CR Électif PMR - Séquence 2

Qianyan WANG & Baoding LIU

1. Introduction

Cette séquence a pour objectif d'améliorer l'application développée à la séquence 1 en récupérant les listes et les items des utilisateurs depuis une API Rest. Pour réaliser les requêtes HTTP, on utilise la librairie Retrofit.

2. Réalisation des requêtes de l'API

- Accès d'Internet

Pour avoir l'accès d'Internet en utilisant cette application, on ajoute la permission dans le fichier AndroidManifest.xml :

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

- Des indépendances

Pour pouvoir utiliser la librairie Retrofit, on l'ajoute dans le fichier build.gradle, et aussi l'utilisation de coroutine pour pouvoir traiter les requêtes de l'API :

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.8"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.3"

implementation 'com.squareup.retrofit2:retrofit:2.8.1'
implementation "com.squareup.retrofit2:converter-gson:2.8.1"
```

- Les modèles de donnée et les méthodes de requête

Pour pouvoir récupérer les données de l'API, on a besoin des modèles de donnée correspondants. Par exemple, la commande suivante permet de se connecter avec le pseudo et le mot de passe,

```
curl --location --request POST 'http://tomnab.fr/todo-api/authenticate?user=tom&password=web'
```

et elle renvoie des données suivantes :


```
{
   "version": 1.1,
   "success": true,
   "status": 202,
   "hash": "25760fd2eec0ce18a85d998b6e73de1c"
}
```


On a donc besoin d'un modèle de donnée à récupérer les informations utiles, le token de l'utilisateur (« hash ») par exemple.

```
data class GetAuthenticate(
    @SerializedName( value: "success")
    val success: String,
    @SerializedName( value: "hash")
    val token: String,
)
```

Le @SerializedName correspond au label dans le texte reçu.

Dans l'interface ToDoService, on définit aussi une méthode qui permet de réaliser cette requête :

```
// to sign in
@POST( value: "http://tomnab.fr/todo-api/authenticate")
suspend fun signIn(
    @Query( value: "user") login: String,
    @Query( value: "password") password: String
): GetAuthenticate
```

Le @POST indique le type de cette requête. Le @Query correspond dans la requête les paramètres après « ? ». Donc on peut compléter la requête dynamiquement à l'aide des paramètres de cette méthode.
Cette méthode renvoie un modèle de donnée qui stock les informations reçues.

Pour les autres requêtes, on fait les mêmes choses pour les réaliser et puis stocker les informations.

- Obtenir les listes de l'utilisateur

```
curl --location --request GET 'http://tomnab.fr/todo-api/lists?hash=b10ab07311337e6484153b0f5793d516'
```

```
{
    "version": 1.1,
    "success": true,
    "status": 200,
    "lists": [
        {
            "id": "2",
            "label": "list_tom"
        },
        {
            "id": "508",
            "label": "Arnold"
        },
```


```
data class GetLists(
    @SerializedName( value: "success")
    val success: String,
    @SerializedName( value: "lists")
    val lists: List<OneList>,
)

data class OneList(
    @SerializedName( value: "id")
    val id: String,
    @SerializedName( value: "label")
    val label: String
)
```

```
// load lists
@GET( value: "http://tomnab.fr/todo-api/lists")
suspend fun getLists(@Query( value: "hash") token: String): GetLists
```

- Obtenir les items d'une liste de l'utilisateur

```
curl --location --request GET 'http://tomnab.fr/todo-api/lists/3/items' \
--header 'hash: b10ab07311337e6484153b0f5793d516'
```

```
{
    "version": 1.1,
    "success": true,
    "status": 200,
    "items": [
        {
            "id": "1066",
            "label": "d",
            "url": null,
            "checked": "0"
        },
        {
            "id": "1077",
            "label": "tomate",
            "url": null,
```


```
data class GetItems(
    @SerializedName( value: "success")
    val success: String,
    @SerializedName( value: "items")
    val lists: List<OneItem>,
)

data class OneItem(
    @SerializedName( value: "id")
    val id: String,
    @SerializedName( value: "label")
    val label: String,
    @SerializedName( value: "url")
    val url: String?,
    @SerializedName( value: "checked")
    var checkedStr: String,
)
```

```
// load items
@GET( value: "http://tomnab.fr/todo-api/lists/{list_id}/items")
suspend fun getItems(@Path( value: "list_id") listId: Int, @Query( value: "hash") token: String): GetItems
```

- Ajouter une nouvelle liste

```
curl --location --request POST 'http://tomnab.fr/todo-api/lists?label=list3_tom' \
--header 'hash: b10ab07311337e6484153b0f5793d516'
```


```
{
    "version": 1.1,
    "success": true,
    "status": 201,
    "list": {
        "id": "602",
        "label": "nouvelle liste"
    }
}
```


```
data class GetAddList(
    @SerializedName( value: "success")
    val success: String,
    @SerializedName( value: "list")
    val aList: OneList,
)
```

```
// add new list
@POST( value: "http://tomnab.fr/todo-api/lists")
suspend fun addList(@Query( value: "label") label: String, @Query( value: "hash") token: String): GetAddList
```

- Ajouter un nouvel item

```
curl --location --request POST 'http://tomnab.fr/todo-api/lists/3/items?label=Nouvel%20item' \
--header 'hash: b10ab07311337e6484153b0f5793d516'
```

```
{
   "version": 1.1,
   "success": true,
   "status": 201,
   "item": {
      "id": "1431",
      "label": "nouvel item",
      "checked": "0",
      "url": null
   }
}
```


```
data class GetAddCheckItem(
    @SerializedName( value: "success")
    val success: String,
    @SerializedName( value: "item")
    val aItem: OneItem,
)
```

```
// add new item
@POST( value: "http://tomnab.fr/todo-api/lists/{list_id}/items")
suspend fun addItem(
    @Path( value: "list_id") listId: Int,
    @Query( value: "label") label: String,
    @Query( value: "hash") token: String
): GetAddCheckItem
```

- Changer le statut d'un item (cocher ou décocher)

```
curl --location --request PUT 'http://tomnab.fr/todo-api/lists/3/items/4?check=1' \
--header 'hash: b10ab07311337e6484153b0f5793d516'
```

```
{
   "version": 1.1,
   "success": true,
   "status": 200,
   "item": {
      "id": "1431",
      "label": "nouvel item",
      "checked": "1",
      "url": null
   }
}
```


```
data class GetAddCheckItem(
    @SerializedName( value: "success")
    val success: String,
    @SerializedName( value: "item")
    val aItem: OneItem,
)
```

```
// change status of a item
@PUT( value: "http://tomnab.fr/todo-api/lists/{list_id}/items/{item_id}")
suspend fun changeItem(
    @Path( value: "list_id") listId: Int,
    @Path( value: "item_id") itemId: Int,
    @Query( value: "check") check: String,
    @Query( value: "hash") token: String
): GetAddCheckItem
```

Le @Path va remplacer dans la requête les éléments corresponds.

Pour réaliser les requêtes à l'aide de cette librairie en utilisant les méthodes dans l'interface, on crée une classe *DataProvider* :

```
object DataProvider {


    private val BASE_URL = "http://tomnab.fr/todo-api/"

    private val retrofit = Retrofit.Builder()
        .baseUrl(BASE_URL)
        .addConverterFactory(GsonConverterFactory.create())
        .build()


    private val service = retrofit.create(ToDoService::class.java)
```

Et dans laquelle on écrit des méthodes correspondes pour les utiliser dans les Activity.

```
// load lists
suspend fun getLists(token: String): GetLists {
    return service.getLists(token)
}

// sign in and get hash value
suspend fun signIn(login: String, password: String): GetAuthenticate {
    return service.signIn(login, password)
}

// load items
suspend fun getItems(listId: String, token: String): GetItems {
    return service.getItems(listId.toInt(), token)
}

// add a new list
suspend fun addList(label: String, token: String): GetAddList {
    // replace blank by %20
    label.replace( oldValue: " ",  newValue: "%20")
    return service.addList(label, token)
}

// add a new item
suspend fun addItem(listId: String, label: String, token: String): GetAddCheckItem {
    // replace blank by %20
    label.replace( oldValue: " ",  newValue: "%20")
    return service.addItem(listId.toInt(), label, token)
}

// check or uncheck a item
suspend fun changeItem(
    listId: String,
    itemId: String,
    check: String,
    token: String
): GetAddCheckItem {
    return service.changeItem(listId.toInt(), itemId.toInt(), check, token)
}
```

Le mot « suspend » signifie que la méthode peut s'exécuter dans un coroutine et qui ne va pas bloquer le UI thread.

3. Changement dans le code source et l'interface (UI)

- MainActivity

On ajoute un EditText dans le xml fichier de MainActivity pour permettre à l'utilisateur d'entrer son mot de passe et de se connecter avec ce mot de passe.

En cliquant sur le bouton OK, on fait appeler la méthode de se connecter à l'API, mais on ne veut pas qu'il se déroule sans bloquant le UI thread, puisqu'on a besoin de la valeur hash renvoi par cette méthode pour faire des autres activités. On utilise donc le *runBloking* et la méthode *join()* pour le réaliser.

```
// get authentication, block Main thread before finished, save token of user
runBlocking {  this: CoroutineScope
    Log.d(CAT,  msg: "runBlocking...  currentThread: ${Thread.currentThread().name}")
    val job = launch {  this: CoroutineScope
        try {
              // test use
              val gets = DataProvider.signIn("liu", "baoding")
            val gets = DataProvider.signIn(login, password)
            val success = gets.success.toBoolean()
            this@MainActivity.success = success
            val token = gets.token
            this@MainActivity.token = token
            // save token of user
            // To save token for all users (check the cb or not), this
            // can't be in the same PreferenceCategory in the XML file
            editor.putString("edtToken", token)
            editor.commit()
        } catch (e: Exception) {
            // reset the field success
            this@MainActivity.success = false
            ToastUtil.newToast( context: this@MainActivity,  content: "ERROR!" + "\n" + "${e.message}")
            Log.i(CAT,  msg: "${e.message}")
        }
    }
    job.join()
}
```

Si la connexion réussit, on fait passer à l'activité suivante, sinon, l'utilisateur est informe :

```
if (!success) {
    ToastUtil.newToast( context: this@MainActivity,  content: "Incorrect password!")
} else {
    // if authenticated
    ToastUtil.newToast( context: this@MainActivity,  content: "Welcome $login!")

    // start activity
    startActivity(intent)
    Log.i(CAT,  msg: "start activity!")
}
```

Uni fois l'application est lancée, on vérifie si le réseau est utilisable.

```
// checkout network
private fun verifReseau(): Boolean {
    // On vérifie si le réseau est disponible,
    // si oui on change le statut du bouton de connexion
    val cnMngr = getSystemService(CONNECTIVITY_SERVICE) as ConnectivityManager
    val netInfo = cnMngr.activeNetworkInfo
    var sType = "Aucun réseau détecté"
    var bStatut = false
    if (netInfo != null) {
        val netState = netInfo.state
        if (netState.compareTo(NetworkInfo.State.CONNECTED) == 0) {
            bStatut = true
            when (netInfo.type) {
                ConnectivityManager.TYPE_MOBILE -> sType = "Réseau mobile détecté"
                ConnectivityManager.TYPE_WIFI -> sType = "Réseau wifi détecté"
            }
        }
    }
    Log.i(CAT, sType)
    return bStatut
}
```

Si le réseau est utilisable, le bouton OK est activé pour pouvoir se connecter, sinon, il est désactivé.

```
// onResume, checkout network
override fun onResume() {
    super.onResume()
    // S'il n'y a pas de reseau, on désactive le bouton
    btnOK.isEnabled = verifReseau()
}
```

Pour le montrer, quand on se met en mode avion, le bouton ne peut pas être cliquéé.

Pour pouvoir réutiliser la valeur hash (le token) de l'utilisateur dans les différentes activités sans leur passer cette valeur par *Intent*, on la stock dans les préférences :

```
// save token of user
// To save token for all users (check the cb or not), this
// can't be in the same PreferenceCategory in the XML file
editor.putString("edtToken", token)
editor.commit()
```

- ChoixListActivity

Dans cette activité, on fait afficher les listes de l'utilisateur connecté. On a besoin donc du token de cet utilisateur.

```
this.sp = PreferenceManager.getDefaultSharedPreferences( context: this)
this.token = this.sp.getString( key: "edtToken",  defValue: "defString").toString()
```

On a aussi besoin de lancer cette démarche dans un coroutine, c'est-à-dire de ne pas bloquer le UI thread, sinon l'application va crasher.

- Créateur des coroutines

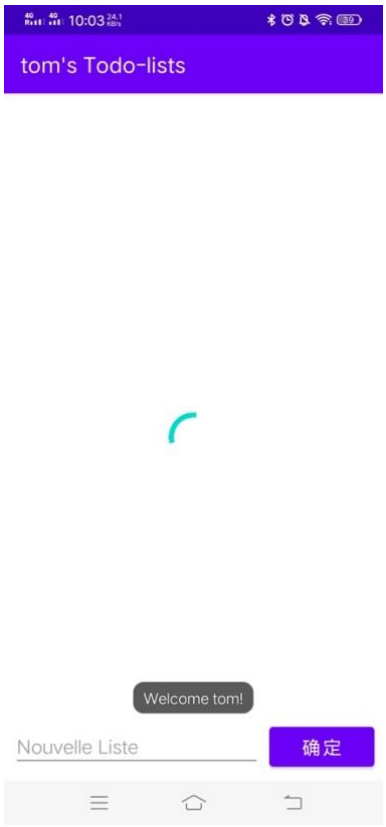
```
// coroutine
private val activityScope = CoroutineScope(
        context: SupervisorJob() +
                Dispatchers.Main
)
```

- Le coroutine pour obtenir les listes

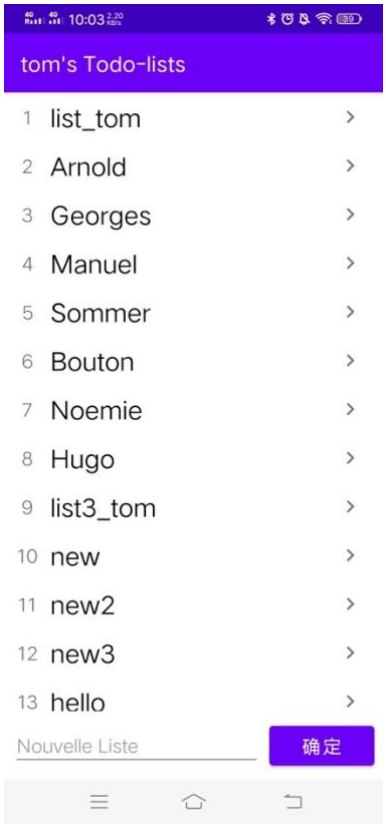
```
// get lists from API
private fun loadLists() {
    activityScope.launch {  this: CoroutineScope
        showProgress( show: true)
        try {
            val lists = DataProvider.getLists(token).lists
            for (list in lists) {
                this@ChoixListActivity.adapter.addData(list)
            }
            Log.i(CAT, lists.toString())
        } catch (e: Exception) {
            ToastUtil.newToast( context: this@ChoixListActivity,  content: "${e.message}")
            Log.i(CAT,  msg: "${e.message}")
        } finally {
            Log.i(CAT,  msg: "LOAD DONE")
        }
        showProgress( show: false)
    }
}
```

La requête de l'API est une opération relativement lente, on fait afficher une icône de chargement pendant l'obtention des data.

```
// display progress bar when loading data
private fun showProgress(show: Boolean) {
    val progress = progress
    val list = reViewList
    progress.isVisible = show
    list.isVisible = !show
}
```



Après avoir fini l'obtention des data, on les affiche.



Cette activité permet aussi d'ajouter une nouvelle liste, en entrant le nom de liste et en cliquant le bouton OK.

- Le coroutine pour ajouter une nouvelle liste

```
// add a new list to API
private fun addList(label: String, token: String) {
    activityScope.launch {  this: CoroutineScope
        showProgress( show: true)
        try {
            val returnList = DataProvider.addList(label, token).aList
            this@ChoixListActivity.adapter.addData(returnList)
            Log.i(CAT, returnList.toString())
        } catch (e: Exception) {
            ToastUtil.newToast( context: this@ChoixListActivity,  content: "${e.message}")
            Log.i(CAT,  msg: "${e.message}")
        } finally {
            Log.i(CAT,  msg: "ADD DONE")
        }
        showProgress( show: false)
    }
}
```

- Réalisation de l'ajoute à l'aide de la méthode *addList* de l'adapter

```
btnOKList.setOnClickListener {  it: View!
    val newListName = etNewList.text.toString()
    if (newListName == "") {
        ToastUtil.newToast(context,  content: "Please enter the name of list")
    } else {
        ToastUtil.newToast(context,  content: "Add \"$newListName\"")

        // add new list
        addList(newListName, this.token)

        etNewList.setText("") //clear the input area
    }
}
```

Quand on clique sur une liste, on passera à l'activité suivante pour afficher les items dans cette liste, en lui passant l'id de la liste cliquée. Pour le réaliser pour toutes les listes ainsi que les listes ajoutées, on réalise un *OnItemClickListener* dans l'adapter des listes.

```
// action when click on the list
adapter.setOnItemClickListener(object : NewListAdapter.OnItemClickListener {
    override fun onItemClick(position: Int) {
        val listName = lists[position].label
        val listId = lists[position].id
        ToastUtil.newToast(context,  content: "this is $listName")
        // pass the clicked list
        intent.putExtra( name: "listName", listName)
        intent.putExtra( name: "listId", listId)
        startActivity(intent)
    }
})
```

- ShowListActivity

Dans cette activité, on fait afficher les items dans la liste cliquée dans l'activité précédente, et elle permet aussi d'ajouter un nouvel item et de changer le statut d'un item en le cochant ou décochant. De même que l'activité précédente, on utilisera des coroutines.

- Créateur des coroutines

```
// coroutine
private val activityScope = CoroutineScope(
    context: SupervisorJob() +
            Dispatchers.Main
)
```

- Le coroutine pour obtenir les items de la liste cliquée

```
// get data from API
private fun loadItems() {
    activityScope.launch {  this: CoroutineScope
        showProgress( show: true)
        try {
            Log.i(CAT, this@ShowListActivity.listId)
            Log.i(CAT, this@ShowListActivity.token)
            val items = DataProvider.getItems(
                this@ShowListActivity.listId,
                this@ShowListActivity.token
            ).lists
            Log.i(CAT, items.toString())
            for (item in items) {
                this@ShowListActivity.adapter.addData(item)
            }
            Log.i(CAT, items.toString())
        } catch (e: Exception) {
            ToastUtil.newToast( context: this@ShowListActivity,  content: "${e.message}")
            Log.i(CAT,  msg: "${e.message}")
        } finally {
            Log.i(CAT,  msg: "LOAD DONE")
            // store the old item status map
            for ((key, value) in adapter.checkStatus) {
                this@ShowListActivity.oldMap[key] = value
            }
        }
        showProgress( show: false)
    }
}
```

- Le coroutine pour ajouter un nouvel item

```
// add a new item to API
private fun addItem(listId: String, label: String, token: String) {
    activityScope.launch {  this: CoroutineScope
        showProgress( show: true)
        try {
            val returnItem = DataProvider.addItem(listId, label, token).aItem
            this@ShowListActivity.adapter.addData(returnItem)
            Log.i(CAT, returnItem.toString())
        } catch (e: Exception) {
            ToastUtil.newToast( context: this@ShowListActivity,  content: "${e.message}")
            Log.i(CAT,  msg: "${e.message}")
        } finally {
            Log.i(CAT,  msg: "ADD DONE")
        }
        showProgress( show: false)
    }
}
```

- Le coroutine pour changer le statut de l'item en le cochant ou décochant

```
// change item status
private fun changeItem(listId: String, itemId: String, check: String, token: String) {
    activityScope.launch {  this: CoroutineScope
          showProgress(true)
        try {
            val returnItem = DataProvider.changeItem(listId, itemId, check, token).aItem
            Log.i(CAT, returnItem.toString())
        } catch (e: Exception) {
            ToastUtil.newToast( context: this@ShowListActivity,  content: "${e.message}")
            Log.i(CAT,  msg: "${e.message}")
        } finally {
            Log.i(CAT,  msg: "ADD DONE")
        }
          showProgress(false)
    }
}
```

De même, pour les opérations qui prennent du temps de faire et qui ont besoin d'afficher les résultats, on fait afficher une icône de chargement pendant ces opérations.

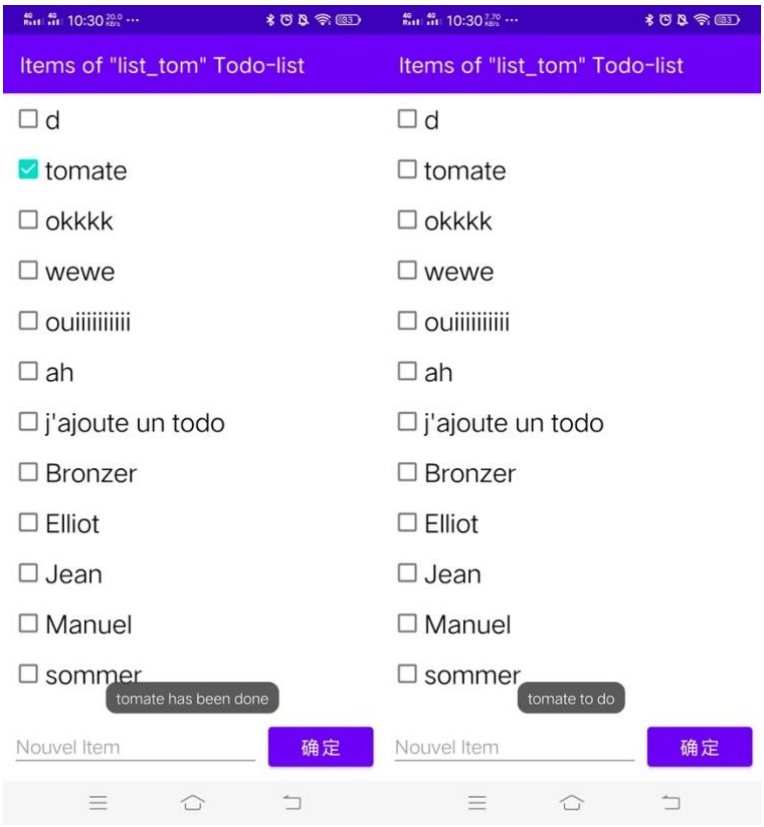
En entrant le nom de l'item et en cliquant sur le bouton OK, on peut ajouter un nouvel item en appelant la méthode *addItem* de l'adapter :

```
btnOKItem.setOnClickListener {  it: View!
    val newItemName = etNewItem.text.toString()
    if (newItemName == null || newItemName == "") {
        ToastUtil.newToast(context,  content: "Please enter the name of item")
    } else {
        ToastUtil.newToast(context,  content: "Add \"$newItemName\"")

        // add new item
        addItem(this.listId, newItemName, this.token)

        etNewItem.setText("") // clear the input area
    }
}
```

Pour changer le statut d'un item, on clique sur l'item. On informe l'utilisateur quand il coche ou décoche un item. Et de même on le réalise en ajoutant un *OnCheckedChangeListener* de l'adapter des items pour que ce fonctionne pour tous les items ainsi que les items ajoutés :

```
// action when checked change on the item
this.adapter.setOnCheckedChangeListener(object : NewItemAdapter.OnCheckedChangeListener {
    override fun onCheckedChange(position: Int, isChecked: Boolean) {
        val itemName = items[position].label
        val itemId = items[position].id
        // change in API
        if (isChecked) {
            ToastUtil.newToast(context,  content: "$itemName has been done")
        } else {
            ToastUtil.newToast(context,  content: "$itemName to do")
        }
    }
})
```

Par exemple :



On ne réalise pas le changement de statut de l'item dans l'API quand on le coche ou décoche dans l'application, puisque c'est une opération relativement fréquente et les requêtes de l'API prennent du temps. On réalise tous les changements dans l'API quand l'utilisateur appuie sur le bouton retour.

Pour le réalise, on utilise un HashMap pour stocker les statuts des items obtenus de l'API dans l'initialisation de l'activité. Quand l'utilisateur appuie sur le bouton retour, on compare l'ancien HashMap avec celui a présent, et on réalise les changements pour les items qui ont des différences au niveau de statut.

- Stocker les statuts initiaux

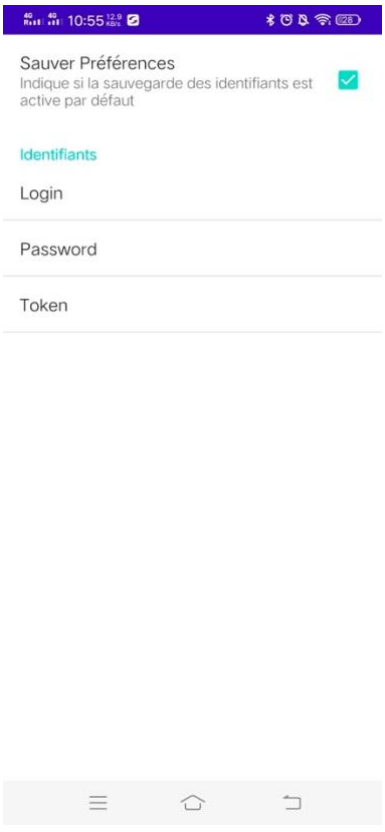
```
// store the old item status map
for ((key, value) in adapter.checkStatus) {
    this@ShowListActivity.oldMap[key] = value
}
```

- Comparer les statuts à présent et réaliser les changements

```
// write all changes in a time when press back button  this: CoroutineScope
override fun onBackPressed() {
    super.onBackPressed()
    activityScope.launch {  this: CoroutineScope
        // compare new status map with old status map
        for ((key, value) in adapter.checkStatus) {
            val itemId = adapter.getDataSet()[key].id
            if (oldMap[key] != value) {
                try {
                    if (value) {
                        changeItem(this@ShowListActivity.listId, itemId, TRUE, token)
                    } else {
                        changeItem(this@ShowListActivity.listId, itemId, FALSE, token)
                    }
                } catch (e: Exception) {
                    Log.i(CAT, e.toString())
                    if (value) {
                        changeItem(this@ShowListActivity.listId, itemId, TRUE, token)
                    } else {
                        changeItem(this@ShowListActivity.listId, itemId, FALSE, token)
                    }
                }
            }
        }
    }
}
```

- SettingsActivity

Dans les préférences, on ajoute le mot de passe et le token de l'utilisateur :



Quand le *CheckBox* est coché, on sauve les préférences de l'utilisateur (le pseudo et le mot de passe), et ces informations sont automatiquement remplies en revenant à l'activité Main. Quand il est décoché, on ne sauve pas ces informations, et elles seront tout effacée en revenant à l'activité Main.

- Sauver ou pas les préférences selon le *CheckBox* est coché ou pas

```
// select menu
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.menu_settings -> {
            ToastUtil.newToast( context: this,  content: "Menu : click on preferences")
            editor.putBoolean("remember", cbRemember.isChecked)
            editor.apply()
            if (!cbRemember.isChecked) {
                editor.putString("login", "")
                editor.putString("password", "")
                editor.apply()
            } else {
                editor.putString("login", edtPseudo.text.toString())
                editor.putString("password", edtPassword.text.toString())
                editor.apply()
            }
            val iGP = Intent( packageContext: this, SettingsActivity::class.java)
            iGP.apply {  this: Intent
                  putExtra("URL","http://tomnab.fr/fixture/")
                startActivity( intent: this)
            }
        }
    }
    return super.onOptionsItemSelected(item)
}
```

- Remettre ou effacer les informations selon le CheckBox est coché ou pas

```
// onStart, load users settings
override fun onStart() {
    super.onStart()
    // relire les preferences partagees de l'app
    val cbR = sp.getBoolean( key: "remember",  defValue: false)

    // actualiser l'etat de la case a cocher
    cbRemember.isChecked = cbR

    // si la case est cochee, on utilise les preferences pour definir le login
    if (cbR) {
        val l = sp.getString( key: "login",  defValue: "login inconnu")
        val p = sp.getString( key: "password",  defValue: "")
        edtPseudo.setText(l)
        edtPassword.setText(p)
    } else {
        // sinon, le champ doit etre vide
        edtPseudo.setText("")
        edtPassword.setText("")
    }
}
```

Pour le token de l'utilisateur, il est utilisé pour les requêtes de l'API, et est différent pour tous les utilisateurs. On l'efface chaque fois quand on revient à l'activité Main, pour garantir la sécurité.

```
// when restart MainActivity, clear the saved token
override fun onRestart() {
    super.onRestart()
    editor.putString("edtToken", "")
    editor.commit()
}
```

- ToastUtil

Comparé à la séquence 1, on ajoute l'utilisation des coroutines. Normalement on ne peut pas faire de *Toast* dans les coroutines, il pose le problème suivant :

```
java.lang.NullPointerException: Can't toast on a thread that has not called
Looper.prepare()
```

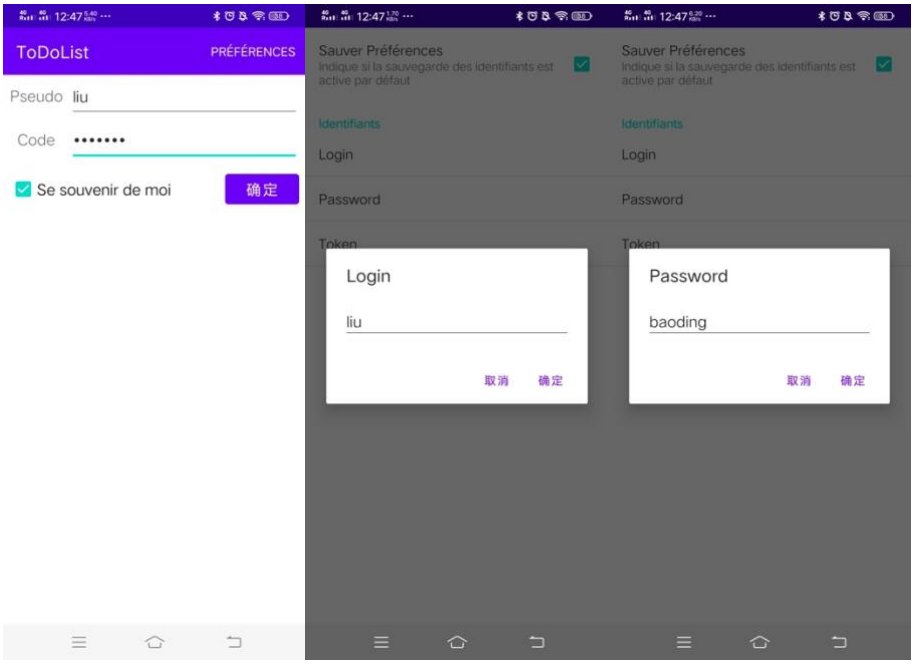
Mais le *Toast* est utile pour le débogage. On fait donc des changements dans la classe ToastUtil pour le repérer. On essaie de faire le Toast, quand il pose ce problème, on faire ce Toast en lui faisant entouré par $Looper.prepare()$ et $Looper.loop()$.

```
try {
    Toast toast = Toast.makeText(context, content, Toast.LENGTH_SHORT);
    toastList.add(toast);
    toast.show();
} catch (Exception e) {
    // to make toast in child thread
    Looper.prepare();
    Toast toast = Toast.makeText(context, content, Toast.LENGTH_SHORT);
    toastList.add(toast);
    toast.show();
    Looper.loop();
}
```

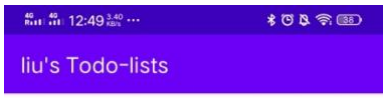
4. Fonctionnement

Après avoir créé un compte dont le pseudo est « liu », le mot de passe est « baoding » dans l'API, on peut bien tester le fonctionnement de notre application.

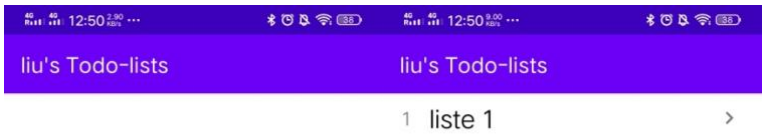
- Saisir le pseudo et le mot de passe



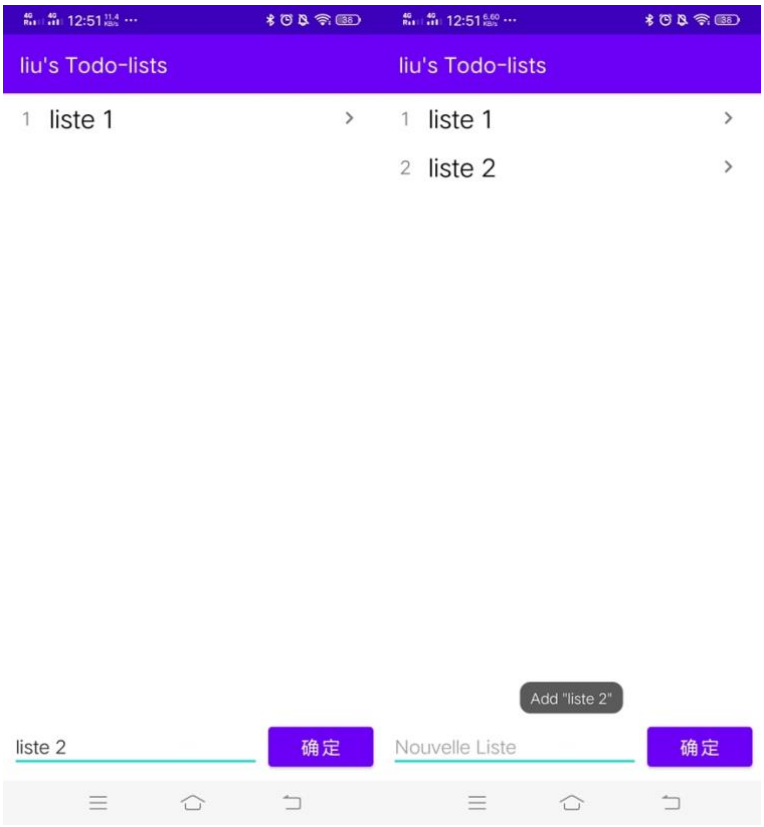
- Cliquer le bouton OK pour se connecter



- Ajouter une nouvelle liste « liste 1 »

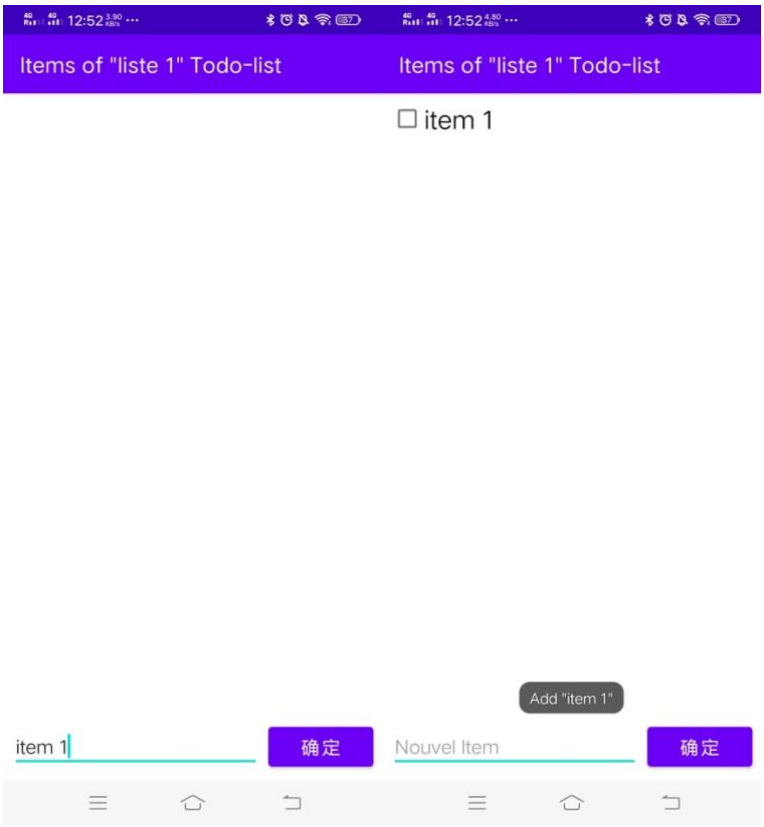


- Ajouter une autre nouvelle liste « liste 2 »

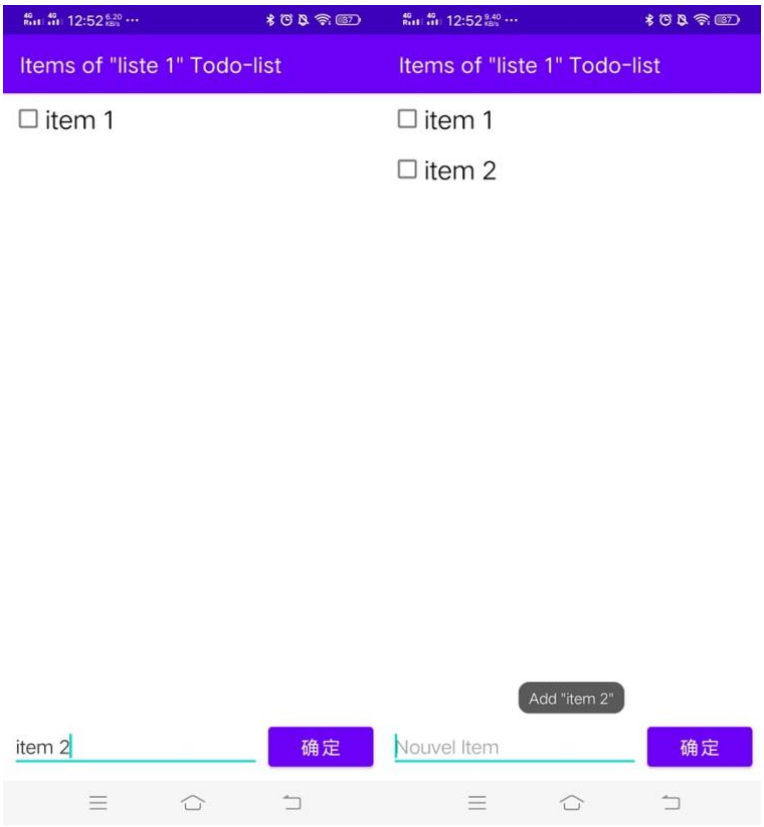


- Cliquer sur la liste « liste 1 »

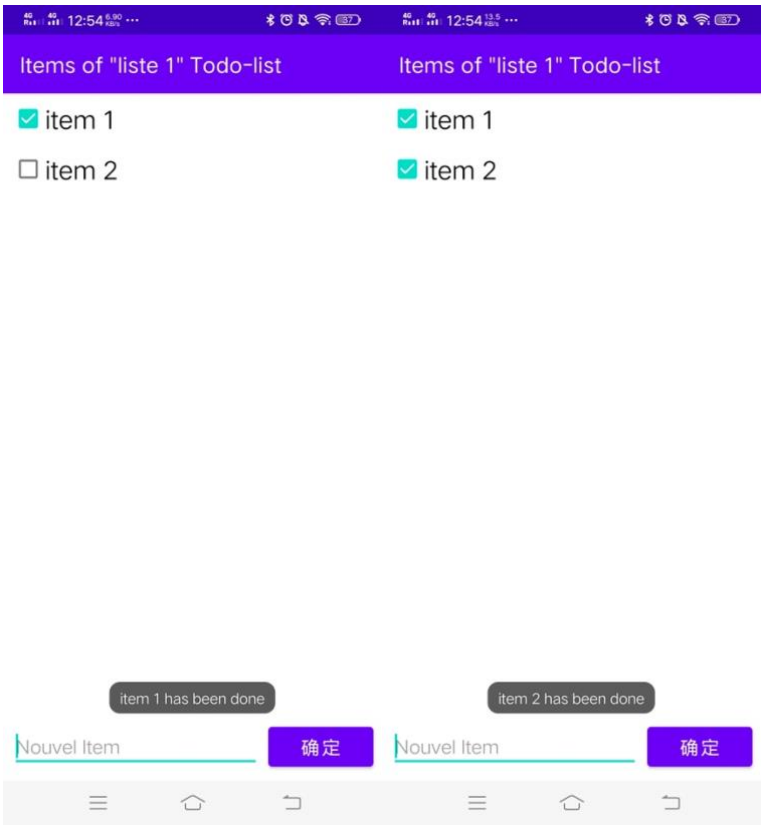
- Ajouter un nouvel item « item 1 »



- Ajouter un autre nouvel item « item 2 »



Cocher ces deux items



Décocher l'item « item 1 »

Appuyer sur le bouton retour pour revenir à l'activité Main, se reconnecter et vérifier

les items de la « liste 1 » dans le compte de l'utilisateur « liu »

5. Conclusion

Dans la version précédente, on utilise le *SharedPreferences* pour stocker les données des utilisateurs, donc toutes les données sont stockées a local. Si on désinstalle l'application, on perd toutes les données.

Dans cette séquence, on a amélioré l'application en utilisant l'API pour stocker les données, qui permet donc de stocker en ligne d'une manière pertinente, et de désinstaller ou se connecter dans un autre appareil sans risque de tout perdre.

En faisant ces améliorations, on a appris comment d'utiliser l'API Rest, et la façon de réaliser les requêtes dans Android avec Kotlin, à l'aide de la librairie *Retrofit*. On a aussi appris comment utiliser les coroutines pour effectuer les opérations qui prennent du temps sans bloquer l'UI thread.

6. Perspectives

- Ajouter une façon compatible avec l'utilisation de l'API de supprimer les listes ou les items
- Donner des avis quand l'utilisateur saisi le pseudo en fonction des comptes dans l'API.

7. Bibliographie

- https://www.delftstack.com/zh/howto/java/java-string-to-boolean/
- https://stackoverflow.com/questions/60175852/cleartext-communication-not-permitted-by-network-security-policy-working-on-my-m
- https://blog.yujinyan.me/posts/understanding-kotlin-suspend-functions/
- https://blog.csdn.net/carson_ho/article/details/73732076
- http://coolaf.com
- https://blog.csdn.net/Leo_Liang_jie/article/details/104989514
- https://blog.csdn.net/qq_36807551/article/details/85316390
- https://www.nhooo.com/kotlin/kotlin-replace-strings.html
- https://blog.csdn.net/qq_37154146/article/details/102837323
- https://blog.csdn.net/qq_31265199/article/details/75499632
- https://www.cnblogs.com/hankzhouAndroid/p/9139757.html
- https://blog.csdn.net/chaoyu168/article/details/88942466
- https://www.jsonla.com/http/test.html