {
        selectedContact = null
        editionMode = false
    }

    Scaffold(
        topBar = {
            if (editionMode) { // Affichage de la bonne barre de navigation
                BackTopBar {
                    closeEdition() // Callback du bouton retour
                }
            } else {
                HomeTopBar(contactsViewModel)
            }
        },
        floatingActionButtonPosition = FabPosition.End,
        floatingActionButton = {
            // Passage en mode édition lors du click sur le bouton
        },
    )
    { padding ->
```

```
        Column(modifier = Modifier.padding(padding)) { }

        Box(Modifier.padding(8.dp)) {
            if (editionMode) { // Affichage du bon composant en fonction de l'état
d'édition
                ScreenContactEditor(selectedContact) { type, contact ->
                    // Appel de la bonne méthode du ViewModel en fonction de type
                    closeEdition()
                }
            } else {
                ScreenContactList(contacts) { contact ->
                    selectedContact = contact
                    editionMode = true
                }
            }
        }
    }
}

@Composable
fun BackTopBar(onBackPressed: () -> Unit) {
    TopAppBar(
        title = { Text(stringResource(R.string.app_name)) },
        navigationIcon = {
            IconButton(onClick = { onBackPressed() }) {
                Icon(Icons.Default.ArrowBack, "Back")
            }
        }
    )
}

@Composable
fun HomeTopBar(contactsViewModel: ContactsViewModel) {
    TopAppBar(
        title = { Text(stringResource(R.string.app_name)) },
        actions = { // Gestion des deux boutons pour la synchronisation
            IconButton(onClick = { contactsViewModel.enroll() }) {
                Icon(painterResource(R.drawable.populate), "Populate")
            }
            IconButton(onClick = { contactsViewModel.refresh() }) {
                Icon(painterResource(R.drawable.synchronize), "Synchronize")
            }
        }
    )
}
```

Pour gérer le mode d'édition ainsi que le contact sélectionné, nous avons ajouté les deux états suivant à la fonction.

```
var editionMode by remember { mutableStateOf(false) }
var selectedContact: Contact? by remember { mutableStateOf(null) }
```

**ScreenContactEditor:**

```
enum class ActionType {
    CANCEL, DELETE, SAVE
}

private val dateFormatter = DateTimeFormatter.ofPattern("dd.MM.yyyy")

@Composable
fun ScreenContactEditor(
    contact: Contact? = null,
    onClose: (ActionType, Contact?) -> Unit
) {
    val context = LocalContext.current
    val newContact = remember { contact?.copy() ?: Contact.empty() }
    val formattedBirthday = remember {
        mutableStateOf(
            newContact.birthday?.let {
                ZonedDateTime.ofInstant(it.toInstant(), it.timeZone.toZoneId())
                    .format(dateFormatter)
            } ?: ""
        )
    }
    val phoneTypes = PhoneType.values()
    val selectedPhoneType = remember {
        mutableStateOf(contact?.type ?: phoneTypes.first())
    }

    Column(Modifier.verticalScroll(rememberScrollState())) {
        Text(/* ... */) // Affichage du titre
        TextFieldWithLabel("Name", newContact.name) {
            newContact.name = it
        }
        // Autres champs TextFieldWithLabel

        Row {Text("Phone type", color = Color.Gray)}
        Row(modifier = Modifier.fillMaxWidth()) {
            phoneTypes.forEach { type ->
               // Affichage des boutons radios avec un onClick pour définir la
valeur selectedPhoneType et de newContact.type
            }
        }

        Row(/* ... */) {
            Button(onClick = { onClose(ActionType.CANCEL, null) }) {/* text */ }

            if (contact != null) { // Affichage du bouton de suppression
uniquement si le contact existe
                Button(onClick = { onClose(ActionType.DELETE, contact) }) { /*
text */ }
            }
```

```
            Button(onClick = {
                // "Validation"
                onClose(ActionType.SAVE, newContact)
            }) { /* text */ }
        }
    }
}

@Composable
fun TextFieldWithLabel(label: String, value: String? = null, onChanged: (String) -
> Unit = {}) {
    val focusManager = LocalFocusManager.current
    val text = remember { mutableStateOf(value ?: "") }

    Row(/* ... */) {
        Text(text = label, /* ... */)
        TextField(value = text.value, onValueChange = {
                text.value = it
                onChanged(it)
            },
            modifier = Modifier
                .weight(4f)
                .onPreviewKeyEvent {
                    if (it.nativeKeyEvent.keyCode == KeyEvent.KEYCODE_TAB &&
it.nativeKeyEvent.action == KeyEvent.ACTION_DOWN) {
                        focusManager.moveFocus(FocusDirection.Down)
                        true
                    } else {
                        false
                    }
                }, /* ... */),
            keyboardOptions = KeyboardOptions(imeAction = ImeAction.Next),
            keyboardActions = KeyboardActions(onNext = {
focusManager.moveFocus(FocusDirection.Down) }),
        )
    }
}
```

Le code ci-dessus permet d'afficher le formulaire d'édition. Un callback est passé en paramètre pour gérer la fermeture de l'écran. Si un contact est passé à la fonction, les modifications seront effectuées sur une copie de ce contact. Sinon, un nouveau contact vide sera créé.

Nous avons créé une fonction composable `TextFieldWithLabel` que nous utilisons pour chacun des champs textuels. Nous avons également ajouté un callback `onChanged` pour pouvoir mettre à jour le contact en cours d'édition. Nous avons géré le focus sur le champ suivant à l'aide de `onPreviewKeyEvent`. La touche `TAB` est également gérée pour simplifier nos tests.

Lorsque nous cliquons sur un bouton, nous appelons le callback `onClose` avec le type d'action (enum `ActionType`) et le contact en cours d'édition.

Le bout de code ci-dessous permet d'effectuer une validation simple sur la date. Si cette dernière n'est pas valide, la date du jour est utilisée. Nous avons fait cela pour ne pas perdre de temps sur cette partie et se concentrer sur la partie synchronisation.

Nous validons également que le champ nom soit bien présent car c'est le seul champ non nullable de la classe `Contact`.

```
if (formattedBirthday.value.isNotEmpty()) {
    val date = runCatching {
        LocalDate.parse(
            formattedBirthday.value,
            dateFormatter
        )
    }.getOrElse {
        LocalDate.now()
    }
    newContact.birthday = Calendar.getInstance().apply {
        set(date.year, date.monthValue - 1, date.dayOfMonth)
    }
}

if (newContact.name.isEmpty()) {
    Toast.makeText(context, "Name is required", Toast.LENGTH_SHORT).show()
    return@Button
}
```

## Mise en place de la synchronisation

Pour pouvoir effectuer des appels réseau, nous avons dû ajouter la permission `INTERNET` dans le fichier `AndroidManifest.xml`.

```
<uses-permission android:name="android.permission.INTERNET" />
```

**Contact:**

```
@Serializable
@Entity
data class Contact(
    @PrimaryKey(autoGenerate = true) var id: Long? = null,
    @Transient
    var remoteId: Long? = null,
    @Transient
    var status: Status = Status.NEW,
    @Serializable(with = CalendarSerializer::class)
    var birthday: Calendar?,
    // Autres champs
) {
```

```
    companion object {
        fun empty() = Contact(
            // Paramètres initilisés à null
        )
    }

    object CalendarSerializer : KSerializer<Calendar> {
        private val formatter = DateTimeFormatter.ofPattern("yyyy-MM-
dd'T'HH:mm:ss.SSSXXX")

        override val descriptor: SerialDescriptor =
            PrimitiveSerialDescriptor("Calendar", PrimitiveKind.STRING)

        override fun serialize(encoder: Encoder, value: Calendar) {
            encoder.encodeString(
                ZonedDateTime.ofInstant(value.toInstant(),
value.timeZone.toZoneId())
                    .format(formatter)
            )
        }

        override fun deserialize(decoder: Decoder): Calendar {
            val zoneDateTime = ZonedDateTime.parse(decoder.decodeString(),
formatter)
            return GregorianCalendar.from(zoneDateTime)
        }
    }
}

enum class Status {
    OK, NEW, MODIFIED, DELETED
}
```

Le model a été modifié afin de rajouter les champs `status` et `remoteId` afin de gérer la synchronisation. La sérialisation a été faite avec le plugin de sérialisation de Kotlin. Nous avons créé une classe `CalendarSerializer` afin de sérialiser/désérialiser le champ `birthday`. L'annotation `@Transient` permet de ne pas sérialiser les champs `status` et `remoteId`.

**ContactsDao:**

```
@Dao
abstract class ContactsDao {
    @Query("SELECT * FROM Contact WHERE status != :status")
    protected abstract fun getContactsWithout(status: Status):
LiveData<List<Contact>>

    @Query("SELECT * FROM Contact WHERE status != :status")
    protected abstract suspend fun getContactsWithoutAsync(status: Status):
List<Contact>

    fun getContacts(): LiveData<List<Contact>> =
```

```
getContactsWithout(Status.DELETED)

    suspend fun getUnsynchronizedContacts(): List<Contact> =
getContactsWithoutAsync(Status.OK)

    @Insert
    abstract fun insertAll(vararg contacts: Contact)

    // insert, update, delete, deleteAll, ...
}
```

Nous avons créé deux méthodes `protected getContactsWithout` et `getContactsWithoutAsync` pour avoir un typage fort dans la requête avec notre `enum Status`.

Deux autres méthodes `getContacts` et `getUnsynchronizedContacts` permettant de récupérer les contacts. La première récupère tous les contacts sauf ceux qui ont été supprimés localement sous forme de `LiveData`.

La seconde récupère tous les contacts qui n'ont pas encore été synchronisés. Cette dernière ne retourne pas de `LiveData` car nous aurons besoin du résultat immédiatement.

**ContactsViewModel:**

```
class ContactsViewModel(application: ContactsApplication) :
    AndroidViewModel(application) {
    private val prefs = EncryptedSharedPreferences.create(/* ... */)
    private val repository = application.repository
    val allContacts = repository.allContacts

    init {
        viewModelScope.launch {
            repository.initSynchronizer(prefs)
        }
    }

    fun enroll() {
        viewModelScope.launch {
            repository.enroll(prefs)
        }
    }

    fun refresh() {
        viewModelScope.launch {
            repository.synchronize()
        }
    }

    fun delete(contact: Contact) {
        viewModelScope.launch {
            repository.delete(contact)
        }
    }
```

```
    fun save(contact: Contact) {
        viewModelScope.launch {
            if (contact.id != null && repository.exists(contact.id!!))
                repository.update(contact)
            else
                repository.add(contact)
        }
    }
}
```

L'implémention du `ViewModel` est très basique et appelle simplement les méthodes correspondantes du `Repository`. Des `EncryptedSharedPreferences` sont créés et seront passé au `Repository` pour gérer l'UUID.

Le `init` permet d'initialiser la synchronisation sur le `Repository`.

**ContactsRepository:**

```
class ContactsRepository(private val contactsDao: ContactsDao) {
    private var synchronizer: ContactsSynchronizer? = null
    val allContacts = contactsDao.getContacts()

    suspend fun initSynchronizer(prefs: SharedPreferences) =
withContext(Dispatchers.IO) {
        ContactsSynchronizer.getOrNewUUID(prefs).onSuccess { uuid ->
            synchronizer = ContactsSynchronizer(uuid)
        }
    }

    suspend fun add(contact: Contact) =
        withContext(Dispatchers.IO) {
            if (synchronizer?.insert(contact) == false) {
                contact.status = Status.NEW
            }
            contactsDao.insert(contact)
        }

    suspend fun update(contact: Contact) =
        withContext(Dispatchers.IO) {
            val synced: Boolean

            if (contact.status == Status.NEW) {
                synced = synchronizer?.insert(contact) == true
            } else {
                synced = synchronizer?.update(contact) == true

                if (!synced) {
                    contact.status = Status.MODIFIED
                }
```

```
            }

            contactsDao.update(contact)
            synced
        }

    suspend fun delete(contact: Contact) =
        withContext(Dispatchers.IO) {
            if (contact.status == Status.NEW ||
synchronizer?.delete(contact.remoteId!!) == true) {
                contactsDao.delete(contact)
                true
            } else {
                contact.status = Status.DELETED
                contactsDao.update(contact)
                false
            }
        }

    suspend fun enroll(prefs: SharedPreferences) = withContext(Dispatchers.IO) {
        contactsDao.deleteAll()
        ContactsSynchronizer.newUUID(prefs).onSuccess { uuid ->
            synchronizer = ContactsSynchronizer(uuid)
            val contacts = synchronizer?.getContacts() ?: return@withContext
            contactsDao.insertAll(*contacts.toTypedArray())
        }
    }

    suspend fun exists(id: Long) =
        withContext(Dispatchers.IO) {
            contactsDao.getContactById(id) != null
        }

    suspend fun synchronize() = withContext(Dispatchers.IO) {
        for (contact in contactsDao.getUnsynchronizedContacts()) {
            val isOk = when (contact.status) {
                Status.NEW -> update(contact)
                Status.MODIFIED -> update(contact)
                Status.DELETED -> delete(contact)
                Status.OK -> continue
            }

            if (!isOk) {
                return@withContext false
            }
        }

        true
    }
}
```

La méthode `initSynchronizer` permet d'initialiser la synchronisation. Elle fait appel au `ContactsSynchronizer` pour récupérer un UUID, soit des préférences si il existe, soit en contactant le serveur pour en obtenir un nouveau. Si l'opération réussi, un `ContactsSynchronizer` est créé.

La méthode `enroll` permet de supprimer toutes les données locales et tente d'obtenir un nouvel UUID. Si l'opération réussi, un nouveau `ContactsSynchronizer` est créé, les contacts sont récupérés depuis le serveur et sont insérés dans la db locale.

Les différentes méthodes vont d'abord essayer de faire appel au `ContactsSynchronizer` pour synchroniser les données avec le serveur. Si l'opération échoue, la modification est marquée comme non synchronisée.

La méthode `update` possède un cas particulier. Si un contact est édité et qu'il n'a jamais été synchonisé (`status NEW`), il s'agit d'une insertion et non d'une mise à jours du point de vue du `ContactsSynchronizer`.

Chacune des méthodes `add`, `update` et `delete` retourne un booléen indiquant si la synchronisation a été effecutée. Cela nous permet, dans la méthode `synchronize` d'arrêter son exécution si une synchronisation échoue.

Le `Repository` peut fonctionner avec ou sans `Synchronizer` permettant ainsi de facilement gérer les cas où le serveur n'est pas disponible pour nous fournir un UUID. Si le `Synchronizer` n'est pas initialisé dans la méthode `initSynchronizer`, ce dernier sera créé lors d'un `enrollment`.

**ContactsSynchronizer:**

```
class ContactsSynchronizer(private val uuid: UUID) {
    companion object {
        private const val baseURL = "https://daa.iict.ch"
        private const val enrollURL = "$baseURL/enroll"
        private const val contactsURL = "$baseURL/contacts"
        private const val UUID_KEY = "UUID_KEY"

        suspend fun getOrNewUUID(sharedPreferences: SharedPreferences):
Result<UUID> =
            withContext(Dispatchers.IO) {
                val storedUUID = sharedPreferences.getString(UUID_KEY, null)?.let
{
                    UUID.fromString(it)
                }
                if (storedUUID != null) {
                    Result.success(storedUUID)
                } else {
                    newUUID(sharedPreferences)
                }
            }

        suspend fun newUUID(sharedPreferences: SharedPreferences): Result<UUID> =
            withContext(Dispatchers.IO) {
                runCatching {
                    val uuid = UUID.fromString(enrollURL.toURL().readText())
                    sharedPreferences.edit().putString(UUID_KEY,
uuid.toString()).apply()
                    uuid
```

```
                }
            }
        }

        private fun newRequest(url: URL, method: String): HttpURLConnection =
            (url.openConnection() as HttpURLConnection).apply {
                setRequestProperty("X-UUID", uuid.toString())
                requestMethod = method
            }

        private suspend fun execute(url: URL, method: String): Result<Pair<Int,
    String>> =
            withContext(Dispatchers.IO) {
                newRequest(url, method).runCatching {
                    responseCode to inputStream.bufferedReader().readText()
                }
            }

        private suspend fun execute(
            url: URL,
            method: String,
            payload: String
        ): Result<Pair<Int, String>> = withContext(Dispatchers.IO) {
            newRequest(url, method).runCatching {
                setRequestProperty("Content-Type", "application/json")
                doOutput = true
                outputStream.bufferedWriter().use {
                    it.write(payload)
                }
                responseCode to inputStream.bufferedReader().readText()
            }
        }

        suspend fun getContacts(): List<Contact> = withContext(Dispatchers.IO) {
            val (code, json) = execute(contactsURL.toURL(), "GET").getOrElse {
                return@withContext emptyList<Contact>()
            }

            if (code != HttpURLConnection.HTTP_OK) {
                return@withContext emptyList<Contact>()
            }

            Json.decodeFromString(ListSerializer(Contact.serializer()), json).map {
                it.status = Status.OK
                it.remoteId = it.id
                it.id = null
                it
            }
        }

        suspend fun insert(contact: Contact): Boolean = withContext(Dispatchers.IO) {
            val id = contact.id
            contact.id = null
```

```
        val payload = Json.encodeToString(Contact.serializer(), contact)
        val (code, json) = execute(contactsURL.toURL(), "POST", payload).apply {
            contact.id = id
        }.getOrElse {
            return@withContext false
        }

        if (code != HttpURLConnection.HTTP_CREATED) {
            return@withContext false
        }

        Json.decodeFromString(Contact.serializer(), json).run {
            contact.remoteId = this.id
            contact.status = Status.OK
        }
        true
    }

    suspend fun update(contact: Contact): Boolean = withContext(Dispatchers.IO) {
        val id = contact.id
        contact.id = contact.remoteId

        val payload = Json.encodeToString(Contact.serializer(), contact)
        val (code, json) = execute("$contactsURL/${contact.id!!}".toURL(), "PUT",
payload)
            .apply {
                contact.remoteId = contact.id
                contact.id = id
            }.getOrElse {
                return@withContext false
            }

        if (code != HttpURLConnection.HTTP_OK) {
            return@withContext false
        }

        Json.decodeFromString(Contact.serializer(), json).run {
            contact.id = id
            contact.remoteId = this.id
            contact.status = Status.OK
        }

        true
    }

    suspend fun delete(id: Long): Boolean = withContext(Dispatchers.IO) {
        val (code, _) = execute("$contactsURL/$id".toURL(), "DELETE").getOrElse {
            return@withContext false
        }

        code == HttpURLConnection.HTTP_NO_CONTENT
    }
}
```

```
private fun String.toURL(): URL {
    return URL(this)
}
```

Pour éviter la répétition de code, une méthode privée `newRequest` permet de récupérer une instance de `HttpURLConnection` avec l'UUID déjà ajouté dans les headers. La méthode `execute` est disponible en deux variantes, une avec un payload et une sans. Elle permet d'exécuter une requête HTTP et de retourner le code de retour et le contenu de la réponse. Son exécution est encapsulée dans un `runCatching` pour gérer les exceptions.

Chacune des méthodes `insert`, `update` et `delete` va gérer le status et l'id du contact. Si la synchronisation réussie, le status sera OK sinon ce dernier ne sera pas modifié.

Le fonctionnement des trois méthode est relativement similaire.

Nous avons créé une méthode d'extension pour les chaines de caractères `toURL` permettant d'améliorer la lisibilité du code.