P2017 : Contrôleur de ronde

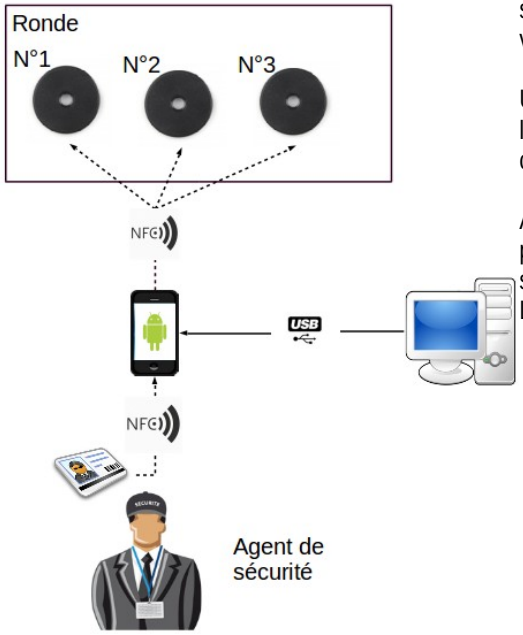
Herbron Tanguy

Dossier technique du projet - partie individuelle

Table des matières

1 - Conception d'une application Android

1.1 - Synoptique de l'application Android



Lorsqu'un agent de sécurité veut effectuer une ronde, il doit tout d'abord s'identifier auprès du smartphone grâce à son badge. Il a ensuite la possibilité de choisir la ronde qu'il souhaite faire via l'application Android.

Une fois la ronde commencée, l'agent de sécurité doit scanner les points de contrôle dans l'ordre, libre à lui d'ajouter une main courante électronique ou une photo en fonction de ce qu'il constate lors de sa ronde.

Après que le dernier point de contrôle ait été scanné, la ronde est terminée et les données peuvent être enregistrées sur le poste de supervision grâce à un câble USB reliant le smartphone sous Android et le poste de supervision.
L'agent a aussi la possibilité de terminer la ronde manuellement en cas de problème.

1.2 - Environnement de développement pour Android

L'application Android a été créée avec le logiciel Android Studio sous le Software Development Kit (SDK) 25 avec comme complément les APIs (Application Programming Interface) bottombar de roughik (https://github.com/roughike/BottomBar) et cameraview-master de google (https://github.com/google/cameraview).

Un problème est survenu avec l'ajout des APIs dans mon programme, elles sont sous licence Apache 2.0. Cette licence stipule qu'en cas de redistribution :

- toute personne en possession d'une copie de l'application doit avoir à sa disposition une copie de cette licence
- je dois indiquer quelles parties du code des APIs j'ai eu à modifier lors de mon développement
- je dois informer l'utilisateur de mon code que je ne suis pas propriétaire de ces APIs
- je dois ajouter à tout fichier NOTICE l'usage de ces APIs et de cette licence

Il est possible d'obtenir une copie de la licence à l'adresse suivante : http://www.apache.org/licenses/LICENSE-2.0

L'application a été développée pour fonctionner sur tous les smartphones entre du SDK 16 (Android 4.1 Jelly Bean) au dernier SDK, version 25 (Android 7.1, Nougat). L'application supporte un grand nombre de smartphone sur le marché mais ne pourra par fonctionner sur l'ensemble de ces derniers. En effet, l'une des conditions d'utilisations de cette application est de posséder un lecteur NFC sur le smartphone.

Dans le cadre de la création de l'interface Android, il a fallu la créer de façon à ce qu'elle soit ergonomique et facile d'usage pour l'utilisateur. Cette création a été grandement facilitée grâce à l'usage des deux APIs mentionnées plus haut.

La conception de cette application a été faite de façon à ce qu'il soit facile de récupérer les données de la base de données du smartphone pour l'exporter sur le poste de contrôle.

1.3 - Descriptif des tâches relatives au smartphone

Au cours de ce projet, j'ai dû développer les fonctionnalités suivantes pour le projet rondier qui se découpe entre les parties sur le smartphone de l'agent, le smartphone de l'administrateur et l'application sur le poste de supervision :

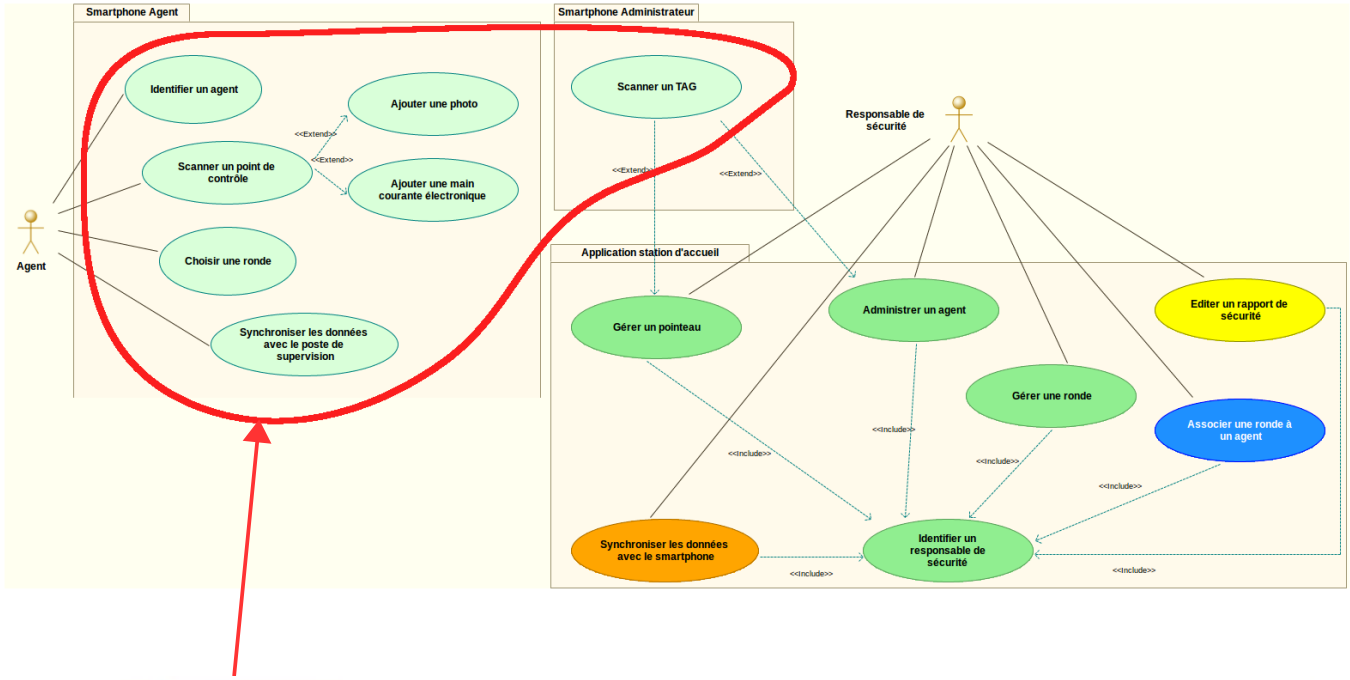
Côté smartphone agent :

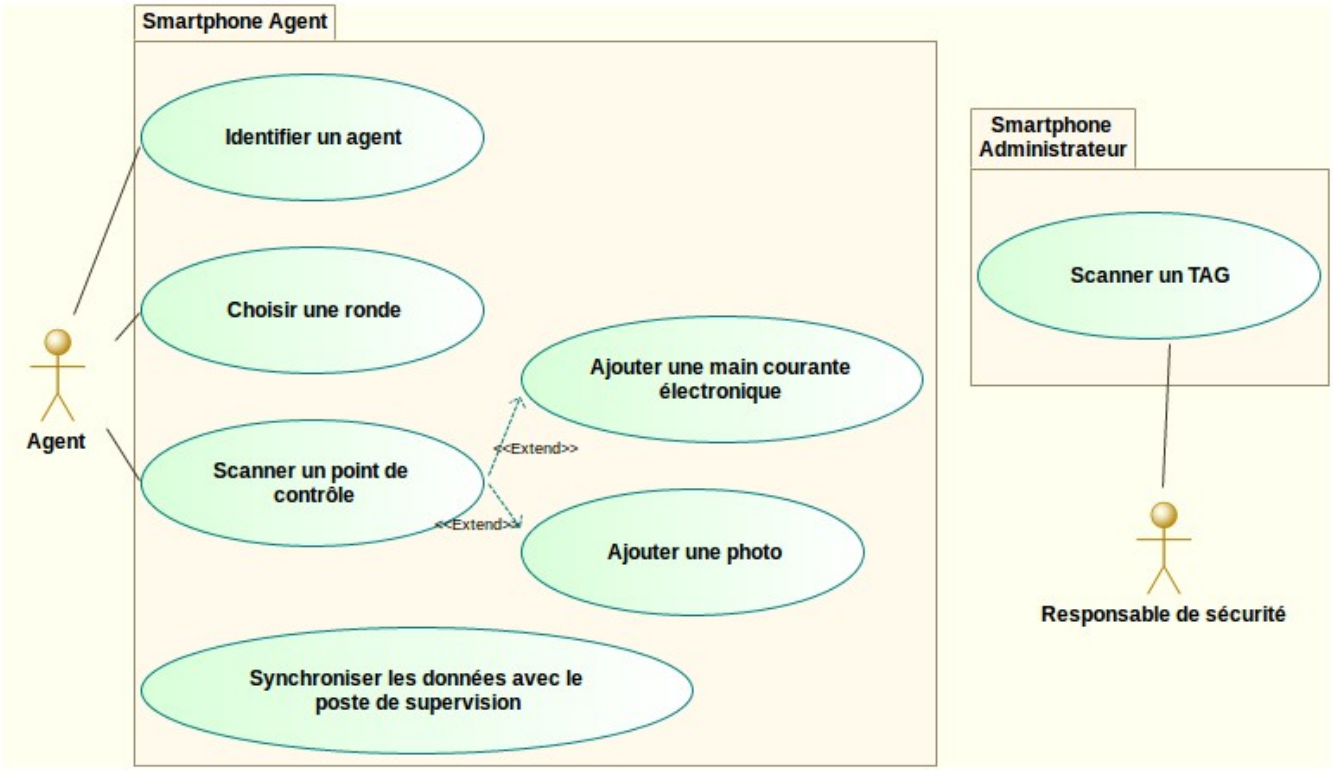
- Identifier un agent
- Choisir une ronde
- Scanner un point de contrôle
- Ajouter une photo à un point de contrôle
- Ajouter une main courante électronique à un point de contrôle
- Synchroniser les données avec le poste de supervision

Côté smartphone administrateur :

- Scanner un tag pour la création de pointeaux sur l'application de supervision

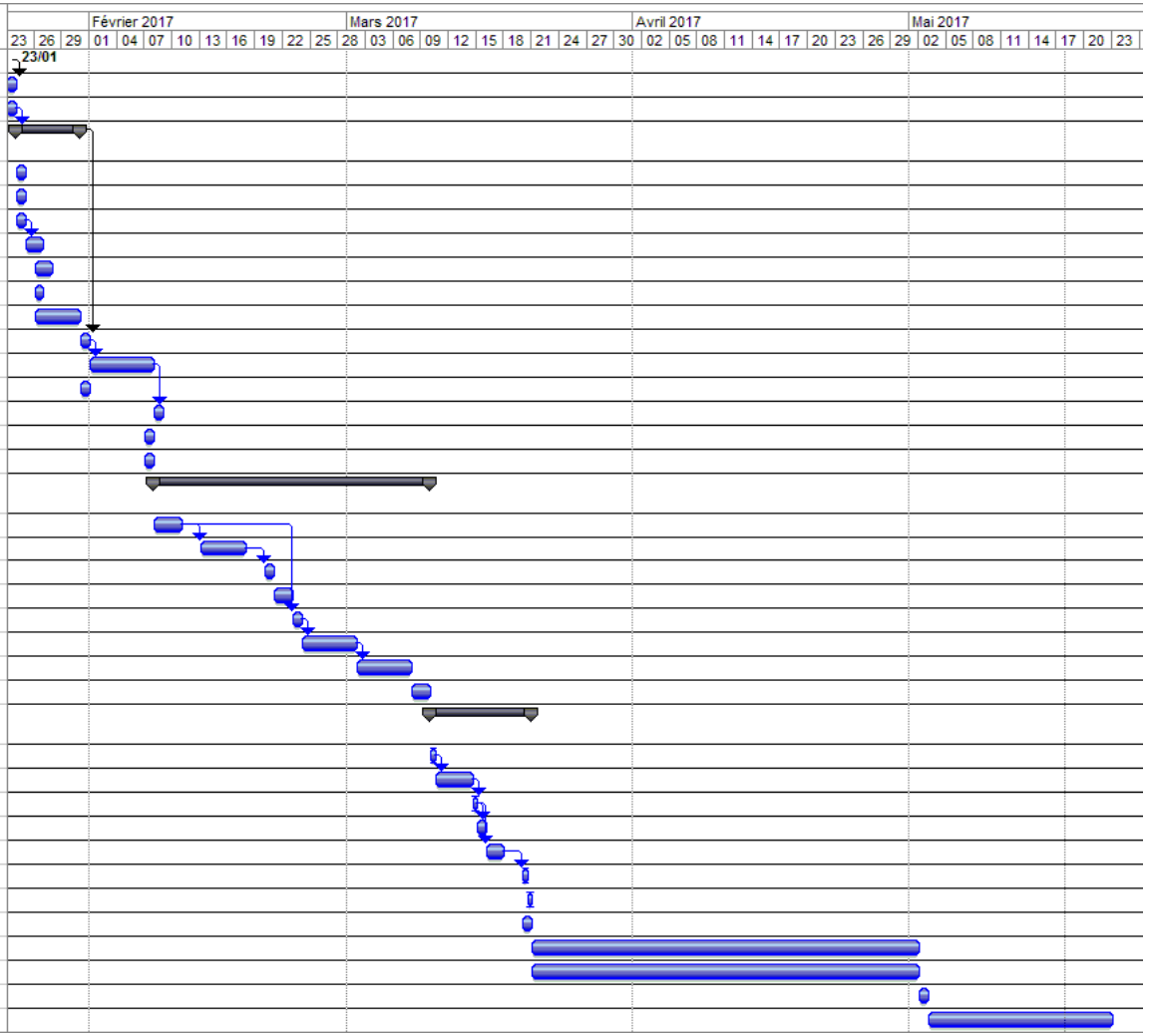
Afin de modéliser la répartition des tâches dans notre projet, nous avons modélisé un diagramme de cas d'utilisations :





Dans le but de mener à bout ma partie du projet sans me perdre dans les tâches à effectuer et terminer, il m'a fallu créer un diagramme de Gantt :

Nom de la tâche	Durée	Début	Fin	Prédécesseurs
Début du projet	0 jour	Lun 23/01/17	Lun 23/01/17	
Prise de connaissance du cahier des charges	1 jour	Lun 23/01/17	Lun 23/01/17	1
Répartition des tâches et planification	1 jour	Lun 23/01/17	Lun 23/01/17	
Définition des cas d'utilisations	**5 jours**	**Mar 24/01/17**	**Lun 30/01/17**	3
Scanner un TAG	1 jour	Mar 24/01/17	Mar 24/01/17	
Scanner un point de contrôle	1 jour	Mar 24/01/17	Mar 24/01/17	
Identifier un agent	1 jour	Mar 24/01/17	Mar 24/01/17	
Choisir une ronde	2 jours	Mer 25/01/17	Jeu 26/01/17	7
Ajouter une photo	2 jours	Jeu 26/01/17	Ven 27/01/17	
Ajouter une main courante électronique	1 jour	Jeu 26/01/17	Jeu 26/01/17	
Synchronisation des données smartphone et poste de supervision	3 jours	Jeu 26/01/17	Lun 30/01/17	
Installation de l'API bottombar	1 jour	Mar 31/01/17	Mar 31/01/17	4
Réalisation des interfaces	5 jours	Mer 01/02/17	Mar 07/02/17	12
Rencontre avec le responsable de l'entreprise	1 jour	Mar 31/01/17	Mar 31/01/17	
Revue de projet - Analyse	1 jour	Mer 08/02/17	Mer 08/02/17	13
Installation de ADB	1 jour	Mar 07/02/17	Mar 07/02/17	
Activation du mode débogage (Android)	1 jour	Mar 07/02/17	Mar 07/02/17	
Développement de l'application Rondier (Android)	**22 jours**	**Mer 08/02/17**	**Jeu 09/03/17**	
Développement du scanner de TAG	3 jours	Mer 08/02/17	Ven 10/02/17	
Mise en place de la base de données	5 jours	Lun 13/02/17	Ven 17/02/17	19
Développement de l'identification d'un agent	1 jour	Lun 20/02/17	Lun 20/02/17	20
Développement du choix d'une ronde	2 jours	Mar 21/02/17	Mer 22/02/17	
Développement du scannage d'un point de contrôle	1 jour	Jeu 23/02/17	Jeu 23/02/17	19
Installation de l'API cameraview-master	4 jours	Ven 24/02/17	Mer 01/03/17	23
Développement de l'ajout d'une photo	4 jours	Jeu 02/03/17	Mar 07/03/17	24
Développement de l'ajout d'une main courante électronique	2 jours	Mer 08/03/17	Jeu 09/03/17	
Développement de la synchronisation entre le smartphone et le poste de supervision	**7 jours**	**Ven 10/03/17**	**Lun 20/03/17**	
Installation de la librairie QDeviceWatcher	0,5 jour	Ven 10/03/17	Ven 10/03/17	
Listage des smartphones connectés au PC	2 jours	Ven 10/03/17	Mar 14/03/17	28
Sélection d'un smartphone	0,5 jour	Mar 14/03/17	Mar 14/03/17	29
Récupération des photos sur le smartphone	1 jour	Mer 15/03/17	Mer 15/03/17	30
Création d'un fichier sqlite et transfert	2 jours	Jeu 16/03/17	Ven 17/03/17	30
Synchronisation des données côté smartphone	0,5 jour	Lun 20/03/17	Lun 20/03/17	32
Récupération des données sur le poste de supervision	0,5 jour	Lun 20/03/17	Lun 20/03/17	
Revue de projet - Conception	1 jour	Lun 20/03/17	Lun 20/03/17	
Fixation de bugs	30 jours	Mar 21/03/17	Mar 02/05/17	
Rédaction du dossier final	30 jours	Mar 21/03/17	Mar 02/05/17	
Test unitaire	1 jour	Mar 02/05/17	Mar 02/05/17	
Mise en commun des programmes et tests finaux	14 jours	Mer 03/05/17	Lun 22/05/17	



2 - Prérequis de développement

2.1 - Environnement de développement

L'application Android a été développée à l'aide d'Android Studio sous le SDK 25. Il est nécessaire d'avoir le dernier SDK disponible pour développer en utilisant les APIs que j'ai dû mettre en œuvre dans l'application.
Afin de monitorer mon smartphone en dehors du développement, il m'a semblé primordial d'avoir accès à l'intégralité du smartphone. J'ai donc changé, grâce au processus nommé rooting détaillé en annexe, les accès qu'un utilisateur a sur tout smartphone Android.
Bénéficier de ces autorisations supplémentaires m'a permis de lire, écrire et manipuler le fichier de base de données de l'application. En effet, sans l'accès root, il est impossible de voir le contenu d'une application (dossier d'installation, dépendances, etc). Il n'est bien sûr pas obligatoire d'avoir un smartphone rooté pour développer cette application mais cela aide grandement comme je le montre par la suite.

Afin de manipuler mon smartphone en ligne de commande à partir de mon PC, il m'a donc fallu installer ADB (Android Debug Bridge) qui permet d'envoyer toutes sortes d'instructions au smartphone via la liaison USB. Ce système de communication me permet notamment de copier des fichiers via un programme facilement, d'installer et lancer des applications ou encore de voir les logs.

2.2 - Installation de ADB

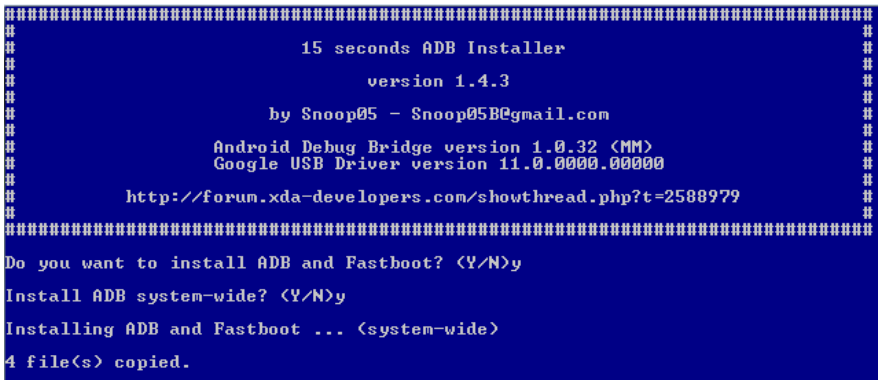
Installation Linux :

Sous Linux, il suffit d'avoir l'accès root et d'exécuter la commande suivante :

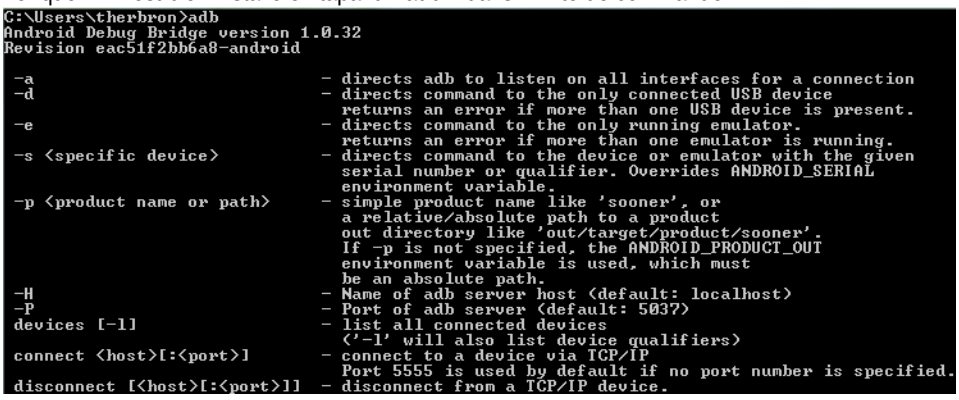
```
apt-get install Android-tools-adb Android-tools-adb
```

Installation Windows :

L'installateur Windows est disponible à l'adresse *https://www.androidfilehost.com/?fid=24591020540821930*, il suffit ensuite de suivre les instructions affichées à l'écran.

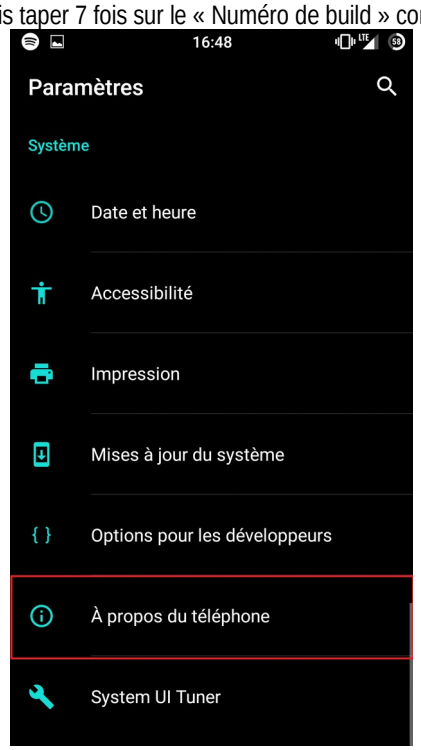


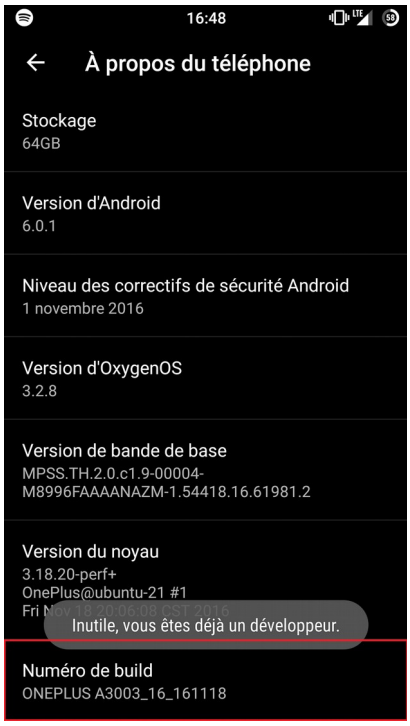
Vous pouvez ensuite vérifier que ADB est bien installé en tapant « adb » dans l'invité de commande :



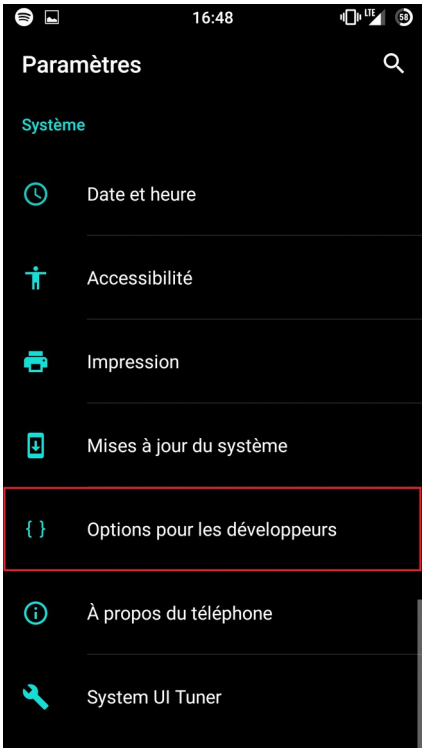
2.3 - Activation du mode Débogage (Android)

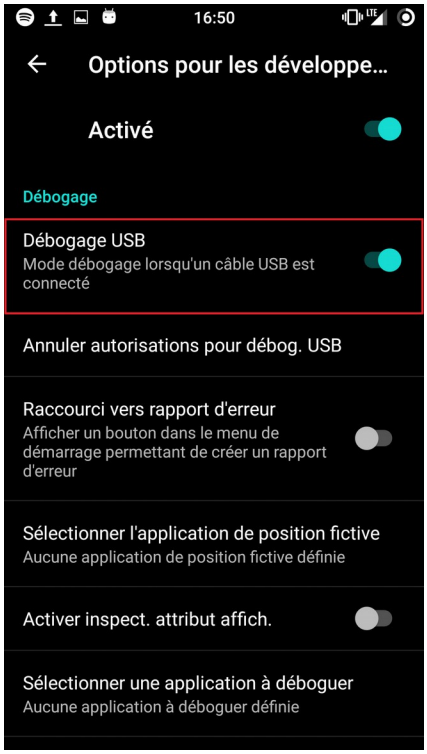
Afin de développer sur mon smartphone, il m'a fallu activer le mode débogage, premièrement pour installer les applications que je créais avec Android Studio puis pour communiquer entre le Smartphone et le PC.
Pour activer le mode de débogage, vous devez vous rendre dans l'application réglage du smartphone, tout en bas dans l'onglet « À propos du téléphone » puis taper 7 fois sur le « Numéro de build » comme présenté ci-dessous :





Il faut ensuite se rendre dans l'onglet « Options pour les développeurs » de l'application paramètre et descendre jusqu'à trouver l'option « Débogage USB » et l'activer.

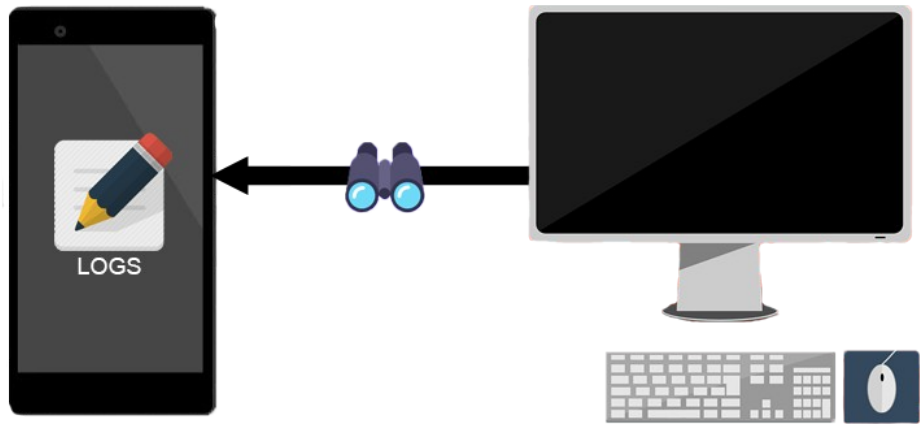




3 - Développement du scannage d'un TAG

3.1 - Conception détaillée

L'une des premièsre tâcshe que j'ai réalisé est la partie permettant à l'administrateur de scanner un TAG MiFare et de l'avoir directement sur le poste de supervision. Afin de transférer les informations entre le smartphone et le poste de supervision, j'ai dû trouver un moyen de communiquer au travers de la liaison USB. J'ai donc utilisé les logs du smartphone que je lis sur le poste de supervision grâce à ADB comme illustré ci-dessous :



Il était donc impératif de pouvoir réagir à la détection d'un TAG MiFare grâce à la puce NFC du téléphone. Il nous faut donc déclarer que l'application utilise le NFC dans le manifeste.

AndroidManifest.xml :

```
1  <uses-permission android:name="android.permission.NFC" />
2
3  <intent-filter>
4      <action android:name="android.nfc.action.TAG_DISCOVERED" />
5      <category android:name="android.intent.category.DEFAULT" />
6  </intent-filter>
```

Ligne 1 : Indique que l'application requiert l'accès au module NFC pour fonctionner.

Lignes 3 & 4 : Indique l'événement qui déclenche la découverte d'un TAG MiFare.

Seulement pour détecter un TAG MiFare, il faut tout d'abord savoir si le téléphone est compatible avec la technologie NFC et si c'est le cas, si la fonctionnalité est activée.

MainActivity.java :

```
1  nfcAdapter = NfcAdapter.getDefaultAdapter(this);
2  if (nfcAdapter == null) {
3   Toast.makeText(this, "NFC non supporté par l'appareil.",
4  Toast.LENGTH_LONG).show();
5  } else {
6    if (!nfcAdapter.isEnabled()) {
7     Toast.makeText(this, "NFC désactivé.", Toast.LENGTH_LONG).show();
8    }
9  }
```

Ligne 1 : Récupération des données sur le module NFC du smartphone.

Ligne 2 : Vérification si le module NFC existe, si ça n'est pas le cas, l'interface de départ de l'application est changée de façon à ce que l'utilisateur soit au courant que le smartphone ne supporte pas le NFC.

Ligne 5 : Si le smartphone dispose d'une puce NFC mais n'est pas activé, l'interface de départ du smartphone est modifiée et l'utilisateur est informé que le NFC doit être activé pour continuer.

Lorsque le NFC est activé.

Si le NFC est désactivé, l'interface est mise à jour.

Après que l'application ait vérifié si le NFC était bien supporté et activé sur le smartphone, il faut créer dans la fonction onResume (pour plus d'information sur l'appel de la fonction onResume, se référer à l'annexe) la condition indiquant la détection d'un tag grâce à la puce NFC.

MainActivity.java :

```
1  String action = intent.getAction();

2  if(NfcAdapter.ACTION_TAG_DISCOVERED.equals(action)) {
3    tagMifare = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
4  }
```

Ligne 1 : Définis la chaîne de caractère qui permet de vérifier qu'une action a été effectuée.

Ligne 2 : Permets de vérifier que l'application appelant la fonction onResume est bien celle correspondant à l'événement décrit plus tôt dans le manifest.

Ligne 3 : Récupération des informations concernant le TAG scanné dans une variable du SDK prévue à cet effet.

Lorsqu'un TAG est donc scanné, les informations le concernant sont inscrites dans les logs du smartphone :

```
Log.i("NouveauTag", tagID);
```

Par la suite, sur l'application de supervision déployée sur le poste de supervision, j'écoute les logs du smartphone grâce à la commande :

```
adb -s <id du smartphone> logcat -c
adb -s <id du smartphone> logcat
```

Le paramètre « -c » de la première ligne permet de vider les logs pour ne pas avoir d'anciennes lectures lorsque l'on démarre la lecture. Le processus tourne ensuite sans interruption et lit toutes les données écrites dans les logs.

J'utilise l'expression régulière suivante pour trier les logs reçus et ne récupérer que l'ID du TAG MiFare ainsi scanné :

```
\sNouveauTag:\s(.*?)\r
```

La partie « \sNouveauTag:\s » signifie que nous cherchons une ligne comportant cette chaîne de caractère. L'ouverture de la parenthèse indique le début de la capture. Je capture ensuite tous les caractères (« . »), pour 0 à un nombre infini d'itérations (« * ») et pour un nombre de caractères infini (« ? »), je finis la capture avant le premier retour ligne.

Le numéro du TAG MiFare est ensuite affiché dans l'application de supervision dans le champ concerné de l'onglet actif.

3.2 - Test unitaire

Conditions initiales	
L'application sur le smartphone est lancée. L'application sur le poste de supervision est lancée. Le smartphone est connecté au poste de supervision grâce à un câble USB.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de l'étape pour scanner un TAG MiFare via une application Android et l'envoyer sur l'application du poste de supervision.	
Opération	**Résultats attendus**
Scanner un TAG MiFare avec le smartphone	L'application Android affiche les informations du TAG. L'ID du TAG apparaît sur l'application du poste de supervision.
Erreur	**Source du problème & Solution**
L'application du poste de supervision affiche des caractères aléatoires ne correspondant pas aux données attendues.	Lorsque les données étaient récupérées, une erreur dans la commande exécutée pour lire les logs faisait qu'il était impossible de ne récupérer que les lignes comportant les IDs. La fonction pipe (« | ») ne pouvait pas être utilisée dans le lancement d'une commande avec QT il m'a donc fallu utiliser l'expression régulière précédemment présentée.

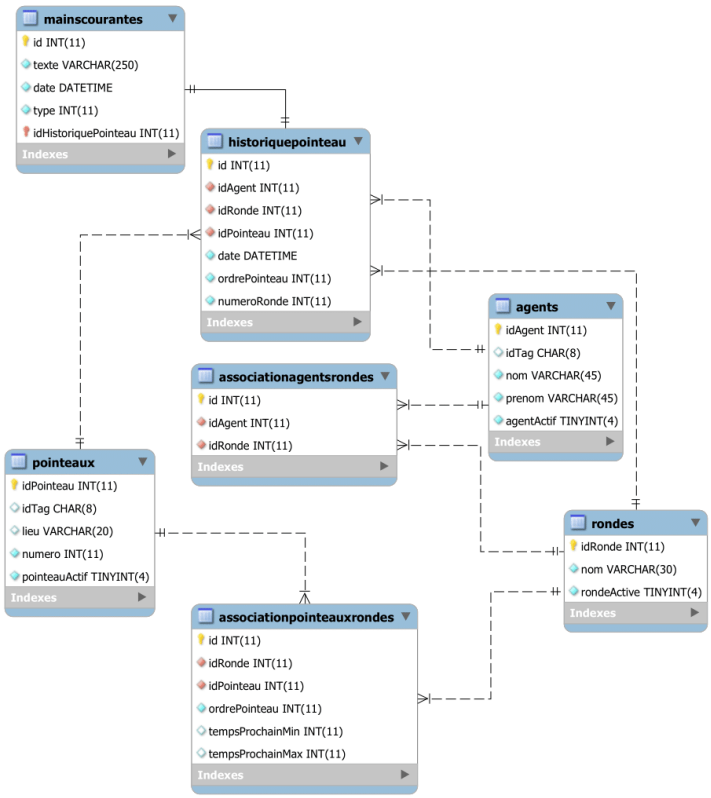
4 - Mise en place de la base de données SQLite pour Android

4.1 - Conception détaillée

Afin de stocker les informations telles que les pointeaux, les rondes, etc, il m'a fallu utiliser le système de base de données SQLite, présent nativement dans l'environnement Android.

J'ai donc commencé par installer le paquet « sqlite3 » sur mon smartphone pour pouvoir avoir une version humainement lisible de la base de données de l'application (nécessite le root). La démarche d'installation est détaillée dans l'annexe.

Après que Gabriel Lemée ait modélisé la base de données, il m'a suffi de la créer dans l'application en suivant le schéma ci-dessous:



Pour créer une table dans le fichier SQLite, il me suffit d'exécuter le code suivant :

GestionBDD.java :

```
1  db.execSQL("CREATE TABLE IF NOT EXISTS " + TABLE_AGENTS + "("
2          + CLE_AGENT_ID + " INTEGER PRIMARY KEY,"
3          + CLE_AGENT_TAG_ID + " VARCHAR(8),"
4          + CLE_NOM_AGENT + " VARCHAR(45),"
5          + CLE_PRENOM_AGENT + " VARCHAR(45)"
6          + ");");
```

Ligne 1 : Crée la table si elle n'existe pas déjà.
Ligne 2 : Insère la colonne ID qui est une clé primaire auto-incrémentée.
Ligne 3 : Insère la colonne stockant l'ID des tags associés aux agents.
Ligne 4 : Insère la colonne stockant le nom des agents.
Ligne 5 : Insère la colonne stockant le prénom des agents.

À chaque instanciation de la classe GestionBDD (voir l'annexe pour le code source complet), chaque table est recréée si elle n'existe pas déjà pour éviter tout problème dans la suite du programme.

Les données sont ensuite transférées sur le smartphone grâce à la partie Synchronisation.

Afin de faciliter la lecture et les mises à jour ultérieures de mon code, il m'a semblé judicieux de mettre le nom des tables et des champs dans des constantes. Exemple :

GestionBDD.java :

```
1  private static final String TABLE_AGENTS = "Agents";
2  private static final String TABLE_RONDES = "Rondes";
3  private static final String TABLE_POINTEAUX = "Pointeaux";
4  private static final String TABLE_HISTORIQUE_POINTEAU = "HistoriquePointeau";
5  private static final String TABLE_MAIN_COURANTE = "MainCourante";
6  private static final String TABLE_ASSOCIATION_POINTEAUX_RONDES =
7  "AssociationPointeauxRondes";
8  private static final String TABLE_ASSOCIATION_AGENTS_RONDES =
9  "AssociationAgentsRondes";
```

Attention : si vous souhaitez mettre à jour la base de données, il est nécessaire d'indenter de 1 la constante DATABASE_VERSION et de relancer l'application. À la prochaine instanciation de la classe, la base de données sera mise à jour grâce à la fonction onUpgrade du système Android qui vérifie les numéros de version des bases de données.

GestionBDD.java :

```
1  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
2      // Supprime les anciennes tables s'il y en a
3      db.execSQL("DROP TABLE IF EXISTS " + TABLE_AGENTS);
4      db.execSQL("DROP TABLE IF EXISTS " + TABLE_RONDES);
5      db.execSQL("DROP TABLE IF EXISTS " + TABLE_POINTEAUX);
6      db.execSQL("DROP TABLE IF EXISTS " + TABLE_HISTORIQUE_POINTEAU);
7      db.execSQL("DROP TABLE IF EXISTS " + TABLE_MAIN_COURANTE);
8      db.execSQL("DROP TABLE IF EXISTS " + TABLE_ASSOCIATION_POINTEAUX_RONDES);
9      db.execSQL("DROP TABLE IF EXISTS " + TABLE_ASSOCIATION_AGENTS_RONDES);
10     // Crée les nouvelles tables
11     onCreate(db);
12 }
```

5 - Réalisation de l'identification d'un agent

5.1 - Conception détaillée

Lorsque l'application est lancée, il est demandé à l'utilisateur de scanner un badge. Si des données sont enregistrées, il est possible à un agent de s'identifier grâce à un badge d'identification fait à partir d'un TAG MiFare.

Lorsque l'agent scanne son badge, l'application va chercher dans la base de données si l'agent est enregistré.

GestionBDD.java :

```
1  String searchQuery = "SELECT * FROM " + TABLE_AGENTS + " WHERE " +
2  CLE_AGENT_TAG_ID + " = \"" + badgeID + "\"";
3  Cursor resultat = db.rawQuery(searchQuery, null);

4  Agent agent;

5  if(resultat.moveToFirst())
6  {
7     agent = new Agent(resultat.getInt(0), resultat.getString(1),
8     resultat.getString(2), resultat.getString(3));
9  }
```

Ligne 1 : Définition de la requête SQL qui cherche l'agent correspondant au tag passé en paramètre de la fonction.

Ligne 3 : Exécution de la requête et récupération des résultats dans une variable.

Ligne 7 : Si un résultat est récupéré, je crée un agent avec les informations trouvées dans la base de données.

Si un agent est trouvé, l'application charge le choix de ronde. Si ça n'est pas le cas, l'utilisateur est informé qu'il n'est pas dans la base de données.

GestionBDD.java :

```
1  agent = bdd.obtenirAgentAvecID(tagID);

2  if(agent != null)
3  {
4      Toast.makeText(this, "Utilisateur trouvé : " + agent.obtenirNom(),
5      Toast.LENGTH_LONG).show();

6      startActivity(activiteChoixRonde);
7  }
8  else
9  {

10     Toast.makeText(this, "Utilisateur non reconnu", Toast.LENGTH_LONG).show();
11 }
```

Ligne 2 : Vérification de l'existence de l'agent.

Ligne 5 : Lancement de l'activité de choix de ronde.

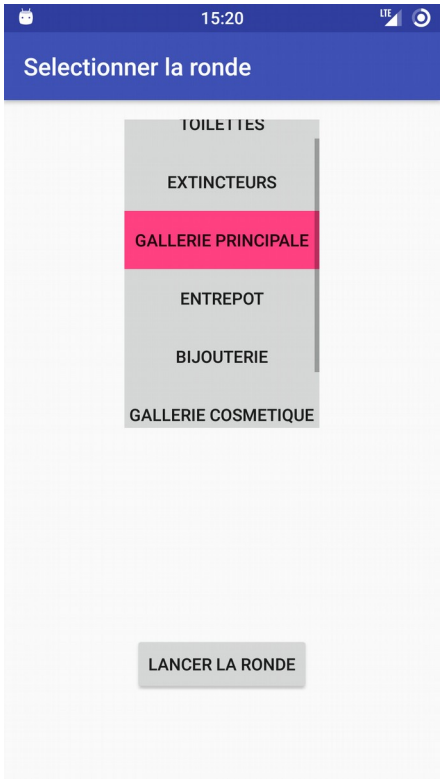
5.2 - Test unitaire

Conditions initiales	
L'application sur le smartphone est lancée. Un agent doit avoir un TAG MiFare enregistré dans la base de données.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de l'étape d'identification d'un agent sur l'application Android.	
Opération	**Résultats attendus**
Scanner un badge d'identification enregistré sur le smartphone.	Pendant 6 secondes le message « Utilisateur trouvé : <Nom de l'agent » est affiché. L'interface de choix de ronde est affichée.
Scanner un badge d'identification non enregistré sur le smartphone.	Pendant 6 secondes le message « Utilisateur non reconnu » est affiché. L'utilisateur peut rescanner un badge.
Erreur	**Source du problème & Solution**
Aucune erreur décelée.	

6 - Réalisation du choix de ronde

6.1 - Conception détaillée

Après que l'agent se soit identifié, l'interface ci-dessous permettant le choix de la ronde est affichée :



Lorsque des rondes sont disponibles, elles sont affichées dans une liste déroulante.

Cette interface est dynamiquement générée en fonction d'une requête effectuée dans la base de données, plus précisément dans la table « AssociationAgentsRondes ».

La requête suivante retourne tous les IDs de rondes associées à l'ID de l'agent passé en paramètre :

```
"SELECT * FROM " + TABLE_ASSOCIATION_AGENTS_RONDES + " WHERE " + CLE_AAR_AGENT_ID + " =
\"" + agent.obtenirId() + "\""
```

Je stocke ensuite tous les résultats dans une variable, comme vu précédemment, et j'effectue le code suivant pour récupérer l'ID de toutes les rondes associées :

```
   GestionBDD.java :
1  listeIDRonde = new ArrayList<Integer>();
2  do {
3      listeIDRonde.add(resultat.getInt(2));
4  }while(resultat.moveToNext());
```

Ligne 1 : La liste comportant tous les IDs des rondes associées à l'agent.

Ligne 3 : Pour chaque résultat, nous ajoutons dans la liste l'ID de ronde correspondant.

Il me suffit ensuite de récupérer les noms de chaque ronde grâce à leur ID et de générer une liste déroulante dans l'interface de l'application.

Lorsqu'un agent sélectionne une ronde, il doit ensuite valider son choix pour être amené vers l'interface Rondier principale qui permet d'effectuer une ronde.

6.2 - Test unitaire

Conditions initiales	
L'agent doit être identifié. L'agent doit avoir des rondes associées.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de l'étape de choix de ronde dans l'application Android.	
Opération	**Résultats attendus**
Appuyer sur le bouton valider sans sélectionner de ronde.	Pendant 6 secondes le message « Choisissez une ronde » est affiché.
Appuyer sur le bouton valider après avoir sélectionné une ronde.	L'interface Rondier est affichée. Le déclenchement d'une ronde est enregistré dans l'application. Le nom de la ronde en cours est transmis aux autres activités.
Erreur	**Source du problème & Solution**
Au lancement de cette activité, si aucune ronde n'était disponible pour l'agent, l'application crachait.	La liste déroulante ne trouvait pas les données nécessaires à son initialisation et faisait cracher l'application. Maintenant, lorsqu'aucune ronde n'est disponible, la liste déroulante est désactivée et remplacée par un texte informant l'utilisateur qu'aucune ronde n'est disponible.

7 - Définition des paramètres partagés

Comme vu précédemment, lorsqu'une ronde est lancée, elle est enregistrée comme lancée par l'application et son nom est « transmis aux autres activités ». Cela signifie que l'état de la ronde est écrit dans un fichier XML libre d'accès par toute l'application. Cette fonctionnalité est gérée de façon native par le système Android.

7.1 - Fonctionnement des SharedPreferences

Lorsque je veux mettre à disposition de toutes les activités certaines données, il me suffit de les partager grâce à la fonctionnalité « SharedPreferences » d'Android comme ci-dessous :

ChoixRonde.java :

```
1  SharedPreferences prefs = getSharedPreferences("prefs", 0);
2  SharedPreferences.Editor editor = prefs.edit();
3  editor.putBoolean("RondeEnCours", etatRonde);
4  editor.commit();
```

Ligne 1 : Je récupère le fichier nommé « prefs.xml » en mode privé.
Ligne 2 : J'indique que je souhaite éditer le fichier.
Ligne 3 : J'indique que je veux écrire la variable « RondeEnCours » dans le fichier et la mettre à « etatRonde » qui est égal à **`true`** si une ronde est en cours et **`false`** si aucune ronde n'est en cours.
Ligne 4 : J'envoie les données que je souhaite écrire dans le fichier.

Par la suite, il m'est possible de récupérer la variable partagée de la manière suivante :

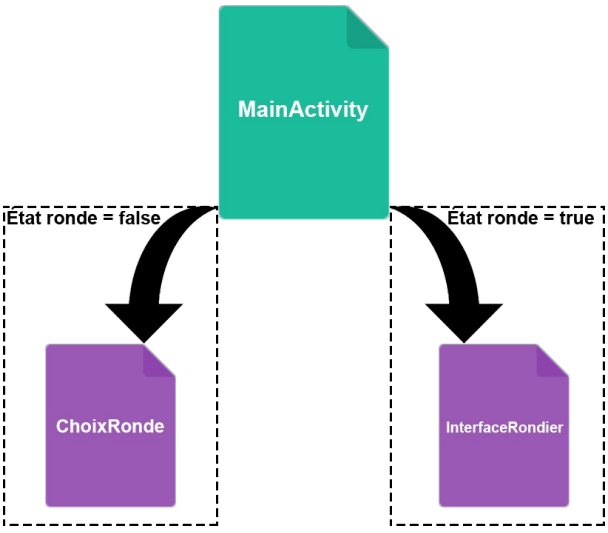
InterfaceRondier.java :

```
1  SharedPreferences prefs = getSharedPreferences("prefs", 0);
2  return prefs.getBoolean("RondeEnCours", false);
```

Ligne 2 : Récupère la variable « RondeEnCours » et retourne sa valeur. Si aucune variable nommé « RondeEnCours » n'est présente dans le fichier « prefs.xml », la valeur est **`false`** retournée.

7.2 - Usages des SharedPreferences

Dans une application Android, il est impossible de réagir à la détection d'un Tag MiFare dans deux activités différentes.
Il m'a donc fallu créer un système permettant d'utiliser l'activité principale (« MainActivity ») de façon à ce qu'elle appelle la partie voulue de code voulue en fonction de la situation.



MainActivity.java :

```
1  if(obtenirEtatRonde())
2  {
3      //Mise en avant de l'activité Rondier
4  }
5  else
6  {
7      //Identification de l'agent
8  }
```

Ici, la fonction obtenirEtatRonde correspond au code précédemment expliqué.

De cette façon, il m'est possible d'utiliser l'activité « MainActivity » comme l'activité scanner de l'application.

Si une ronde est en cours, le point de contrôle scanné est manipulé par l'activité Rondier décrite plus tard.

7.3 - Test unitaire

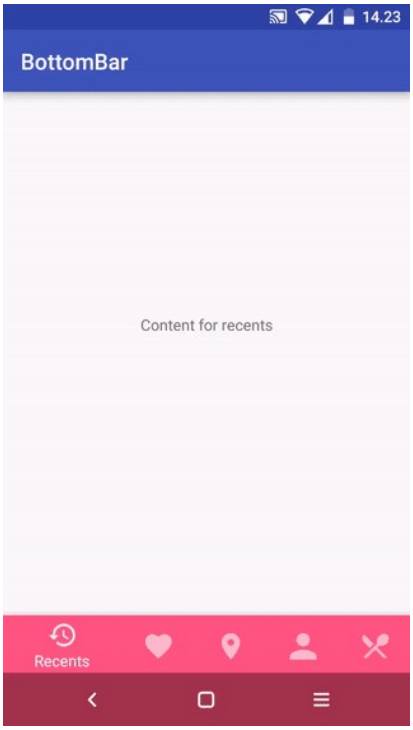
Conditions initiales	
L'application doit être lancée. Un TAG MiFare doit être enregistré dans la base de données.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement des préférences partagées.	
Opération	**Résultats attendus**
S'identifier et lancer une ronde	Il est inscrit dans les préférences qu'une ronde est en cours.
Quitter et fermer l'application puis la relancer. Scanner un TAG.	Le passage du pointeau est enregistré.
Erreur	**Source du problème & Solution**
L'application crash.	L'interface Rondier est lancée mais aucun agent ni aucune ronde n'est chargé causant le crash de l'application. Afin de résoudre ce problème, j'ai ajouté dans la méthode onDestroy (voir Annexe), j'ajoute une méthode permettant de mettre la ronde en cours à false.

8 - Installations des APIs

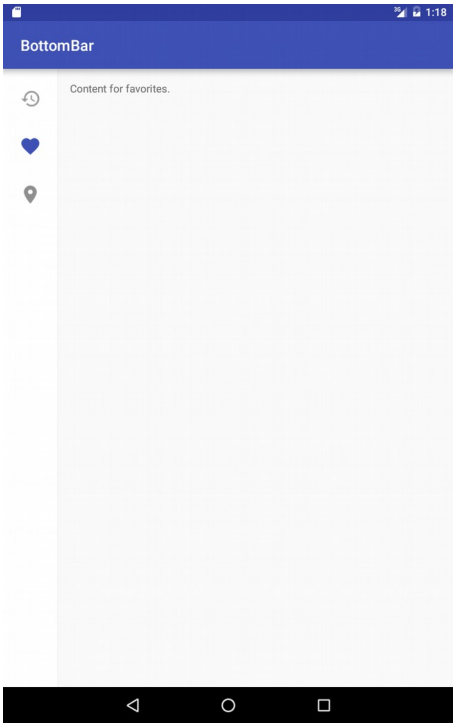
8.1 - API concernant l'interface utilisateur

Comme dit précédemment, lors de la présentation du projet, il m'a fallu utiliser deux APIs pour faciliter la prise en main ainsi que la compatibilité de l'application.

La première API que j'ai dû installer, bottombar de roughike, me permet de créer une partie de l'interface Rondier permettant d'effectuer une ronde. Cette API est disponible sur github à l'adresse https://github.com/roughike/BottomBar et est sous licence Apache 2.0.



La bottombar permet de naviguer entre différents onglets grâce à un simple clique.



Il est possible de mettre la bottombar comme une barre de navigation sur le côté de l'application pour un déploiement sur tablette par exemple.

Pour ajouter l'API à l'application, il faut commencer par télécharger et copier dans un dossier pour les APIs dans le projet de l'application, ici « library », les fichiers présents sur le github.
Ensuite, dans le fichier « settings.graddle », ajouter :

```
include ':app', ':library:bottom-bar'
```

Dans le fichier « build.gradle » du module app :

```
Dependencies {
    compile 'com.roughike:bottom-bar:2.2.0'
    compile project(':library:bottom-bar')
}
```

Il est nécessaire d'ajouter les dépendances comme décrit ci-dessus.
Il suffit ensuite de relancer Android Studio pour recharger tous les fichiers du projet et l'importation est terminée.

8.2 - API concernant la caméra

Le but de l'application développée étant d'être installable sur un grand notre de smartphone (de Android 4.1 aux dernières versions) il m'a semblé judicieux d'utiliser l'API cameraview-master de Google permettant de créer une camera facile d'usage pour l'utilisateur et compatible avec la plupart des smartphones disponibles sur le marché. Cette API est disponible sur github à l'adresse https://github.com/google/cameraview et est aussi sous licence Apache 2.0.

Cette API permet de régler la position du flash ou encore la capture sur la caméra arrière ou la caméra frontale.

La première étape de l'installation de cette API est de télécharger et de copier dans un dossier pour les APIs dans le projet de l'application, ici « library », les fichiers présents sur le github.

Les fichiers de dépendances sont ensuite à modifier, pour « settings.graddle » :

```
include ':app', ':library:cameraview-master'
```

Ensuite, pour le « build.gradle » du module app :

```
Dependencies {
    compile project(':library:camereaview-master')
}
```

Après ces modifications effectuées, vous devez relancer Android Studio et l'installation est terminée.
Attention : pour utiliser cette API, vous devez obligatoirement compiler votre projet avec le SDK 25 même si vous ne l'installez par sur un smartphone sous le SDK 25.

9 - Création dynamique de l'interface Rondier

9.1 - Conception détaillée

Lorsque l'interface Rondier est chargée, elle n'est, au niveau du code, que composée de trois volets qui n'ont aucune connexion.

activity_liste_pointeaux.xml :

```
1  <ViewFlipper
2      android:id="@+id/vfDisposition">
3      <ScrollView
4          android:id="@+id/svMainCourante">
5          <LinearLayout
6              android:id="@+id/layoutMainCourante">
7          </LinearLayout>
8      </ScrollView>
9      <ScrollView
10         android:id="@+id/svListePointeau">
11         <LinearLayout
12             android:id="@+id/layoutListePointeaux">
13         </LinearLayout>
14     </ScrollView>
15     <ScrollView
16         android:id="@+id/svPhoto">
17         <LinearLayout
18             android:id="@+id/layoutPhotos">
19         </LinearLayout>
20     </ScrollView>
21 </ViewFlipper>
```

Vous pouvez ici voir qu'aux lignes 3, 9 et 15, je crée des « ScrollViews » qui permettent d'avoir une interface que nous pouvons faire défiler s'il y a trop d'éléments. L'élément « ViewFlipper » permet de définir que chaque liste définie par la suite est un nouveau volet.

Ensuite, dans la déclaration de l'interface, je définis la bottom bar :

activity_liste_pointeaux.xml :

```
1  <com.roughike.bottombar.BottomBar
2      android:id="@+id/bottombar">
```

Bien sûr, la déclaration ci-dessus expliquée n'est pas celle utilisée dans le code source, le code a été étoffé afin de ne pas encombrer le document.

Après ces déclarations, je dois connecter tous ces éléments via le code de l'activité Rondier.
Pour se faire, il m'a fallu me familiariser avec l'API bottombar. Développer quelques applications uniquement pour tester ses différentes fonctionnalités m'a beaucoup aidé. Je ne détaillerai pas dans ce document les applications qui ont pour unique but de m'aider à m'habituer à l'API.

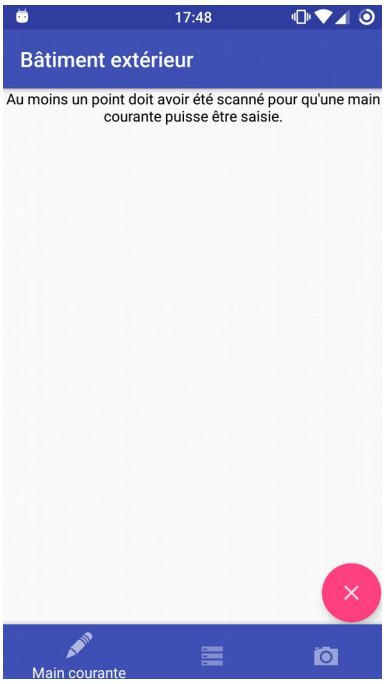
InterfaceRondier.java :

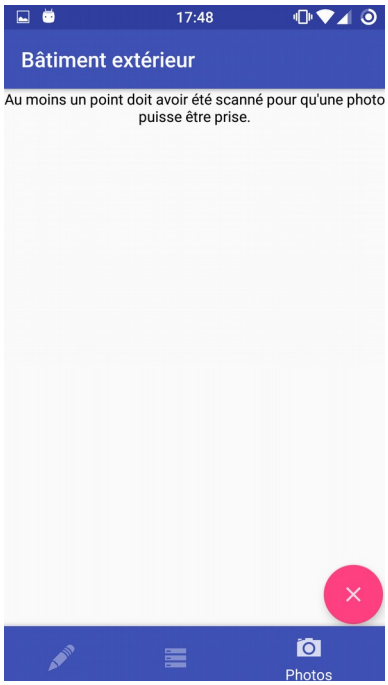
```
1  bottomBar = (BottomBar) findViewById(R.id.bottombar);
2  bottomBar.setOnTabSelectListener(new OnTabSelectListener() {
3      @Override
4      public void onTabSelected(@IdRes int tabId) {
5          if(tabId == R.id.tab_mainsCourantes)
6          {
7              flipperPrincipal.setDisplayedChild(0);
8          }
9          if(tabId == R.id.tab_listePointeaux)
10         {
11             flipperPrincipal.setDisplayedChild(1);
12         }
13         if(tabId == R.id.tab_photos)
14         {
15             flipperPrincipal.setDisplayedChild(2);
16         }
17     }
18 });
19 bottomBar.setDefaultTab(R.id.tab_listePointeaux);
```

Ligne 1 : Récupération de l'élément « bottombar » à partir du fichier XML de l'interface.
Ligne 2 : Définition de l'action à faire à la sélection d'un volet.
Ligne 8 : Si le volet sélectionné est celui correspondant à celui des mains courantes, le « ViewFlipper » charge les données associées .
Ligne 19 : Le volet affiché par défaut est celui listant les pointeaux de la ronde.

Lorsque les onglets sont chargés, les modifications précédemment faites sont conservées.

Au démarrage de l'interface Rondier, les interfaces Photos et Mains Courantes ne sont pas utilisables par l'utilisateur comme aucun pointeau n'a encore été scanné.



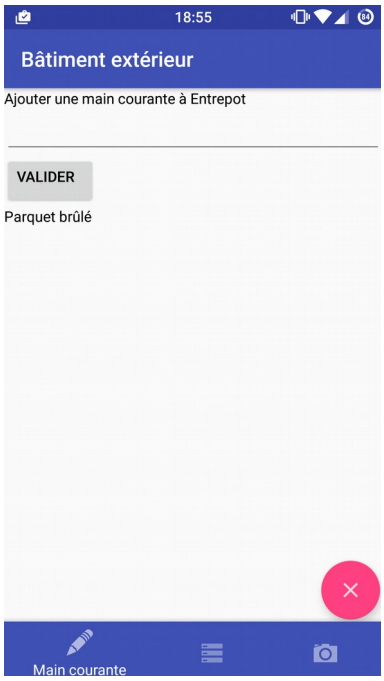


Lorsqu'un pointeau est scanné pour la première fois, les interfaces Mains Courantes et Photos sont mises à jour automatiquement de façon à charger les mains courantes correspondantes ainsi que les photos.
Cette mise à jour de l'interface permet aussi à l'utilisateur d'ajouter ses propres mains courantes et photos.

InterfaceRondier.java :

```
1  if(pointeauScanne != null && findViewById(R.id.boutonAjouterPhoto) == null)
2  {
3      supprimerTxtViewAucunPointeauScanne();
4      creerInterfaceAjouterMainCourante();
5      creerBoutonAjouterPhoto();
6  }
```

Dans cette partie du code, si le pointeau scanné est reconnu et aucun bouton d'ajout de photo n'est détecté, cela signifie qu'il s'agit du premier pointeau scanné. Dans ce cas, je supprime les textes vus dans les interfaces précédents et je crée les interfaces Main Courante et Photo comme ci-dessous.



Lorsqu'il est possible d'ajouter une main courante, il est dit pour quel pointeau il est possible de l'ajouter et la liste des mains courantes déjà présentes pour ce dernier.

Le bouton d'ajout de photo est nommé en fonction du dernier pointeau scanné et les photos déjà prises pour ce pointeau sont affichées.

9.2 - Test unitaire

Conditions initiales	
Un agent doit être identifié.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de la génération de l'interface Rondier.	
Opération	**Résultats attendus**
Choisir une ronde et la valider.	L'interface Rondier est chargée.
Cliquer sur l'icône ajouter une main courante et ajouter une main courante.	La main courante ajoutée est affichée.
Erreur	**Source du problème & Solution**
L'application crash.	Au moment de l'enregistrement de la main courante, aucun pointeau n'a encore été scanné causant une erreur à l'insertion dans la base de données. Pour contrer ce problème, il m'a suffi de modifier les interfaces comme vues ci-dessus.

10 - Génération de la liste des pointeaux

10.1 - Conception détaillée

Lorsqu'une ronde est lancée, l'application doit lister les pointeaux associés à cette ronde pour indiquer à l'agent de sécurité vers quel pointeau il doit se diriger.
Pour se faire, je fais une requête dans la table « AssociationPointeauxRondes » retournant tous les IDs des pointeaux associés.

```
"SELECT * FROM " + TABLE_ASSOCIATION_POINTEAUX_RONDES + " WHERE " + CLE_APR_RONDE_ID + "
=\"" + idRonde + "\" ORDER BY " + CLE_APR_ORDRE_POINTEAU
```

La requête ci-dessus retourne donc la liste des pointeaux associés à la ronde passés en paramètre en les classant par ordre de passage afin d'avoir le bon ordre créé sur l'application de supervision.
Cette requête est utilisée lors de la création d'une ronde. En effet, le constructeur prend en paramètre une liste de pointeaux. La liste de pointeaux est en elle-même créée grâce aux IDs récupérés précédemment.

Ronde.java :

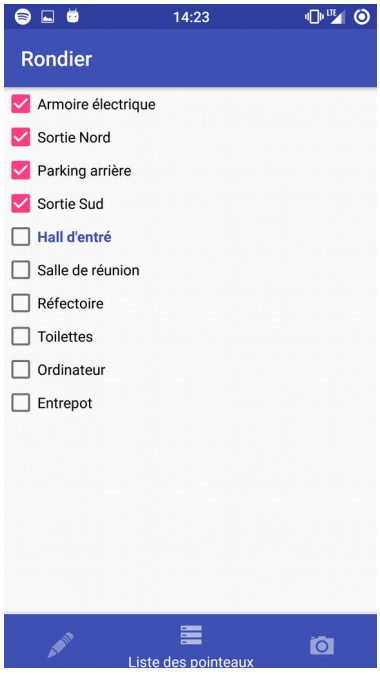
```
1  public Ronde(int id, String nom, List<Pointeau> pointeauList, int numeroRonde)
2  {
3      this.id = id;
4      this.nom = nom;
5      this.pointeauList = pointeauList;
6      this.numeroRonde = numeroRonde;
7  }
```

Ensuite, il me suffit de créer une checkbox pour chaque pointeau présent dans la ronde :

InterfaceRondier.java :

```
1  for(int i = 0; i < ronde.obtenirListePointeaux().size(); i++)
2  {
3      creerCheckboxPointeau(i);
4  }
```

Ici, la fonction « creerCheckboxPointeau » se contente de créer la checkbox (donc une opération purement graphique, se référer à l'annexe pour plus d'informations) pour le pointeau correspondant à la valeur de « i » aillant une valeur entre 0 et la taille maximum de la liste de pointeaux associés à la ronde.
L'interface créée est la suivante :



10.2 - Test unitaire

Conditions initiales	
Un agent de sécurité doit être identifié.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de la génération de la liste des pointeaux.	
Opération	**Résultats attendus**
Lancer une ronde.	La liste des pointeaux est générée.
Erreur	**Source du problème & Solution**
Les pointeaux listés ne sont pas dans l'ordre de passage.	Lors de la réception des pointeaux, je ne classais pas les pointeaux par ordre de passage dans la requête SQL vue précédemment. J'ai donc ajouté le « ORDER BY ordrePassage » à la requête SQL.

11 - Réalisation de la vérification des pointeaux scannés

11.1 - Conception détaillée

Lorsqu'un pointeau est scanné par l'agent de sécurité, il doit être indiqué comme passé dans l'interface. Pour se faire, il m'a fallu programmer une boucle passant toute la liste de pointeaux.

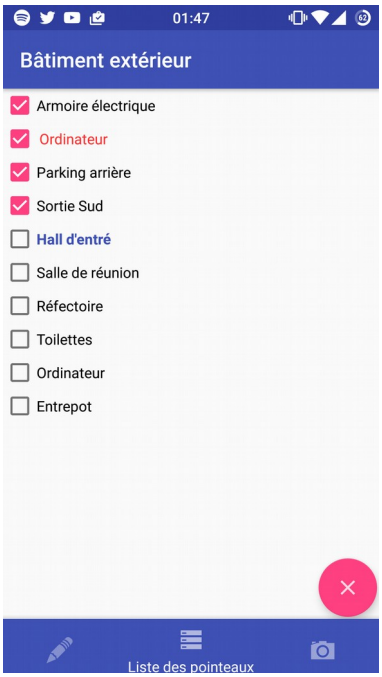
InterfaceRondier.java :

```
1  while(loop < layoutListePointeaux.getChildCount() && !pointeauVerifie)
2  {
3      if(layoutListePointeaux.getChildAt(index).getTag().toString().equals(
4      pointeauScanne.obtenirIdTag()))
5      {
6          CheckBox chckBox = (CheckBox) layoutListePointeaux.getChildAt(index);

7          if(!chckBox.isChecked())
8          {
9              chckBox.setChecked(true);
10             pointeauVerifie = true;
11         }
12     }
13     loop++;
   }
```

Ligne 1 : Tant que nous ne dépassons pas la taille de la liste de pointeaux et tant qu'aucun pointeau n'a été vérifié, la boucle continue.
Ligne 3 : Si le nom de la checkbox à l'index actuel est égal au nom du pointeau scanné, nous entrons dans la condition et récupérons la checkbox concernée comme telle (ligne 6).
Ligne 7-9-10 : Si la checkbox n'est pas déjà cochée, elle est cochée puis le pointeau est marqué comme vérifié ce qui fait que la vérification des pointeaux se fait à la première validation, à l'index actuel.

À la fin de cette vérification, si aucun pointeau n'est trouvé dans la ronde, un pointeau est ajouté à la liste en rouge pour indiquer qu'il ne fait pas partie de la ronde.



Après qu'un pointeau ait été scanné, que le pointeau soit présent dans la ronde ou non, l'agent de sécurité a la possibilité d'ajouter des mains courantes et/ou des photos.

Peu importe que le pointeau soit présent dans la ronde ou pas, le scannage de ce dernier est toujours enregistré dans la table « HistoriquePointeau » de la base de données.

Le code ci-dessous me permet son enregistrement :

GestionBDD.java :

```
1  ContentValues values = new ContentValues();
2  values.put(CLE_HP_AGENT_ID, idAgent);
3  values.put(CLE_HP_RONDE_ID, idRonde);
4  values.put(CLE_HP_POINTEAU_ID, idPointeau);
5  values.put(CLE_HP_DATE, tempsPresent.getString(0));
6  values.put(CLE_HP_ORDRE_POINTEAU, ordrePointeau);
7  values.put(CLE_HP_NUMERO_RONDE, numeroRonde);

8  db.insert(TABLE_HISTORIQUE_POINTEAU, null, values);
```

Ligne 2 : Définition de l'ID de l'agent ayant scanné le pointeau.
Ligne 3 : Définition de l'ID de la ronde dans laquelle le pointeau a été scanné.
Ligne 4 : Définition de l'ID du pointeau scanné.
Ligne 5 : Définition de la date à laquelle le pointeau a été scanné.
Ligne 6 : Définition de l'ordre dans lequel le pointeau a été scanné.
Ligne 7 : Définition du numéro de la ronde dans laquelle le pointeau a été scanné (permets de différencier le passage de deux rondes identiques).

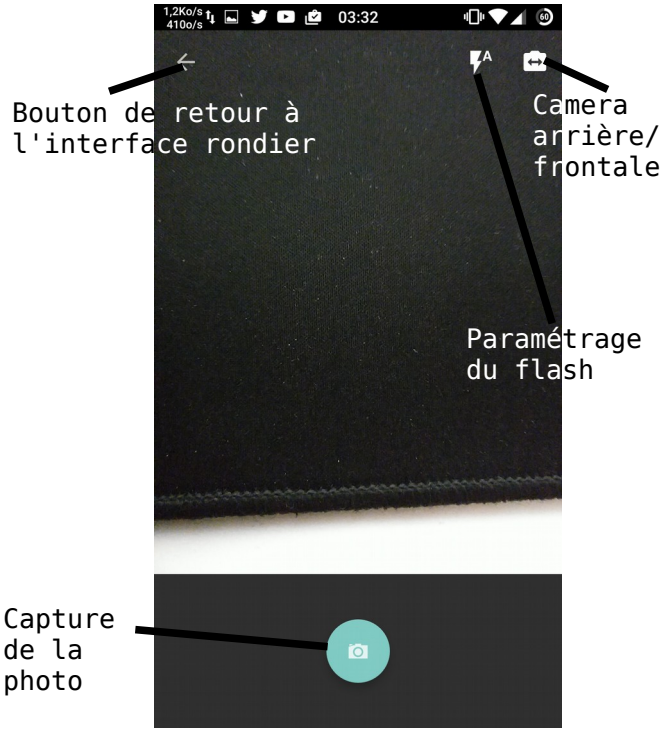
11.2 - Test unitaire

Conditions initiales	
Une ronde doit être lancée.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de la vérification des pointeaux scannés.	
Opération	**Résultats attendus**
Scanner un pointeau présent dans la ronde.	Le pointeau scanné est coché sur l'interface.
Scanner un pointeau qui n'est pas présent dans la ronde.	Le pointeau est ajouté à la liste des pointeaux en rouge.
Erreur	**Source du problème & Solution**
L'application crash au moment où un pointeau non présent dans la ronde est scanné.	Lorsqu'un pointeau non présent dans la ronde est scanné, le pointeau n'était pas trouvé dans la liste ce qui causait un OutOfBounds Exception. J'ai dû changer le corps de l'activité pour prendre en compte si un pointeau n'est pas trouvé dans la liste.

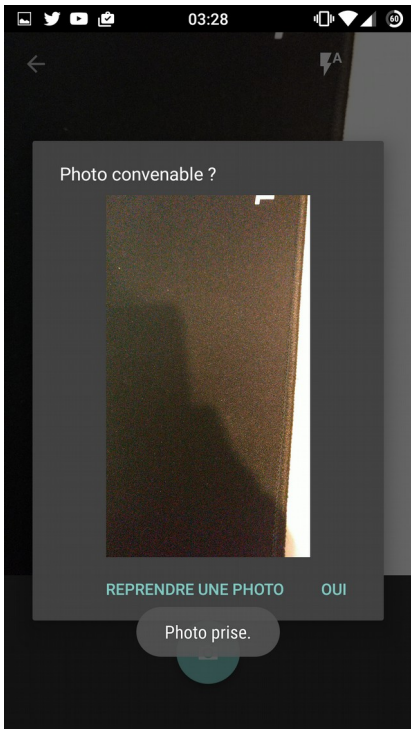
12 - Réalisation de la capture de photo

12.1 - Conception détaillée

Lorsqu'un pointeau est scanné, qu'il soit présent ou non dans la ronde, il est possible d'ajouter une photo grâce au bouton prévu à cet effet dans l'interface de listage des photos.
Ce bouton permet de lancer l'activité « camera » construite à partir de l'API cameraview-master.
L'usage de cette API facilite le réglage de la camera comme par exemple l'usage du flash (automatique, activé ou désactivé) et l'orientation de la camera (frontale ou arrière).



L'interface reste basique mais comble tous les besoins que peut avoir l'utilisateur pour avoir une photo de qualité.



Une fois la photo capturée, il est possible pour l'utilisateur de reprendre une photo si ce dernier juge que celle précédemment prise n'est pas correcte.

Lorsque l'agent de sécurité décide d'enregistrer la photo qu'il vient de capturer, elle est enregistrée dans le dossier « /sdcard/data/Android/com.project.rondierprojet/files/Documents » du smartphone. Cette photo est enregistrée au format jpg et la résolution maximale est de 1920x1080 pixels. Si un smartphone possédant une camera pouvant prendre des photos de 3996x2160 pixels, la photo sera bridée à 1920x1080 pixels. Si un smartphone possède une caméra pouvant prendre des photos en 1280x720, la photo sera enregistrée avec une résolution de 1280x720.

Le choix du format jpg pour l'enregistrement des photos est justifié par la restriction de stockage que peut avoir un smartphone. Même si de nos jours, les derniers smartphones haut de gamme n'auraient aucun problème à enregistrer jusqu'à 25000 photos en 4k seulement l'application est faite de façon à pouvoir être déployée sur le plus grand nombre de smartphone, y compris les smartphones « lowcost ».

Lors de l'enregistrement de la photo, le nom est défini de façon à ce que nous puissions savoir à quoi correspond la photo grâce au nom.
Le format adopté est « nomRonde|nomPointeau|JJ.MM.AAAA|HH.mm.ss.jpg.

Une fois la photo enregistrée dans les fichiers, elle est enregistrée dans la base de données afin d'être facilement récupérable pour la visualisation des statistiques de fin de ronde sur le poste de supervision.

Pour se faire, j'utilise le code suivant :

GestionBDD.java :

```
1  ContentValues values = new ContentValues();
2  values.put(CLE_MAIN_COURANTE_TEXT, chemin);
3  values.put(CLE_MAIN_COURANTE_ID_HISTORIQUE_POINTEAU, resultat.getInt(0)+1);
4  values.put(CLE_MAIN_COURANTE_DATE, tempsPresent.getString(0));
5  values.put(CLE_MAIN_COURANTE_TYPE, "1");

6  db.insert(TABLE_MAIN_COURANTE, null, values);
```

Ligne 2 : Le chemin correspond au chemin jusqu'au fichier plus le nom de la photo.
Ligne 3 : L'ID de l'historique pointeau correspond à l'ID du dernier enregistrement dans la table d'historique.
Ligne 4 : La date correspond à la date et heure de l'enregistrement.
Ligne 5 : Le type de la main courante sert à savoir s'il s'agit d'une main courante (0) ou d'une photo (1).
Ligne 6 : Insertion dans la base de données.

12.2 - Test unitaire

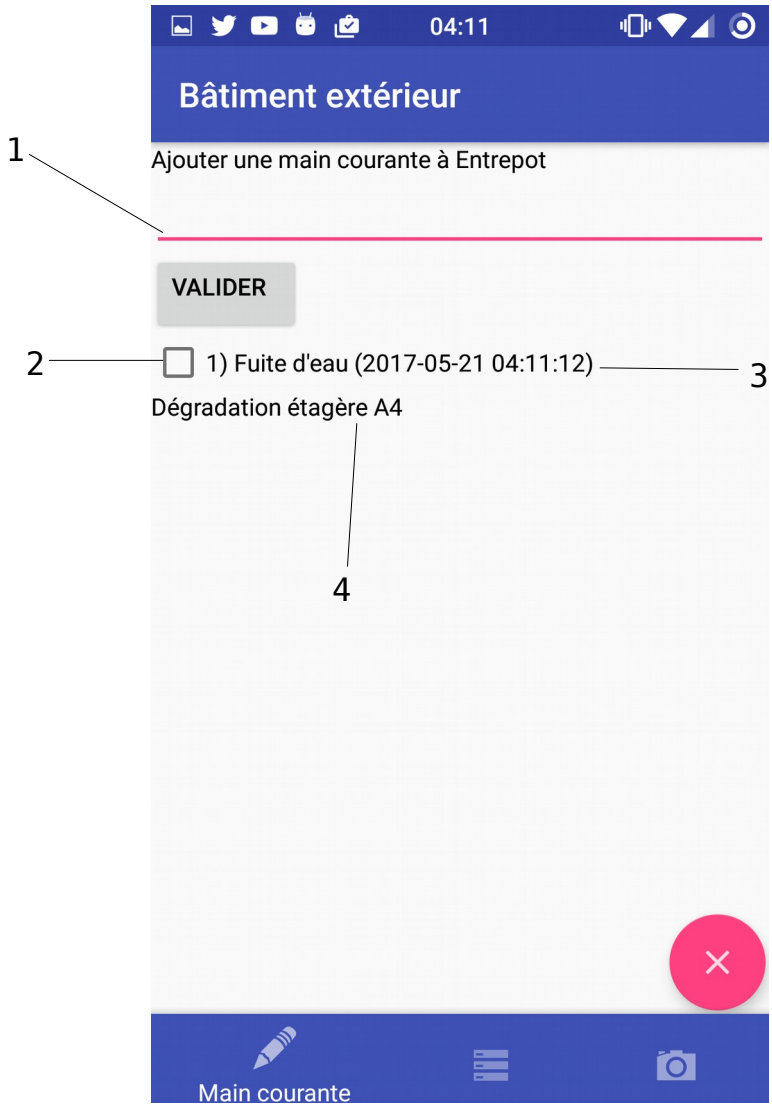
Conditions initiales	
Une ronde doit être lancée. Un pointeau doit avoir été scanné.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de capture de photo.	
Opération	**Résultats attendus**
Lancer l'activité camera en cliquant sur le bouton « Ajouter une photo »	L'interface camera se charge correctement.
Prendre une photo et l'enregistrer.	La photo est enregistrée dans les fichiers du téléphone et l'utilisateur en est informé.
Erreur	**Source du problème & Solution**
Il n'est pas possible de revenir vers l'interface rondier.	Aucun bouton de retour n'avait été prévu pour que l'utilisateur continue sa ronde sans quitter l'application. Pour régler ce problème, il m'a suffi d'ajouter le bouton retour comme présenté dans les interfaces précédentes.

13 - Rélisation de la saisie d'une main courante

13.1 - Conception détaillée

Si l'agent de sécurité pense qu'une information doit être remontée auprès de son supérieur, il lui est possible de déposer une main courante pour le dernier pointeau scanné.

Cela se fait grâce à l'interface précédemment vue dans la partie concernant la création des interfaces au lancement de l'activité rondier.



1 : Champ de saisie d'une main courante

2 : Bouton de validation de la main courante saisie (le champ 1 ne doit pas être vide)

3 : Main courante saisie lors d'un ancien passage demandant une confirmation

4 : Main courante saisie lors du passage actuel

Grâce au système de validation des mains courantes précédemment saisies, il est possible pour le responsable de la supervision de visionner l'évolution d'un problème sur le trajet d'une ronde.

Lorsqu'une main courante est saisie et ajoutée à l'interface, elle est enregistrée dans la table MainCourante de la base de données.
Cet enregistrement est fait avec le code suivant :

GestionBDD.java :

```
1  ContentValues values = new ContentValues();
2  values.put(CLE_MAIN_COURANTE_TEXT, enregistrement);
3  values.put(CLE_MAIN_COURANTE_ID_HISTORIQUE_POINTEAU, resultat.getInt(0)+1);
4  values.put(CLE_MAIN_COURANTE_DATE, tempsPresent.getString(0));
5  values.put(CLE_MAIN_COURANTE_TYPE, "0");

6  db.insert(TABLE_MAIN_COURANTE, null, values);
```

Ligne 2 : Le texte saisi pour la main courante.
Ligne 3 : L'ID du dernier historique correspondant au dernier pointeau scanné.
Ligne 4 : La date et l'heure à laquelle la main courante a été saisie.
Ligne 5 : Le type de la main courante, ici 0 pour indiquer qu'il s'agit d'une main courante textuelle.

13.2 - Test unitaire

Conditions initiales	
Une ronde doit être lancée. Un pointeau doit avoir été scanné.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de la saisie de main courante.	
Opération	**Résultats attendus**
Saisir une main courante et valider.	La main courante est ajoutée à l'interface et enregistrée dans la base de données.
Valider une main courante déjà saisie.	La main courante est réenregistrée dans la base de données.
Erreur	**Source du problème & Solution**
Aucune erreur décelée.	

14 - Prise en compte des cas particuliers

Lors de la création d'une ronde ou d'un agent, il est possible que des erreurs soient faites comme ne pas associer de ronde à un agent ou ne pas associer de pointeau à une ronde. Auparavant, ces oublies causaient de grands crashs de l'application et corrompaient la base de données.
Afin de résoudre ce problème, il m'a fallu créer des interfaces et des morceaux de code qui permettent de prendre en compte ces exceptions.

Concernant un agent sans aucune ronde :

ChoisirRonde.java :

```
1  if(listeRondesBadge == null)
2  {
3      txtView.setText("Aucune ronde prévue.");
4  }
```

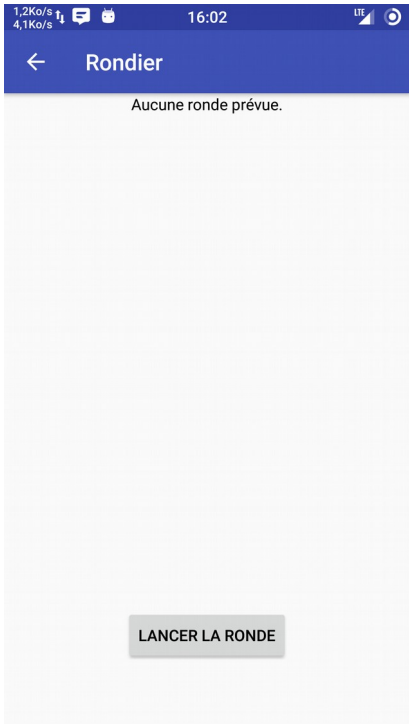
De cette façon, l'application ne tente pas de créer des boutons pour lancer des rondes dans le vide.

La partie concernant l'absence de pointeau dans une ronde est sensiblement identique :

InterfaceRondier.java :

```
1  if(ronde.obtenirListePointeaux() == null)
2  {
3      txtView.setText("Aucun point dans cette ronde.");
4  }
```

Le principe est ici identique et évite aussi le crash de l'application.
Lorsque ce genre de situation advient, l'agent de sécurité peut quitter la ronde en cours s'il en a lancé une ou simplement relancer l'application.



Lorsqu'aucune ronde n'est disponible, il est possible de retourner en arrière.

Lorsqu'aucun pointeau n'a été associé à une ronde, il est possible de quitter la ronde.

15 - Réalisation de la fin de ronde

15.1 - Conception détaillée

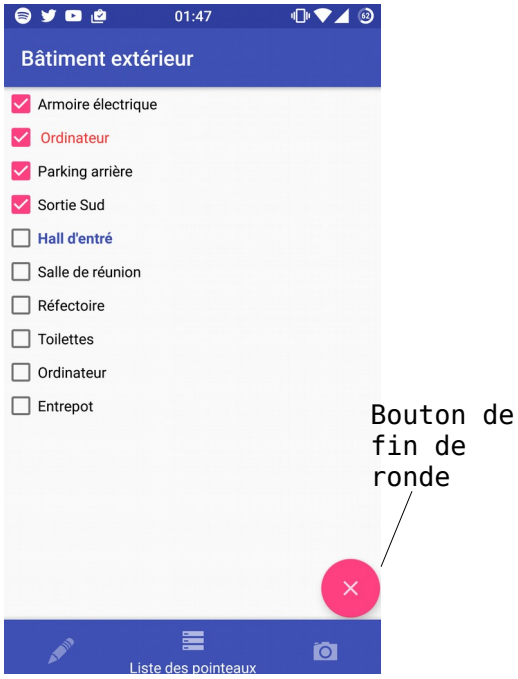
Une fois tous les points scannés ou à la demande de l'agent, il est nécessaire de pouvoir quitter la ronde, de la marquer comme terminé.
La demande de fin de ronde peut être automatique :

InterfaceRondier.java :

```
1  if(loop == layoutListePointeaux.getChildCount()-1)
2  {
3      new AlertDialog.Builder(InterfaceRondier.this)
4          .setMessage(R.string.dernier_pointeau)
5          .setPositiveButton(R.string.terminer, new DialogInterface.OnClickListener() {
6              @Override
7              public void onClick(DialogInterface dialogInterface, int i) {
8                  redemarrerApplication();
9              }
10         })
11        .setNegativeButton(R.string.continuer, new DialogInterface.OnClickListener(){})
12        .show();
13 }
```

Ligne 1 : Si le pointeau scanné est le dernier de la liste.
Ligne 3 : Création du message de confirmation.
Ligne 4 : Ajout du texte « Voulez-vous vraiment terminer la ronde en cours. »
Ligne 5 : Ajout du bouton de validation qui redémarre l'application (fonction détaillée plus en détail en dessous).
Ligne 11 : Ajout du bouton d'annulation.

Il est aussi possible pour l'agent de sécurité de quitter la ronde manuellement grâce au bouton prévu à cet effet.



Le bouton de fin de ronde est toujours visible.

Une boîte de dialogue est ensuite générée pour confirmer la fin de la ronde.

Si l'agent de sécurité valide que la ronde est terminée, les informations sont enregistrées dans l'application et cette dernière est redémarrée :

InterfaceRondier.java :

```
1  private void redemarrerApplication()
2  {
3      SharedPreferences prefs = getSharedPreferences("prefs", 0);
4      SharedPreferences.Editor editor = prefs.edit();
5      editor.clear();
6      editor.commit();

7      startActivity(mainActivity);
8      finish();
9  }
```

Ligne 3 : Récupération du fichier « prefs » comportant les données transférées.
Ligne 4 : Lancement de l'édition du fichier.
Ligne 5 : Remise à zéro du fichier.
Ligne 6 : Insertion des données.
Ligne 7 : Redémarrage de l'activité « MainActivity ».
Ligne 8 : Fermeture de l'activité actuelle.

Le fait de remettre le fichier de préférences à zéro permet de supprimer toutes les données. Cela empêche une corruption de l'application par d'anciennes données au moment où elle est relancée après avoir terminé une ronde.

15.2 - Test unitaire

Conditions initiales	
Une ronde doit être lancée.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de la fin de ronde.	
Opération	**Résultats attendus**
Quitter la ronde grâce au bouton de sortie de ronde.	L'application est redémarrée.
Scanner un badge agent.	L'interface de choix de ronde est affichée.
Erreur	**Source du problème & Solution**
L'application crash.	Lorsque l'application était relancée, l'interface rondier était relancée, mais sans donnée pouvant la construire (pointeau, ronde, agent) ce qui causait une NullPointerException et faisait crasher l'application. Afin de régler ce problème, j'ai dû remettre à zéro le fichier de préférences.

16 - Réalisation de la synchronisation de la partie Android

16.1 - Conception détaillée

Afin de transférer les données entre le poste de supervision et le smartphone, il m'a fallu trouver un moyen de communiquer entre les deux parties. Pour se faire, j'ai dû utiliser un événement spécifique d'Android pour faire appel à une certaine partie du programme. J'ai donc créé la classe « Synchronisation » qui n'est accessible que via un appel ADB (détaillé dans la synchronisation du côté de l'application de supervision).

Pour définir le filtre d'appel, il a fallu créer l'activité Synchronisation avec des paramètres particuliers :

AndroidManifest.xml :

```
1  <activity
2      android:name=".Synchronisation"
3      <intent-filter>
4          <action android:name="android.intent.action.SYNC"/>
5      </intent-filter>
6  </activity>
```

Dans cette déclaration de l'activité, je déclare que l'activité peut être appelée grâce à l'action « SYNC » ligne 4.

Par la suite, une fois que l'activité est lancée, elle doit vérifier si l'action qui l'appelle est bien « SYNC » pour éviter toute confusion :

Synchronisation.java :

```
1  if(getIntent().getAction().equals(Intent.ACTION_SYNC))
2  {
3      handler.postDelayed(new Runnable() {
4          @Override
5          public void run() {
6              try
7              {
8                  dbExterne = SQLiteDatabase.openDatabase("ControleurDeRonde.db", null,
9                  SQLiteDatabase.OPEN_READWRITE);
10             } catch (SQLiteException e)
11             {
12                 //Erreur d'ouverture
13             }
14             if(dbExterne != null)
15             {
16                 GestionBDD db = new GestionBDD(Synchronisation.this);
17                 db.transfererMainCouranteHistoriquePointeau(dbExterne);
18                 db.transfererPointeauxAgentsRondes(dbExterne);
19             }
20             Log.d(getString(R.string.tag_log), "BDDSynchroFin");
21             finish();
21         }
23     }, 2000);
24 }
```

Ligne 1 : Test de l'action appelant l'activité, s'il s'agit bien de l'action « SYNC », le programme continu.
Ligne 3 : La création d'un handler permet de donner un délai à l'exécution de la méthode. L'attente permet de ne pas utiliser la base de données avant qu'elle ait été totalement transférée sur le smartphone.
Ligne 8 : Ouverture de la base de données venant de l'application de supervision du poste de supervision.
Ligne 14 à 19 : Si la base de données a été ouverte, l'application synchronise sa propre base de données avec la base de données transférée pour la synchronisation.
Ligne 20 : L'application Android envoie un signal dans les logs pour marquer la fin de la synchronisation des données.
Ligne 21 : L'activité « Synchronisation » se ferme automatiquement.

Afin de tester cette partie de l'application, j'ai dû développer un petit programme sur PC. Il s'avère que ce programme de test a été adopté comme solution finale de l'application de supervision.

17 - Conception de la synchronisation du poste de supervision

17.1 - Environnement de développement

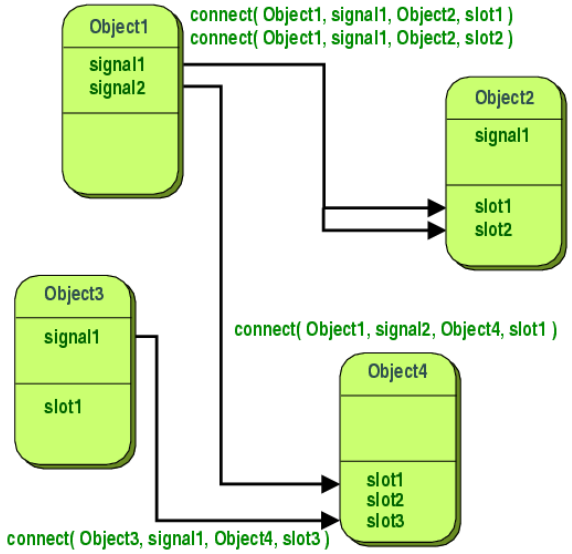
La partie de l'application de supervision concernant la synchronisation a été développée à l'aide du logiciel de développement Qt 5.7 et utilise dans son programme ADB pour le transfert des données. ADB est donc nécessaire lors du déploiement du projet final.

Afin de pouvoir détecter la connexion des smartphones, il m'a été nécessaire d'ajouter une librairie nommée QdeviceWatcher. Cette librairie est distribuée librement via github (https://github.com/wang-bin/qdevicewatcher) et sous licence GNU. Cette librairie n'a pas été modifiée par mes soins et vous pouvez trouver une copie de la licence GNU à l'adresse https://www.gnu.org/licenses/old-licenses/lgpl-2.1.fr.html.

Pour installer cette librairie, il a suffi d'ajouter les fichiers dans le projet Qt et d'inclure la librairie dans le programme de Qt :

```
#include "qdevicewatcher.h"
```

Cette librairie émet des signaux à chaque fois qu'un périphérique est connecté ou déconnecté :

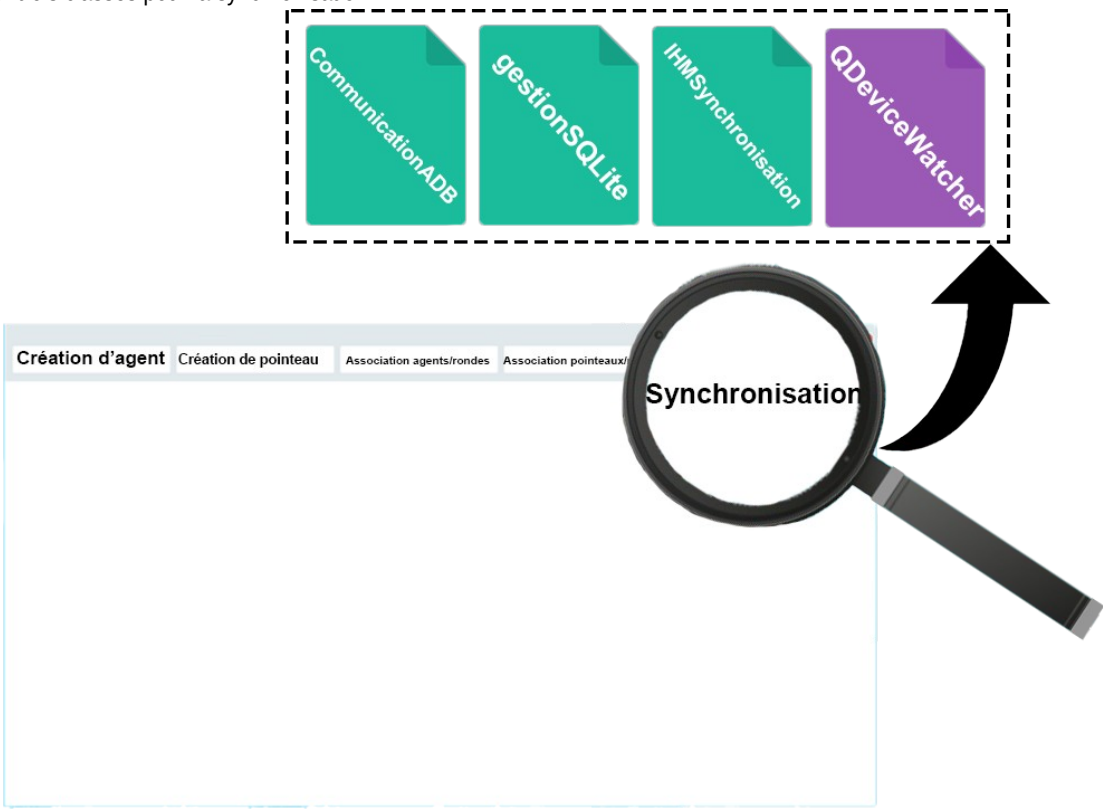


Il est ensuite possible de rattacher ces signaux à ce que l'on appelle des slots dans Qt. Cela nous donne donc la possibilité d'effectuer une action particulière lorsqu'un périphérique est connecté.

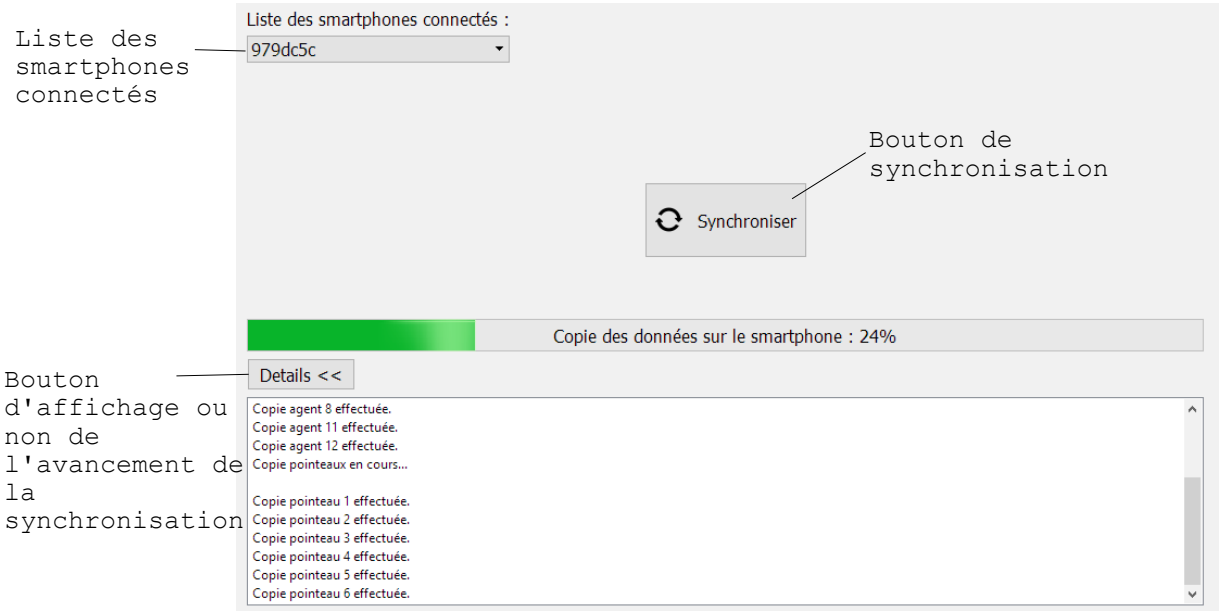
La synchronisation utilise aussi la base de données du poste de supervision pour envoyer les données vers le smartphone.

17.2 - Architecture de la partie synchronisation

J'ai dû développer trois classes pour la synchronisation :



CommunicationADB : Cette classe permet de dialoguer avec le smartphone grâce au logiciel ADB. Permet aussi de transférer des fichiers entre le smartphone et le poste de supervision.
GestionSQLite : Cette classe manipule l'intégralité des fichiers SQLite utilisés pour la synchronisation.
IHMSynchronisation : La classe interface permet à l'utilisateur de choisir quel smartphone il souhaite synchroniser et de voir l'avancement d'une synchronisation.



18 - Réalisation du listage des smartphones connectés

18.1 - Conception détaillée

Avant de pouvoir synchroniser les données entre un smartphone et le poste de supervision, il est nécessaire de pouvoir différencier les différents téléphones connectés.
Pour se faire, lorsqu'un smartphone est connecté, j'appelle grâce à un slot attaché au signal onDeviceAdded de QDeviceWatcher une méthode permettant de retourner la liste de tous les smartphones connectés via ADB.

```
C:\Users\Projet>adb devices
List of devices attached
979dc5c device
68ds684dsf6 device
s6azd84 device
```

La commande « adb devices » retourne comme montré précédemment la liste des smartphones connectés. Par la suite, j'utilise l'expression régulière suivante pour ne récupérer que les IDs.

```
\n(.*?)\t
```

Cette expression capture tous les caractères sur toutes les lignes (« **(.*?)** ») se trouvant après un retour ligne (« **\n** ») et avant une tabulation (« **\t** »). Je récupère donc une liste ressemblant à la liste suivante :

```
1 979dc5c
2 68ds684dsf6
3 s6azd84
```

Cette liste est ensuite ajoutée à la liste déroulante de l'interface.

18.2 - Test unitaire

Conditions initiales	
Vous devez disposer de deux smartphones sous Android et de deux câbles USB. L'un des deux smartphones doit être connecté au poste de supervision. L'application de supervision doit être lancée.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement du listage des smartphones connectés au poste de supervision.	
Opération	**Résultats attendus**
Sélectionner le smartphone dans la liste déroulante.	Le smartphone est sélectionné et le bouton de synchronisation est activé.
Connecter le second smartphone.	La liste déroulante des smartphones est actualisée.
Erreur	**Source du problème & Solution**
Le smartphone précédemment scanné est désélectionné.	Lorsqu'un smartphone était branché avant la sélection du premier, la liste était actualisée en supprimant son contenu puis en la recréant. Pour pallier le problème, j'ai développé une méthode pour mettre à jour la liste manuellement, sans la vider à chaque connexion/déconnexion.

19 - Réalisation du transfert des photos

19.1 - Conception détaillée

Le transfert des photos devait commencer par un listage des photos prises et stockées dans le répertoire « /sdcard/Android/data/com.project.rondierprojet/files/Pictures » du smartphone.
Pour se faire, j'utilise la commande suivante de ADB :

```
adb -s <id du smartphone> shell ls
/sdcard/Android/data/com.project.rondierprojet/files/Pictures/
```

Cette commande retourne une liste sous la forme :

```
Bâtiment A|Entrepôt|11.05.2017|17.34.59.jpg
Bâtiment A|Couloir|11.05.2017|17.47.32.jpg
Bâtiment A|Toilettes|11.05.2017|17.54.02.jpg
Bâtiment A|Bureau|11.05.2017|18.08.25.jpg
```

Il me suffit ensuite de lancer la commande suivante pour chaque photo retournée par la commande précédente :

```
adb -s <id du smartphone> pull
/sdcard/Android/data/com.project.rondierprojet/files/Pictures/<nom de la photo> Photos/
```

Grâce à cela, toutes les photos sont transférées sur le poste de supervision. Par la suite, dans le but de ne pas surcharger la mémoire du smartphone, je supprime toutes les photos prises par l'application côté Android :

```
adb -s <id du smartphone> shell rm
/sdcard/Android/data/com.project.rondierprojet/files/Pictures/*
```

Les photos sont ensuite accessibles sur le poste de supervision et le nommage est conservé dans le but de pouvoir les trier sans avoir besoin d'avoir un programme.

20 - Réalisation de l'envoi de données au smartphone

20.1 - Conception détaillée

Pour synchroniser les rondes, les pointeaux, les agents ainsi que les associations du poste de supervision vers l'application Android, il a fallu trouver un moyen d'envoyer des données entre les deux plates-formes. L'envoi de données sous forme de message n'étant pas possible, j'ai décidé d'adopter le transfert de fichiers SQLite entre le poste de supervision et le smartphone.

Je commence donc par créer une copie du fichier que je vais envoyer :

Gestionsqlite.cpp :

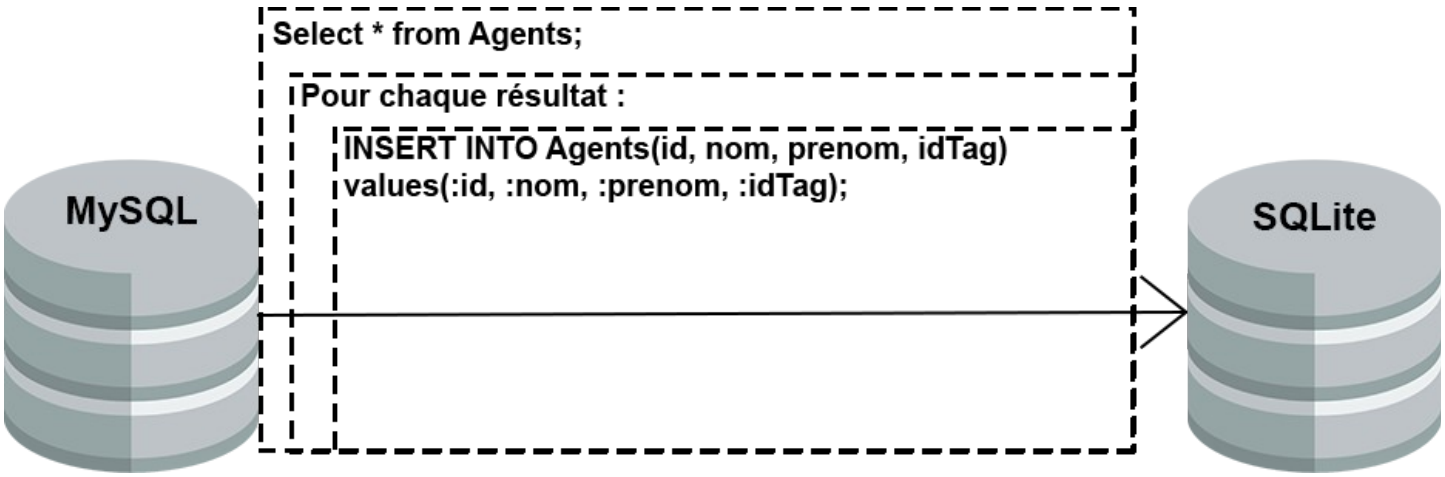
```
1  if(QFile::copy(cheminModeleSQLite, cheminSQLiteTransfere))
2  {
3      qDebug() << "Copie du fichier SQLite effectuée.";
4  }
```

Le fichier est alors rempli avec les données de la base de données MySQL du poste de supervision :

Gestionsqlite.cpp :

```
1  void GestionSQLite::synchroniserDonneesPC()
2  {
3      synchroniserAgents();
4      synchroniserPointeaux();
5      synchroniserRondes();
6      synchroniserAssociationAgentsRondes();
7      synchroniserAssociationPointeauxRondes();
8  }
```

Pour copier les données entre les différents systèmes de base de données, il m'a fallu utiliser un système particulier :



Le système est ensuite identique pour tous les éléments.

Après la copie des fichiers, il me suffit d'envoyer le fichier SQLite sur le smartphone pour qu'il soit traité par la classe « Synchronisation » présentée précédemment. L'envoi du fichier se fait grâce à la commande suivante :

```
adb -s <id du smartphone> push <chemin de l'application supervision>
/sdcard/Android/data/com.project.rondierprojet/files/Documents
```

20.2 - Test unitaire

<table>
<tr><th colspan="2">Conditions initiales</th></tr>
<tr><td colspan="2">Des informations dans la base de données MySQL doivent être enregistrées.
Un smartphone doit être connecté au poste de supervision.</td></tr>
<tr><th colspan="2">Procédure de test</th></tr>
<tr><td colspan="2">Cette procédure de test vérifie le fonctionnement du transfert des données du poste de supervision vers le smartphone.</td></tr>
<tr><th>Opération</th><th>Résultats attendus</th></tr>
<tr><td>Sélectionner le smartphone sur lequel synchroniser les données.</td><td>Le smartphone est sélectionné et le bouton de synchronisation est activé.</td></tr>
<tr><td>Synchroniser les données.</td><td>Les données sont présentes sur le smartphone.</td></tr>
<tr><th>Erreur</th><th>Source du problème & Solution</th></tr>
<tr><td>Le smartphone ne reçoit aucune donnée.</td><td>À la copie des données dans le fichier SQLite, les query utilisées pour faire les injections SQL communiquent entre elles parce qu'elles ne sont pas définies correctement.
À l'instanciation des queries, il faut utiliser le constructeur comportant une chaîne de caractères qui permet de différencier deux queries.
Nous passons donc de :
<code>requeteMySQL = new QsqlQuery(bddMySQL);
requeteSQLite = new QSqlQuery(bddSQLite);</code>
à
<code>requeteMySQL = new QSqlQuery("requeteMySQL", bddMySQL);
requeteSQLite = new QSqlQuery("requeteSQLite", bddSQLite);</code></td></tr>
</table>

21 - Réalisation de la récupération des données du smartphone

21.1 - Conception détaillée

Une fois le fichier SQLite envoyé sur le smartphone, il faut commencer par lancer la classe « Synchronisation » de l'application Android pour qu'elle traite les informations envoyées.
Pour se faire, j'ai dû utiliser la commande ADB suivante :

```
adb -s <id du smartphone> shell am start -n com.project.rondierprojet/.Synchronisation
-a android.action.intent.SYNC
```

Cette commande lance donc l'activité « Synchronisation » avec comme action de démarrage l'action « action.SYNC » ce qui permet de lancer le programme du côté Android correctement. Comme vu précédemment, une fois que l'application a terminé la synchronisation de données, elle envoie un message dans les logs. Pour récupérer ce message, j'utilise la même méthode que pour la partie du Scannage de tag.

Une fois que le message est donc reçu, le fichier est récupéré sur le poste de supervision grâce à la commande :

```
adb -s <id du smartphone> pull
/sdcard/Android/data/com.project.projetrondier/files/Documents/ControleurDeRonde.db
<chemin de l'application supervision>
```

Il suffit ensuite de recopier les données du fichier ainsi récupéré vers la base de données MySQL en utilisant le même système que défini plus tôt.

21.2 - Test unitaire

Conditions initiales	
Le smartphone doit avoir collecté des informations.	
Procédure de test	
Cette procédure de test vérifie le fonctionnement de la récupération des données de l'application Android.	
Opération	**Résultats attendus**
Lancer une synchronisation sur un smartphone préalablement sélectionné.	La synchronisation se lance, l'activité correspondant à la classe « Synchronisation » de l'application Android est affichée un bref instant sur le smartphone.
La synchronisation se termine.	Les données ont bien été copiées sur le poste de supervision.
Erreur	**Source du problème & Solution**
Aucune donnée n'est ajoutée à la base de données de l'application de supervision.	Lors de la copie du fichier SQLite sur le poste de supervision, j'écrasais le fichier précédemment utilisé et faisais des requêtes sans redéfinir mes queries. Pour résoudre le problème, il m'a suffi de ne pas écraser le fichier et d'ouvrir le fichier ainsi copié avec des variables différentes dans le code.

22 - Bilan du développement effectué

Au cours du développement de mon application Android, il m'a été possible de développer l'intégralité des fonctionnalités demandées. La seule chose qu'il me resterait à ajouter serait un bouton « À propos de » au lancement de l'application qui permettrait d'indiquer à l'utilisateur que des APIs qui ne m'appartiennent pas ont été utilisées pour le développement de ce projet.

En ce qui concerne la partie sur le poste de supervision de synchronisation, j'ai pu développer la plus grande partie de l'application, le programme fonctionne en théorie, il me manque néanmoins un certain nombre de fixes à faire surtout concernant le lancement d'une seconde synchronisation après avoir déjà synchronisé un smartphone.
Il manque en plus de cela la mise en commun avec l'application squelette de Gabriel Lemée pour avoir un système pratique pour le responsable de sécurité.

Une amélioration que j'ai pu ajouter à la synchronisation a été la barre de progrès et les informations concernant l'avancement de la synchronisation des informations. Ces informations sont, il me semble, nécessaires pour que l'utilisateur sache ce qu'il se déroule sur le poste de supervision et pour qu'il ne pense pas que l'application a planté.

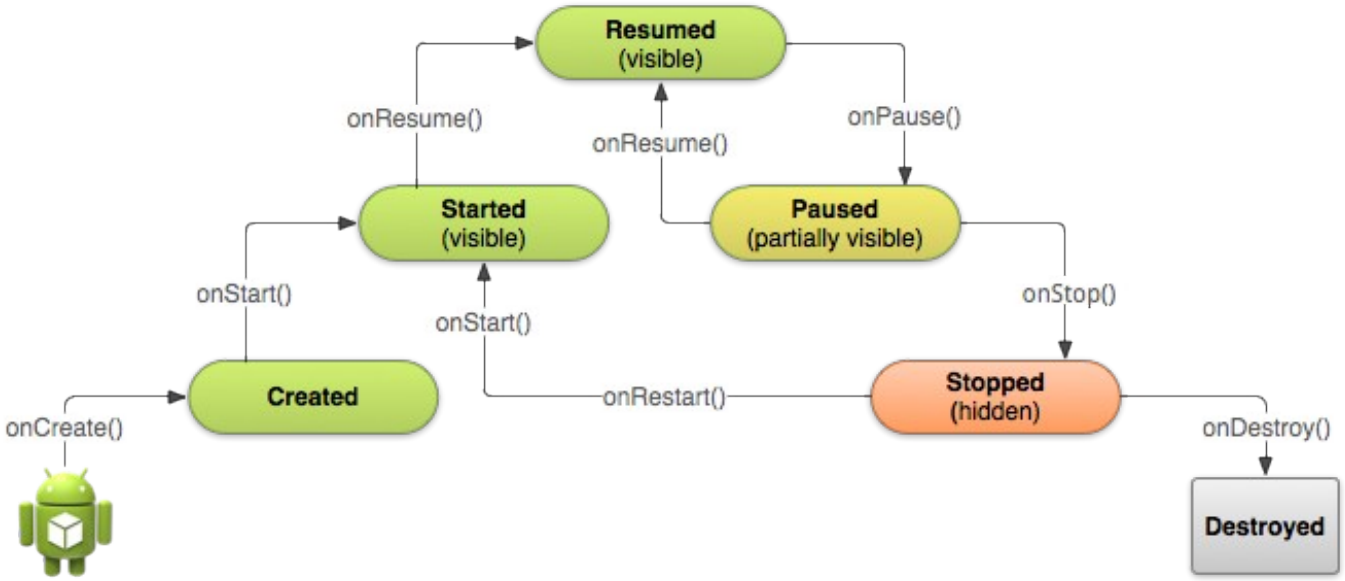
Dans l'ensemble, j'ai trouvé ce projet complet tant au niveau des supports de développement qu'au niveau des technologies abordées. Il a été une grande source d'enrichissement pour moi pendant toute la durée du projet.
En revanche, je regrette que le déploiement de l'application Android n'ait pas prévu que le smartphone soit équipé de carte SIM. Cette restriction m'a déçue dans le sens où il m'aurait plu d'ajouter un système de cartographie des rondes.

23 - Annexes

23.1 - Mise en place de l'accès root sur un smartphone

Le « rootage » d'un smartphone étant une opération pouvant endommager le smartphone, il est fortement recommander d'utiliser un tutoriel adapté au modèle de votre smartphone. Certaines procédures peuvent se ressembler mais il n'en existe aucune qui soit universelle.
En ce qui me concerne, il m'a fallu « flasher » un système de récupération customisé et, grâce à ce dernier, flasher une fois de plus un paquet du nom de SuperSU pour avoir les accès root sur mon smartphone.
ATTENTION : Une fois le smartphone rooté, certaines applications et fonctionnalités ne fonctionneront plus (Ex : ApplePay, McAfee...)

23.2 - Fonctionnement d'une application Android



Lorsqu'une application est lancée, le système Android fait appel aux fonctions comme décrites ci-dessus au fur et à mesure de l'avancement de l'application.

Les plus utilisées sont les fonctions onCreate et onResume qui permettent d'effectuer des tâches à la création (instanciation) d'une classe et à la récupération de cette dernière. Les fonctions comme onPause, onStop ou onDestroy ont un ordre, selon mon expérience, un peu flou et fournissent une bonne alternative aux destructeurs.

23.3 - Installation de sqlite3 sur Android

Afin d'installer sqlite3 sous Android, il vous faut pour commencer un smartphone avec les accès root.

Étape 1 : Téléchargez et installez ensuite l'application Titanium Backup sur le Google play.

Étape 2 : Lancez et quittez l'application Titanium Backup.

Étape 3 : Lancez un navigateur de fichier comme « Root browser » qui permet de naviguer dans les fichiers root et allez vers « /data/data/com.keramidas.TitaniumBackup/files/ »

Étape 4 : Copier le fichier « sqlite3 » et collez le dans « /system/xbin/ »

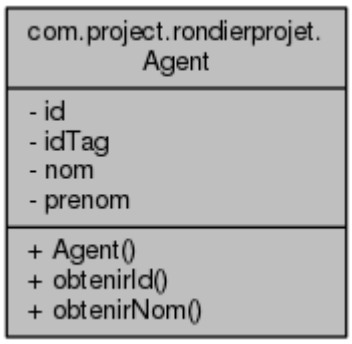
Étape 5 : Définissez les permissions du fichier à « -rwxr-xr-x »

Étape 6 : Redémarrez le smartphone. Sqlite est maintenant installé et accessible via adb.

23.4 - Doxygen de l'application Android

23.4.1 - Documentation des classes

23.4.1.1 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.AGENT



FONCTIONS MEMBRES PUBLIQUES

Agent (int **id**, String **idTag**, String **nom**, String **prenom**)

*Constructeur **Agent** Permet de créer un agent en fournissant son id, son nom ainsi que son prénom.*

int **obtenirId** ()

Obtenir l'id d'un agent.

String **obtenirNom** ()

Obtenir le nom d'un agent.

ATTRIBUTS PRIVÉS

int **id**
String **idTag**
String **nom**
String **prenom**

DOCUMENTATION DU CONSTRUCTEUR

com.project.rondierprojet.Agent.Agent (int *id*, String *idTag*, String *nom*, String *prenom*)

Constructeur **Agent** Permet de créer un agent en fournissant son id, son nom ainsi que son prénom.

Paramètres:

idTag	Id correspondant au tag MiFare associé à l'agent.
nom	Nom de l'agent.
prenom	Prénom de l'agent.

DOCUMENTATION DES FONCTIONS MEMBRES

int com.project.rondierprojet.Agent.obtenirId ()

Obtenir l'id d'un agent.

Permet d'obtenir l'id attribué à un agent.

Renvoie:

Retourne l'id de l'agent.

String com.project.rondierprojet.Agent.obtenirNom ()

Obtenir le nom d'un agent.

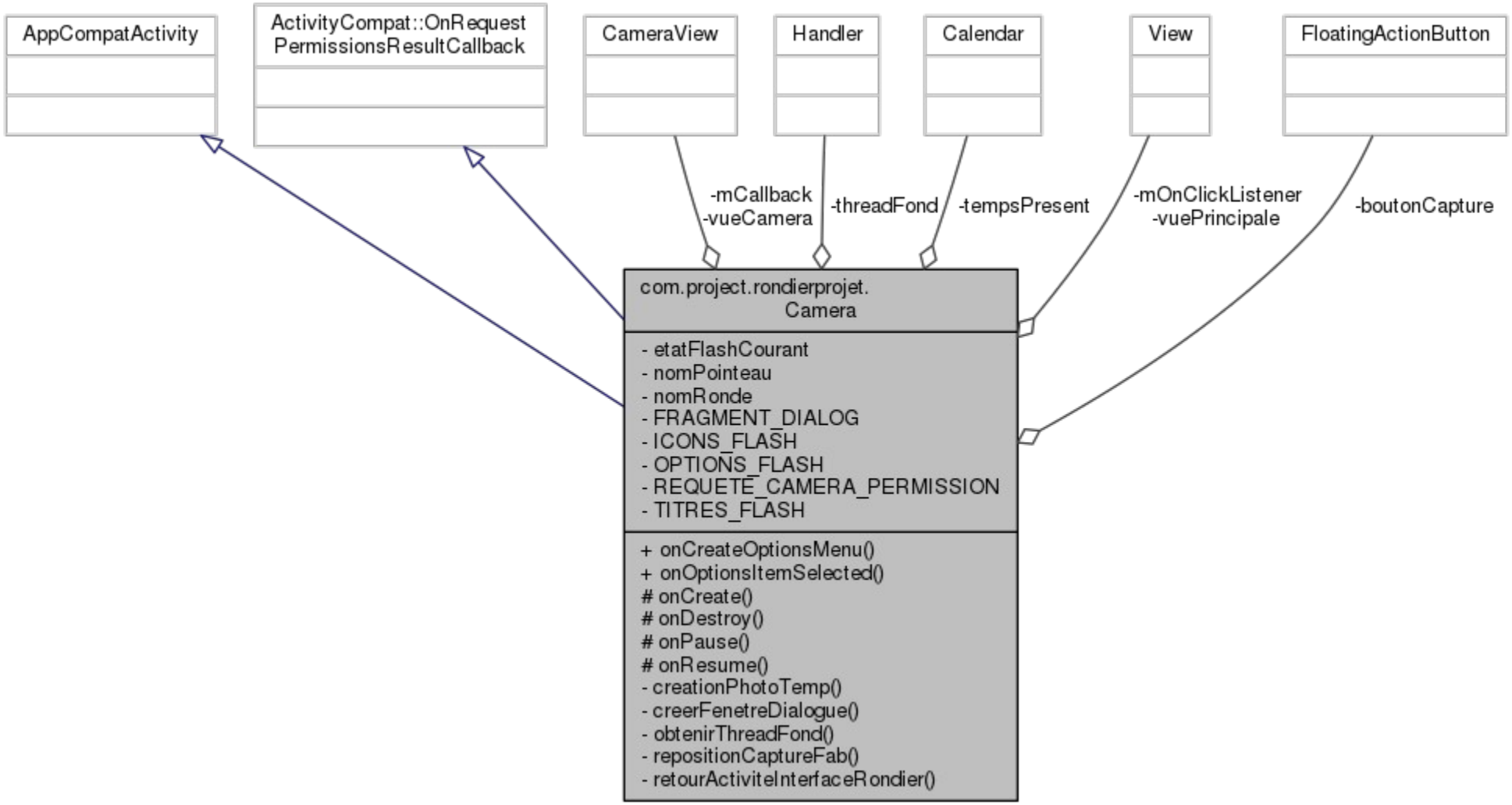
Premet d'obtenir le nom attribué à un agent.

Renvoie:

Le nom de l'agent.

23.4.1.2 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.CAMERA

Graphe de collaboration de com.project.rondierprojet.Camera:



FONCTIONS MEMBRES PUBLIQUES

boolean **onCreateOptionsMenu** (Menu menu)

Méthode créant le menu de la barre d'outils.

boolean **onOptionsItemSelected** (MenuItem item)

Méthode en charge des événements de la barre d'outils.

FONCTIONS MEMBRES PROTÉGÉES

void **onCreate** (Bundle savedInstanceState)

Méthode appelée automatiquement après l'instanciation de la classe.

void **onDestroy** ()

Méthode appelée automatiquement lorsque l'activité est détruite par le système Android.

void **onPause** ()

Méthode appelée automatiquement lorsque l'activité est mise en pause.

void **onResume** ()

Méthode appelée automatiquement après l'initialisation de la classe.

FONCTIONS MEMBRES PRIVÉES

File **creationPhotoTemp** (File dossierTemp, byte[] data)

Méthode permettant d'écrire la photo capturée dans la mémoire du smartphone.

void **creerFenetreDialogue** (ImageView tmpImage, final byte[] data, final File photoTemp)

Méthode permettant de créer une fenêtre de dialogue.

Handler **obtenirThreadFond** ()

Méthode permettant de créer un thread pour l'arrière plan de l'activité.

void **repositionCaptureFab** ()

Méthode permettant de replacer le bouton de capture de photo.

void **retourActiviteInterfaceRondier** ()

*Méthode permettant de retourner à l'activité **InterfaceRondier**.*

ATTRIBUTS PRIVÉS

FloatingActionButton **boutonCapture**
int **etatFlashCourant**
CameraView.Callback **mCallback**
View.OnClickListener **mOnClickListener**
String **nomPointeau**
String **nomRonde**
Calendar **tempsPresent**
Handler **threadFond**
CameraView **vueCamera**
View **vuePrincipale**

ATTRIBUTS PRIVÉS STATIQUES

static final String **FRAGMENT_DIALOG** = "dialog"
static final int[] **ICONS_FLASH**
static final int[] **OPTIONS_FLASH**
static final int **REQUETE_CAMERA_PERMISSION** = 1
static final int[] **TITRES_FLASH**

DOCUMENTATION DES FONCTIONS MEMBRES

File com.project.rondierprojet.Camera.creationPhotoTemp (File *dossierTemp*, byte[] *data*) `[private]`

Méthode permettant d'écrire la photo capturée dans la mémoire du smartphone.

Cette méthode écrit dans un dossier temporaire passé en paramètre la photo passée en paramètre.

Paramètres:

dossierTemp	Le chemin vers le dossier temporaire.
data	Les données au format JPEG de l'image.

Renvoie:

La photo temporaire créée.

void com.project.rondierprojet.Camera.creerFenetreDialogue (ImageView *tmpImage*, final byte[] *data*, final File *photoTemp*) `[private]`

Méthode permettant de créer une fenêtre de dialogue.

Cette méthode crée et affiche une fenêtre de dialogue permettant de montrer une photo entrée en paramètre à l'utilisateur. L'utilisateur peut ensuite décider de conserver cette photo et donc de l'enregistrer dans un répertoire défini par le système ou refuser de garder cette photo. Les photos enregistrées sont situées dans /sdcard/Android/data/<nom paquet="" application>=""></nom>/files/Pictures.

Paramètres:

tmpImage	La prévisualisation de l'image dans l'interface.
data	Les données au format JPEG de l'image.
photoTemp	L'image utilisée pour créer la prévisualisation.

Handler com.project.rondierprojet.Camera.obtenirThreadFond () `[private]`

Méthode permettant de créer un thread pour l'arrière plan de l'activité.

Cette méthode crée, s'il n'en existe pas déjà un, un thread utilisé pour actualiser le fond de l'activité (ici l'aperçu que fourni la camera).

Renvoie:

Le thread créé.

void com.project.rondierprojet.Camera.onCreate (Bundle *savedInstanceState*) `[protected]`

Méthode appelée automatiquement après l'instanciation de la classe.

La méthode onCreate est une méthode appelée par le système Android à l'instanciation de la classe. Cette méthode est ici utilisée pour attacher les différents éléments du code avec l'interface graphique.

Paramètres:

savedInstanceState	Etat antérieur de l'activité.

boolean com.project.rondierprojet.Camera.onCreateOptionsMenu (Menu *menu*)

Méthode créant le menu de la bar d'outils.

Cette méthode crée dynamiquement les éléments du menu de la bar d'outils.

Paramètres:

menu	Le menu qui doit être créé.

Renvoie:

Vrai dans tous les cas.

void com.project.rondierprojet.Camera.onDestroy () `[protected]`

Méthode appelée automatiquement lorsque l'activité est détruite par le système Android.

La méthode onDestroy est une méthode appelée par le système Android lorsque ce dernier détruit l'activité (besoin de ressources système, application fermée, etc). Cette méthode est ici utilisée pour détruire, en fonction de la version Android, l'interface de prévision de façon correcte.

boolean com.project.rondierprojet.Camera.onOptionsItemSelected (MenuItem *item*)

Méthode en charge des événements de la bar d'outils.

Cette méthode change les icones de la bar d'outils ainsi que les réglages de la caméra lorsque l'utilisateur clique sur les éléments de la bar d'outils.

Paramètres:

item	L'item du menu sélectionné.

Renvoie:

Faux dans tous les cas.

void com.project.rondierprojet.Camera.onPause () `[protected]`

Méthode appelée automatiquement lorsque l'activité est mise en pause.

La méthode onPause est une méthode appelée par le système Android à la mise en pause de cette activité (application mise en arrière plan, verrouillage du smartphone, etc.) Cette méthode est ici utilisée pour arrêter l'aperçu de la caméra lorsque l'activité n'est plus visible par l'utilisateur.

void com.project.rondierprojet.Camera.onResume () `[protected]`

Méthode appelée automatiquement après l'initialisation de la classe.

La méthode onResume est une méthode appelée par le système Android après l'initialisation de la classe. Cette méthode est ici utilisée pour vérifier si l'application dispose de toutes les autorisations nécessaires et, si ça n'est pas le cas, demande à l'utilisateur de les lui accorder.

void com.project.rondierprojet.Camera.repositionCaptureFab () `[private]`

Méthode permettant de replacer le bouton de capture de photo.

Cette méthode permet de repositionner le bouton de capture en fonction de la taille de l'écran sur lequel l'application est lancée.

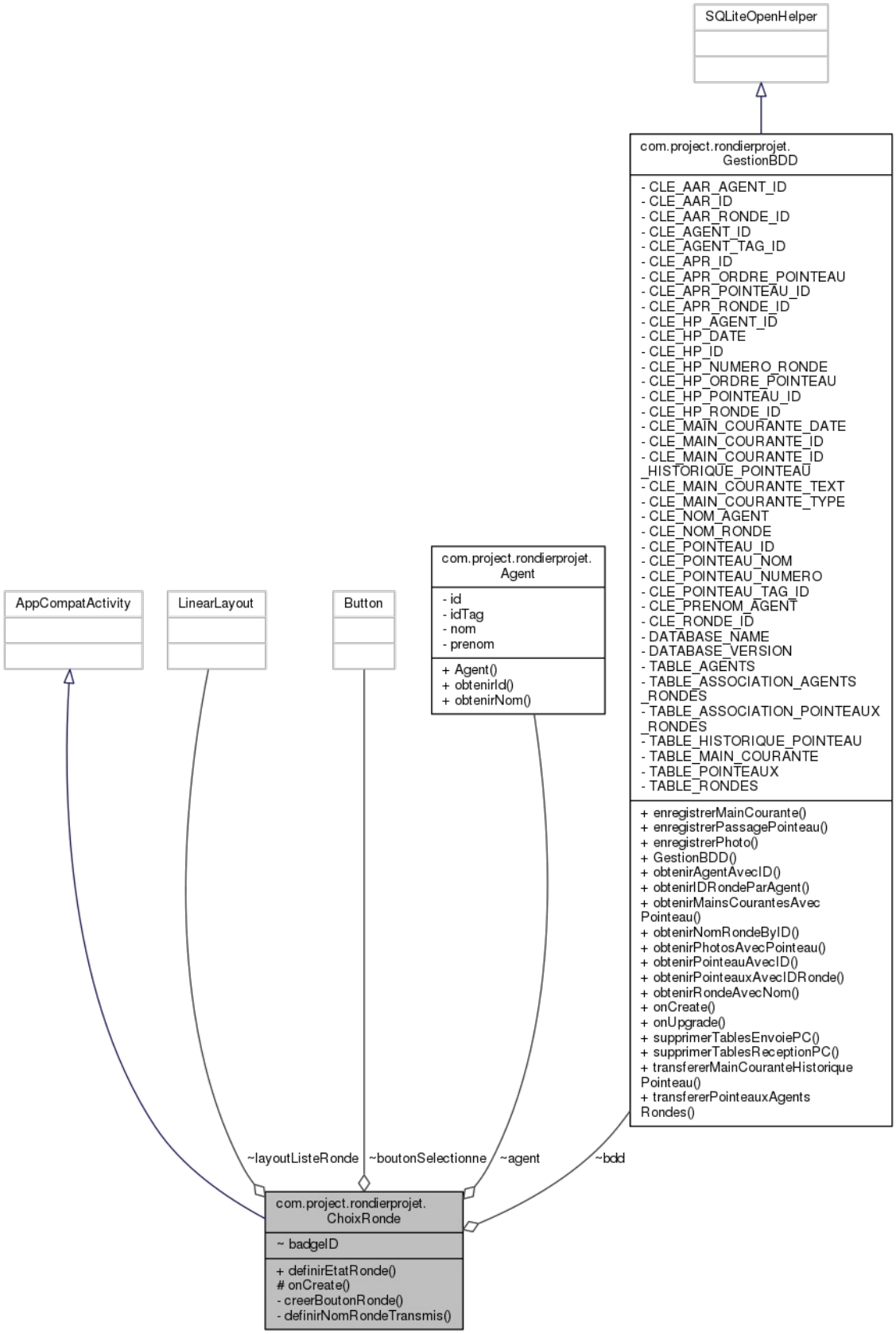
void com.project.rondierprojet.Camera.retourActiviteInterfaceRondier () `[private]`

Méthode permettant de retourner à l'activité **InterfaceRondier**.

Cette méthode permet de retourner à l'activité **InterfaceRondier** tout en terminant l'activité **Camera**.

23.4.1.3 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.CHOIXRONDE

Graphe de collaboration de com.project.rondierprojet.ChoixRonde:



FONCTIONS MEMBRES PUBLIQUES

void **definirEtatRonde** (boolean etatRonde)

Definir si une ronde est en cours ou non dans les fichiers partagers.

FONCTIONS MEMBRES PROTÉGÉES

void **onCreate** (Bundle savedInstanceState)

Methode appelée au lancement de l'activité.

FONCTIONS MEMBRES PRIVÉES

void **creerBoutonRonde** (String text)

Créer un bouton pour la sélection d'une ronde par l'agent.

void **definirNomRondeTransmis** ()

Définir l'id de l'agent qui lance la ronde ainsi que le nom de la ronde sélectionnée.

DOCUMENTATION DES FONCTIONS MEMBRES

void com.project.rondierprojet.ChoixRonde.creerBoutonRonde (String *text*) `[private]`

Créer un bouton pour la sélection d'une ronde par l'agent.

Permet de créer un bouton du nom d'une ronde disponnible pour l'agent. Cette méthode crée aussi les évènements lorsqu'un clique est fait sur le bouton.

Paramètres:

text	Le texte à mettre dans le bouton.

void com.project.rondierprojet.ChoixRonde.definirEtatRonde (boolean *etatRonde*)

Definir si une ronde est en cours ou non dans les fichiers partagers.

Permet de définir dans les paramètres communs à toutes les activités si une ronde est en cours ou non.

Paramètres:

etatRonde	Retourne vrai si une ronde est en cours sinon retourne faux.

void com.project.rondierprojet.ChoixRonde.definirNomRondeTransmis () `[private]`

Définir l'id de l'agent qui lance la ronde ainsi que le nom de la ronde sélectionnée.

Permet dé définir dans les paramètres communs à toutes les activités l'id de l'agent qui lance une ronde ainsi que le nom de la ronde qui est lancée.

void com.project.rondierprojet.ChoixRonde.onCreate (Bundle *savedInstanceState*) `[protected]`

Methode appelée au lancement de l'activité.

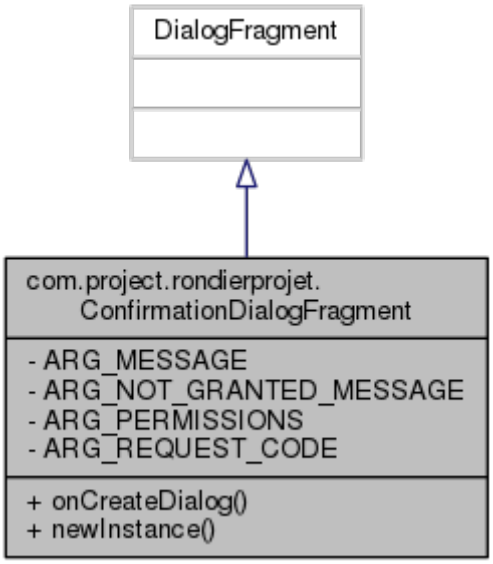
La méthode onCreate est une méthode appelée par le système Android au lancement de l'activité associée. Cette méthode est ici utilisée pour récupérer l'ID des rondes associées à l'agent s'étant identifié précédemment. Grâce aux ID de ces rondes, cette méthode crée une liste de rondes disponnible pour l'agent et l'affiche sur l'interface associée ainsi qu'un bouton valider.

Paramètres:

savedInstanceState	Etat antérieur de l'activité.

23.4.1.4 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.CONFIRMATIONDIALOGFRAGMENT

Graphe de collaboration de com.project.rondierprojet.ConfirmationDialogFragment:



FONCTIONS MEMBRES PUBLIQUES

Dialog **onCreateDialog** (Bundle savedInstanceState)

FONCTIONS MEMBRES PUBLIQUES STATIQUES

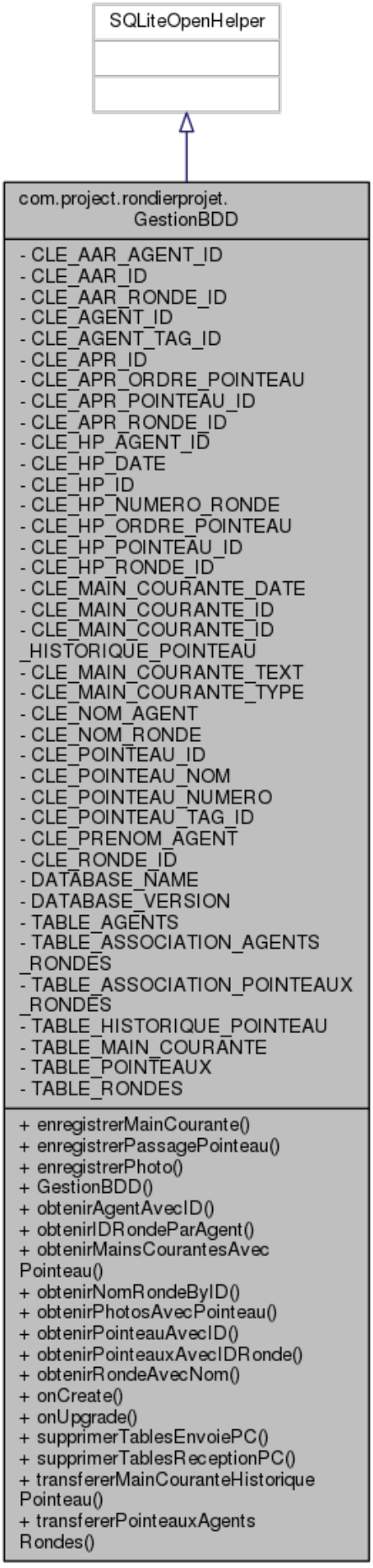
static **ConfirmationDialogFragment newInstance** (@StringRes int message, String[] permissions, int requestCode,@StringRes int notGrantedMessage)

ATTRIBUTS PRIVÉS STATIQUES

static final String **ARG_MESSAGE** = "message"
static final String **ARG_NOT_GRANTED_MESSAGE** = "not_granted_message"
static final String **ARG_PERMISSIONS** = "permissions"
static final String **ARG_REQUEST_CODE** = "request_code"

23.4.1.5 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.GESTIONBDD

Graphe de collaboration de com.project.rondierprojet.GestionBDD:


SQLiteOpenHelper
com.project.rondierprojet.
GestionBDD
- CLE_AAR_AGENT_ID
- CLE_AAR_ID
- CLE_AAR_RONDE_ID
- CLE_AGENT_ID
- CLE_AGENT_TAG_ID
- CLE_APR_ID
- CLE_APR_ORDRE_POINTEAU
- CLE_APR_POINTEAU_ID
- CLE_APR_RONDE_ID
- CLE_HP_AGENT_ID
- CLE_HP_DATE
- CLE_HP_ID
- CLE_HP_NUMERO_RONDE
- CLE_HP_ORDRE_POINTEAU
- CLE_HP_POINTEAU_ID
- CLE_HP_RONDE_ID
- CLE_MAIN_COURANTE_DATE
- CLE_MAIN_COURANTE_ID
- CLE_MAIN_COURANTE_ID
_HISTORIQUE_POINTEAU
- CLE_MAIN_COURANTE_TEXT
- CLE_MAIN_COURANTE_TYPE
- CLE_NOM_AGENT
- CLE_NOM_RONDE
- CLE_POINTEAU_ID
- CLE_POINTEAU_NOM
- CLE_POINTEAU_NUMERO
- CLE_POINTEAU_TAG_ID
- CLE_PRENOM_AGENT
- CLE_RONDE_ID
- DATABASE_NAME
- DATABASE_VERSION
- TABLE_AGENTS
- TABLE_ASSOCIATION_AGENTS
_RONDES
- TABLE_ASSOCIATION_POINTEAUX
_RONDES
- TABLE_HISTORIQUE_POINTEAU
- TABLE_MAIN_COURANTE
- TABLE_POINTEAUX
- TABLE_RONDES
+ enregistrerMainCourante()
+ enregistrerPassagePointeau()
+ enregistrerPhoto()
+ GestionBDD()
+ obtenirAgentAvecID()
+ obtenirIDRondeParAgent()
+ obtenirMainsCourantesAvec
Pointeau()
+ obtenirNomRondeByID()
+ obtenirPhotosAvecPointeau()
+ obtenirPointeauAvecID()
+ obtenirPointeauxAvecIDRonde()
+ obtenirRondeAvecNom()
+ onCreate()
+ onUpgrade()
+ supprimerTablesEnvoiePC()
+ supprimerTablesReceptionPC()
+ transfererMainCouranteHistorique
Pointeau()
+ transfererPointeauxAgents
Rondes()

FONCTIONS MEMBRES PUBLIQUES

void **enregistrerMainCourante** (String enregistrement)

Insère une nouvelle ligne dans la table MainsCourantes Cette méthode permet d'insérer une nouvelle ligne dans la table MainsCourantes.

void **enregistrerPassagePointeau** (int idAgent, int idPointeau, int idRonde, int ordrePointeau, int numeroRonde)

Insère une nouvelle ligne dans la table HistoriquePointeau Cette méthode permet d'insérer une nouvelle ligne dans la table HistoriquePointeau.

void **enregistrerPhoto** (String chemin)

Insère une nouvelle ligne dans la table MainsCourantes Cette méthode permet d'insérer une nouvelle ligne dans la table MainsCourantes.

GestionBDD (Context context)

Constructeur appelé à la création d'une instance bdd.

Agent obtenirAgentAvecID (String badgeID)

Retourne un agent en fonction de l'ID passé en paramètre.

ArrayList< Integer > **obtenirIDRondeParAgent** (**Agent** agent)

Retourne une liste d'entier correspondants aux ID des rondes associées à un agent.

ArrayList< String > **obtenirMainsCourantesAvecPointeau** (**Pointeau** pointeau)

Retourne toutes les mains courantes correspondantes au pointeau passé en paramètre.

String **obtenirNomRondeByID** (int id)

Retourne le nom d'une ronde grâce à son ID.

ArrayList< String > **obtenirPhotosAvecPointeau** (**Pointeau** pointeau)

Retourne toutes photos correspondantes au pointeau passé en paramètre.

Pointeau obtenirPointeauAvecID (String idTag)

Retourne un pointeau grâce à l'ID passé en paramètre.

List< **Pointeau** > **obtenirPointeauxAvecIDRonde** (int idRonde)

Retourne une liste de pointeaux qui correspond aux pointeaux associés à une ronde.

Ronde obtenirRondeAvecNom (String nom)

Retourne une ronde grâce au nom passé en paramètre.

void **onCreate** (SQLiteDatabase db)

Méthode appelée automatiquement après l'instanciation de la classe.

void **onUpgrade** (SQLiteDatabase db, int oldVersion, int newVersion)

*Supprimer, si les tables existent, l'intégralité des tables pour ensuite les recréer via la méthode **onCreate()**.*

void **supprimerTablesEnvoiePC** ()

Vide les tables Agents, Rondes, Pointeaux, Association Pointeaux/Rondes, Association Agents/Rondes.

void **supprimerTablesReceptionPC** ()

*Vide les tables Historique **Pointeau** et Main Courante.*

void **transfererMainCouranteHistoriquePointeau** (SQLiteDatabase dbSortie)

Transfère les données des tables MainCourante et HistoriquePointeau dans la base de données passée en paramètre.

void **transfererPointeauxAgentsRondes** (SQLiteDatabase dbEntree)

Transfère les données des tables Agents, Rondes, Pointeaux, AssociationAgentsRondes et AssociationPointeauxRondes dans la base de données passée en paramètre.

ATTRIBUTS PRIVÉS STATIQUES

static final String **CLE_AAR_AGENT_ID** = "idAgent"
static final String **CLE_AAR_ID** = "id"
static final String **CLE_AAR_RONDE_ID** = "idRonde"
static final String **CLE_AGENT_ID** = "idAgent"
static final String **CLE_AGENT_TAG_ID** = "idTag"
static final String **CLE_APR_ID** = "id"
static final String **CLE_APR_ORDRE_POINTEAU** = "ordrePointeau"
static final String **CLE_APR_POINTEAU_ID** = "idPointeau"
static final String **CLE_APR_RONDE_ID** = "idRonde"
static final String **CLE_HP_AGENT_ID** = "idAgent"
static final String **CLE_HP_DATE** = "date"
static final String **CLE_HP_ID** = "id"
static final String **CLE_HP_NUMERO_RONDE** = "numeroRonde"
static final String **CLE_HP_ORDRE_POINTEAU** = "ordrePointeau"
static final String **CLE_HP_POINTEAU_ID** = "idPointeau"
static final String **CLE_HP_RONDE_ID** = "idRonde"
static final String **CLE_MAIN_COURANTE_DATE** = "date"
static final String **CLE_MAIN_COURANTE_ID** = "id"
static final String **CLE_MAIN_COURANTE_ID_HISTORIQUE_POINTEAU** = "idHistoriquePointeau"
static final String **CLE_MAIN_COURANTE_TEXT** = "texte"
static final String **CLE_MAIN_COURANTE_TYPE** = "type"
static final String **CLE_NOM_AGENT** = "nom"
static final String **CLE_NOM_RONDE** = "nom"
static final String **CLE_POINTEAU_ID** = "idPointeau"
static final String **CLE_POINTEAU_NOM** = "nom"
static final String **CLE_POINTEAU_NUMERO** = "numero"
static final String **CLE_POINTEAU_TAG_ID** = "idTag"
static final String **CLE_PRENOM_AGENT** = "prenom"
static final String **CLE_RONDE_ID** = "idRonde"
static final String **DATABASE_NAME** = "ControleurDeRonde.db"
static final int **DATABASE_VERSION** = 37
static final String **TABLE_AGENTS** = "Agents"
static final String **TABLE_ASSOCIATION_AGENTS_RONDES** = "AssociationAgentsRondes"
static final String **TABLE_ASSOCIATION_POINTEAUX_RONDES** = "AssociationPointeauxRondes"
static final String **TABLE_HISTORIQUE_POINTEAU** = "HistoriquePointeau"
static final String **TABLE_MAIN_COURANTE** = "MainCourante"
static final String **TABLE_POINTEAUX** = "Pointeaux"
static final String **TABLE_RONDES** = "Rondes"

DOCUMENTATION DES CONSTRUCTEURS ET DESTRUCTEUR

com.project.rondierprojet.GestionBDD.GestionBDD (Context *context*)

Constructeur appelé à la création d'une instance bdd.

Ce constructeur permet de créer le fichier sqlite s'il n'existe pas déjà et de réinitialiser les tables si un changement de version a eu lieu.

Paramètres:

context	Le contexte de l'application.

DOCUMENTATION DES FONCTIONS MEMBRES

void com.project.rondierprojet.GestionBDD.enregistrerMainCourante (String *enregistrement*)

Insère une nouvelle ligne dans la table MainsCourantes Cette méthode permet d'insérer une nouvelle ligne dans la table MainsCourantes.

Cette ligne contient le texte saisie (passé en paramètre de la méthode), l'ID du dernier pointeau scanné ainsi que le type de la main courante (ici 0 qui correspond à une main courante textuelle).

Paramètres:

enregistrement	La main courante qui doit être enregistrée.

void com.project.rondierprojet.GestionBDD.enregistrerPassagePointeau (int *idAgent*, int *idPointeau*, int *idRonde*, int *ordrePointeau*, int *numeroRonde*)

Insère une nouvelle ligne dans la table HistoriquePointeau Cette méthode permet d'insérer une nouvelle ligne dans la table HistoriquePointeau.

Cette ligne contient l'idAgent, l'idPointeau, l'idRonde et l'ordre de passage saisis en paramètre de la méthode. En plus de ces informations, la ligne est accompagnée par la date d'enregistrement au format AAAA-MM-DD hh:mm:ss.

Paramètres:

idAgent	L'ID de l'agent qui a scanné le pointeau.
idPointeau	L'ID du pointeau qui a été scanné.
idRonde	L'ID de la ronde dans laquelle le pointeau a été scanné.
ordrePointeau	L'ordre dans lequel le pointeau a été scanné.

void com.project.rondierprojet.GestionBDD.enregistrerPhoto (String *chemin*)

Insère une nouvelle ligne dans la table MainsCourantes Cette méthode permet d'insérer une nouvelle ligne dans la table MainsCourantes.

Cette ligne contient le chemin où la photo est enregistrée (passé en paramètre de la méthode), l'ID du dernier pointeau scanné ainsi que le type de la main courante (ici 1 qui correspond à une photo).

Paramètres:

chemin	Le chemin dans lequel la photo va être enregistrée.

Agent com.project.rondierprojet.GestionBDD.obtenirAgentAvecID (String *badgeID*)

Retourne un agent en fonction de l'ID passé en paramètre.

Retourne un agent composé de son nom, son prénom ainsi que l'ID de son badge grâce à l'ID du badge passé en paramètre. Si aucun agent n'est connu dans la base de données avec l'ID de ce badge, null est renvoyé.

Paramètres:

badgeID	L'ID du tag MiFare utilisé pour retrouver l'agent.

Renvoie:

Si un agent existe, retourne l'agent. Sinon, retourne null.

ArrayList<Integer> com.project.rondierprojet.GestionBDD.obtenirIDRondeParAgent (Agent *agent*)

Retourne une liste d'entier correspondants aux ID des rondes associées à un agent.

Retourne une liste d'entier correspondants aux ID des rondes associées inscrits dans la table association agents-rondes pour un agent passé en paramètre.

Paramètres:

agent	L'agent avec lequel nous récupérons la liste des rondes.

Renvoie:

La liste des rondes sous forme d'ID.

ArrayList<String> com.project.rondierprojet.GestionBDD.obtenirMainsCourantesAvecPointeau (Pointeau *pointeau*)

Retourne toutes les mains courantes correspondantes au pointeau passé en paramètre.

Cette méthode retourne toutes les mains courantes ayant été saisies qui correspondent au pointeau passé en paramètre. Le tableau de string retourné comporte aussi les heures et dates auxquelles les mains courantes ont été saisies.

Paramètres:

pointeau	Le pointeau utilisé pour retrouver la liste des mains courantes associées.

Renvoie:

Liste des mains courantes avec l'heure et la date correspondant sous forme de tableau de string.

String com.project.rondierprojet.GestionBDD.obtenirNomRondeByID (int *id*)

Retourne le nom d'une ronde grâce à son ID.

Retourne une chaine de caractère correspondant à l'ID de la ronde passé en paramètre.

Paramètres:

id	L'identifiant de la ronde dont nous voulons récupérer le nom.

Renvoie:

Nom de la ronde.

ArrayList<String> com.project.rondierprojet.GestionBDD.obtenirPhotosAvecPointeau (Pointeau *pointeau*)

Retourne toutes photos correspondantes au pointeau passé en paramètre.

Cette méthode retourne toutes les photos ayant été prises qui correspondent au pointeau passé en paramètre.

Paramètres:

pointeau	Le pointeau utilisé pour retrouver la liste des photos associées.

Renvoie:

Liste des photos sous forme de tableau de string.

Pointeau com.project.rondierprojet.GestionBDD.obtenirPointeauAvecID (String *idTag*)

Retourne un pointeau grâce à l'ID passé en paramètre.

Retourne un pointeau constitué d'un ID, de l'ID du tag MiFare associé, de son nom et du numéro qui lui est attribué.

Paramètres:

idTag	L'ID du tag MiFare utilisée pour retrouver le pointeau.

Renvoie:

Si un pointeau est trouvé, retourne le pointeau. Sinon renvoi null.

List<Pointeau> com.project.rondierprojet.GestionBDD.obtenirPointeauxAvecIDRonde (int *idRonde*)

Retourne une liste de pointeaux qui correspond aux pointeaux associés à une ronde.

Retourne une liste de pointeaux correspondant pointeaux inscrits dans la table d'association rondes-pointeaux pour l'ID de la ronde passé en paramètre.

Paramètres:

idRonde	L'ID de la ronde utilisé pour retrouver la liste des pointeaux associés.

Renvoie:

La liste des pointeaux demandés.

Ronde com.project.rondierprojet.GestionBDD.obtenirRondeAvecNom (String *nom*)

Retourne une ronde grâce au nom passé en paramètre.

Retourne une ronde composée de son ID et de son nom grâce à une chaine de caractère passée en paramètre.

Paramètres:

nom	Le nom de la ronde que nous voulons récupérer.

Renvoie:

Si une ronde existe, retourne la ronde. Sinon, retourne null.

void com.project.rondierprojet.GestionBDD.onCreate (SQLiteDatabase *db*)

Méthode appelée automatiquement après l'instanciation de la classe.

La méthode onCreate est une méthode appelée par le système Android à l'instanciation de la classe. Cette méthode est ici utilisée pour créer toutes les tables nécessaires au bon fonctionnement du système. Si les tables existent déjà, rien n'est changé. Toutes les créations sont faites grâce aux variables précédemment définies.

Paramètres:

db	La base de données dans laquelle il faut créer les tables.

void com.project.rondierprojet.GestionBDD.onUpgrade (SQLiteDatabase *db*, int *oldVersion*, int *newVersion*)

Supprimer, si les tables existent, l'intégralité des tables pour ensuite les recréer via la méthode **onCreate()**.

Permet de supprimer l'intégralité des tables si elles existent et de relancer leurs créations au changement du numéro de version de la base de données. Cette méthode permet aussi de remplir les tables selon une base de données type pour effectuer des testes.

Paramètres:

db	La base de données concernée par la mise à jour.
oldVersion	Le numéro de l'ancienne version de la base de données.
newVersion	Le numéro de la nouvelle version de la base de données.

void com.project.rondierprojet.GestionBDD.supprimerTablesEnvoiePC ()

Vide les tables Agents, Rondes, Pointeaux, Association Pointeaux/Rondes, Association Agents/Rondes.

Cette méthode permet de supprimer les tables Agents, Rondes, Pointeaux, Association Pointeaux/Rondes et Association Agents/Rondes puis les recrée via la fonction **onCreate()**.

void com.project.rondierprojet.GestionBDD.supprimerTablesReceptionPC ()

Vide les tables Historique **Pointeau** et Main Courante.

Cette méthode permet de supprimer les tables Historique **Pointeau** et Main Courante puis les recrée via la fonction **onCreate()**.

void com.project.rondierprojet.GestionBDD.transfererMainCouranteHistoriquePointeau (SQLiteDatabase *dbSortie*)

Transfère les données des tables MainCourante et HistoriquePointeau dans la base de données passée en paramètre.

Cette méthdoe transfère les données des tables MainCourante et HistoriquePointeau présentent sur le smartphone vers une base de données passée en paramètre.

Paramètres:

dbSortie	La base de données utilisée pour envoyer les données vers le poste de supervision.

void com.project.rondierprojet.GestionBDD.transfererPointeauxAgentsRondes (SQLiteDatabase *dbEntree*)

Transfère les données des tables Agents, Rondes, Pointeaux, AssociationAgentsRondes et AssociationPointeauxRondes dans la base de données passée en paramètre.

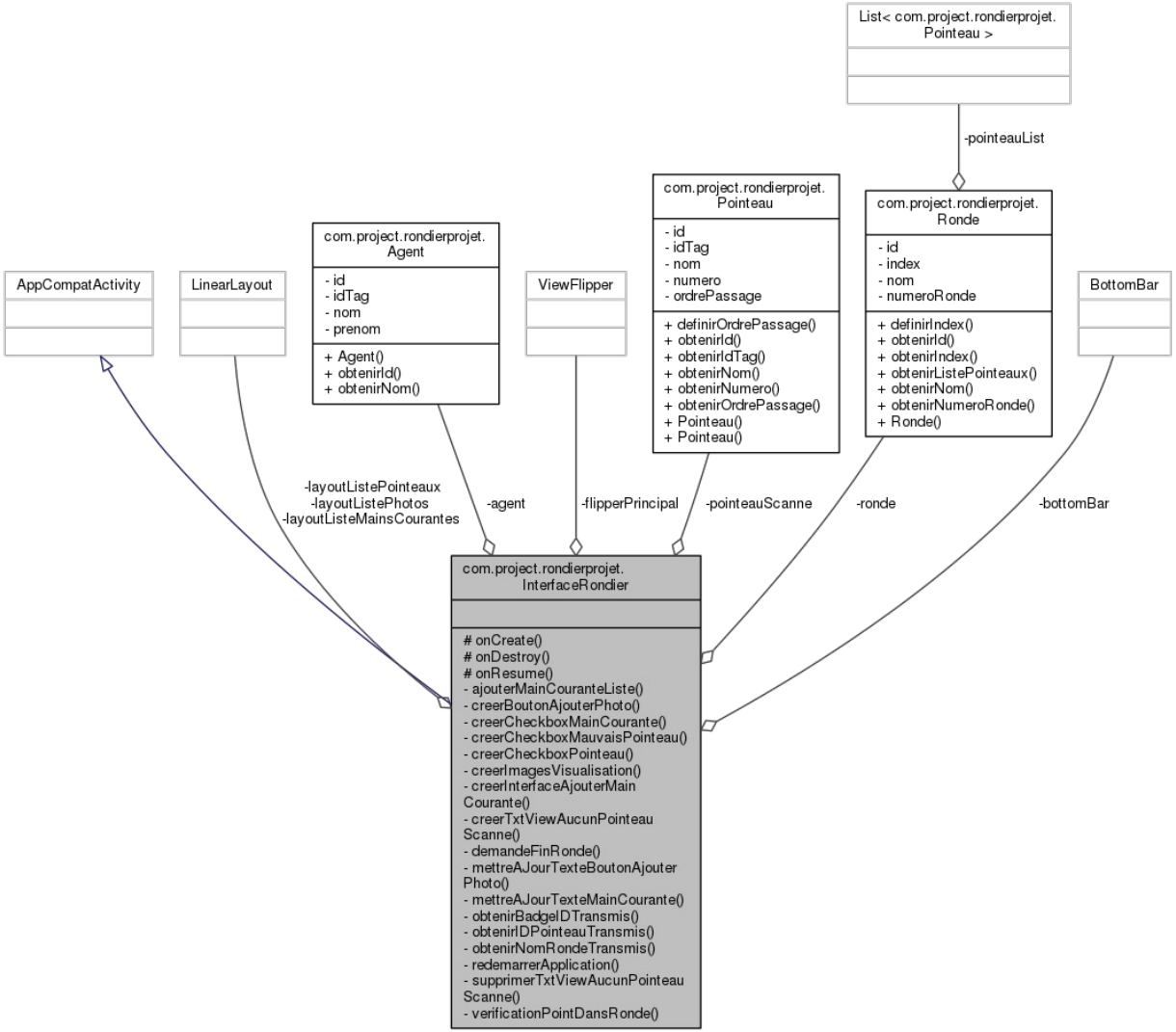
Cette méthdoe transfère les données des tables Agents, Rondes, Pointeaux, AssociationAgentsRondes et AssociationPointeauxRondes présentent sur le smartphone vers une base de données passée en paramètre.

Paramètres:

dbEntree	La base de données utilisée pour récupérer les données du poste de supervision.

23.4.1.6 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.INTERFACERONDIER

Graphe de collaboration de com.project.rondierprojet.InterfaceRondier:



FONCTIONS MEMBRES PROTÉGÉES

void **onCreate** (Bundle savedInstanceState)

Méthode appelée au lancement de l'activité.

void **onDestroy** ()

Méthode appelée automatiquement lorsque l'activité est détruite par le système Android.

void **onResume** ()

Méthode appelée automatiquement après l'initialisation de la classe.

FONCTIONS MEMBRES PRIVÉES

void **ajouterMainCouranteListe** (String mainCourante)

Méthode ajoutant la main courante saisie à la liste dans l'interface d'ajout de main courante.

void **creerBoutonAjouterPhoto** ()

Méthode créant le bouton d'ajout de photo.

void **creerCheckboxMainCourante** (int index, String mainCourante, String date)

Méthode créant une checkbox pour la main courante passée en paramètre.

void **creerCheckboxMauvaisPointeau** (**Pointeau** pointeau, int indexRonde)

Méthode créant et plaçant une checkbox pour le pointeau passé en paramètre à l'index passé en paramètre.

void **creerCheckboxPointeau** (int index)

Méthode créant une checkbox pour le pointeau passé en paramètre.

void **creerImagesVisualisation** (**Pointeau** pointeau)

Méthode créant la liste des photos prises et correspondants au dernier pointeau scanné.

void **creerInterfaceAjouterMainCourante** ()

Méthode créant l'interface d'ajout de main courante.

void **creerTxtViewAucunPointeauScanne** ()

Méthode créant les interfaces (vierges avec un message d'alerte) pour la prise de photo et la saisie de main courante.

void **demandeFinRonde** ()

Méthode créant une boîte de dialogue demandant la confirmation de l'arrêt manuel d'une ronde.

void **mettreAJourTexteBoutonAjouterPhoto** ()

Méthode mettant à jour le bouton d'ajout de photo.

void **mettreAJourTexteMainCourante** ()

Méthode mettant à jour le bouton d'ajout de main courante.

String **obtenirBadgeIDTransmis** ()

Méthode retournant l'ID du badge de l'agent passé entre les différentes activités sous forme de string.

String **obtenirIDPointeauTransmis** ()

Méthode retournant l'ID du pointeau passé entre les différentes activités.

String **obtenirNomRondeTransmis** ()

Méthode retournant le nom de la ronde passé entre les différentes activités.

void **redemarrerApplication** ()

Méthode permettant de redémarrer l'application.

void **supprimerTxtViewAucunPointeauScanne** ()

Méthode supprimant les interfaces créées avec la fonction creerTxtViewAucunPointeauScanne.

boolean **verificationPointDansRonde** (int index)

Méthode vérifiant si le dernier pointeau scanné est égale au pointeau indiqué en paramètre.

ATTRIBUTS PRIVÉS

Agent agent
BottomBar **bottomBar**
ViewFlipper **flipperPrincipal**
LinearLayout **layoutListeMainsCourantes**
LinearLayout **layoutListePhotos**
LinearLayout **layoutListePointeaux**
Pointeau pointeauScanne
Ronde ronde

DOCUMENTATION DES FONCTIONS MEMBRES

void com.project.rondierprojet.InterfaceRondier.ajouterMainCouranteListe (String *mainCourante*) `[private]`

Méthode ajoutant la main courante saisie à la liste dans l'interface d'ajout de main courante.

Cette méthode ajoute la main courante passée en paramètre sous forme de string à la liste des mains courantes saisies de l'interface d'ajout de main courante.

Paramètres:

mainCourante	La main courante qui doit être ajoutée.

void com.project.rondierprojet.InterfaceRondier.creerBoutonAjouterPhoto () `[private]`

Méthode créant le bouton d'ajout de photo.

Cette méthode crée le bouton qui, quand pressé, lancera l'activité camera permettant à l'utilisateur de prendre une photo. Le bouton ici défini passe le nom de la ronde en cours ainsi que le nom du dernier point scanné à l'activité camera.

void com.project.rondierprojet.InterfaceRondier.creerCheckboxMainCourante (int *index*, String *mainCourante*, String *date*) `[private]`

Méthode créant une checkbox pour la main courante passée en paramètre.

Cette méthode crée une checkbox avec le texte de la main courante et la date correspondante qui ont été passés en paramètre. La checkbox ainsi créée est ensuite ajoutée à la liste des mains courantes affichée dans l'interface principale.

Paramètres:

mainCourante	La main courante utilisée pour créer la checkbox.
date	La date utilisée pour créer la checkbox.

void com.project.rondierprojet.InterfaceRondier.creerCheckboxMauvaisPointeau (Pointeau *pointeau*, int *indexRonde*) `[private]`

Méthode créant et plaçant une checkbox pour le pointeau passé en paramètre à l'index passé en paramètre.

Cette méthode crée une checkbox avec le nom du pointeau passé en paramètre inscrit en rouge. La checkbox ainsi créée est ensuite ajoutée à la liste des pointeau à scanner dans l'interface principale après le dernier pointeau coché.

Paramètres:

pointeau	Le pointeau utilisé pour créer la checkbox.
indexRonde	L'index auquel la checkbox doit être créée.

void com.project.rondierprojet.InterfaceRondier.creerCheckboxPointeau (int *index*) `[private]`

Méthode créant une checkbox pour le pointeau passé en paramètre.

Cette méthode crée une checkbox avec le nom du pointeau passé en paramètre grâce à son index. La checkbox ainsi créée est ensuite ajoutée à la liste des pointeaux à scanner dans l'interface principale.

Paramètres:

index	L'index auquel la checkbox doit être créée.

void com.project.rondierprojet.InterfaceRondier.creerImagesVisualisation (Pointeau *pointeau*) `[private]`

Méthode créant la liste des photos prises et correspondants au dernier pointeau scanné.

Cette méthode crée la liste des photos prises par l'utilisateur qui correspondent au dernier pointeau scanné passé en paramètre. Les photos sont récupérées et triées parmis le dossier où elles sont toutes stockées.

Paramètres:

pointeau	Pointeau utilisé pour trouver les photos.

void com.project.rondierprojet.InterfaceRondier.creerInterfaceAjouterMainCourante () [private]

Méthode créant l'interface d'ajout de main courante.

Cette méthode crée les différents layouts et boutons permettant la saisie de mains courantes par l'utilisateur. L'interface consiste en une zone de texte, un bouton de validation ainsi qu'une liste déroulante pour afficher les mains courantes saisies et validées.

void com.project.rondierprojet.InterfaceRondier.creerTxtViewAucunPointeauScanne () [private]

Méthode créant les interfaces (vierges avec un message d'alerte) pour la prise de photo et la saisie de main courante.

Cette méthode crée des interfaces vides avec un message à l'utilisateur l'informant qu'aucun pointeau n'a été scanné et qu'il est obligatoire d'en avoir scanné au moins un pour prendre une photo ou saisir une main courante.

void com.project.rondierprojet.InterfaceRondier.demandeFinRonde () [private]

Méthode créant une boîte de dialogue demandant la confirmation de l'arrêt manuel d'une ronde.

Cette méthode crée une boîte de dialogue demandant à l'utilisateur s'il souhaite arrêter manuellement la ronde en cours ou la continuer.

void com.project.rondierprojet.InterfaceRondier.mettreAJourTexteBoutonAjouterPhoto () [private]

Méthode mettant à jour le bouton d'ajout de photo.

Cette méthode met à jour le bouton d'ajout de photo avec le nom du dernier pointeau scanné.

void com.project.rondierprojet.InterfaceRondier.mettreAJourTexteMainCourante () [private]

Méthode mettant à jour le bouton d'ajout de main courante.

Cette méthode met à jour le bouton d'ajout de main courante avec le nom du dernier pointeau scanné.

String com.project.rondierprojet.InterfaceRondier.obtenirBadgeIDTransmis () [private]

Méthode retournant l'ID du badge de l'agent passé entre les différentes activités sous forme de string.

Cette méthode récupère dans les fichiers partagés entre les différentes activités l'ID du badge de l'agent identifié sous la forme d'une string.

Renvoie:

Si un agent est inscrit, retourne une string contenant l'ID du badge de l'agent. Sinon, retourne null.

String com.project.rondierprojet.InterfaceRondier.obtenirIDPointeauTransmis () [private]

Méthode retournant l'ID du pointeau passé entre les différentes activités.

Cette méthode récupère dans les fichiers partagés entre les différentes activités l'ID du pointeau.

Renvoie:

L'ID du pointeau partagé.

String com.project.rondierprojet.InterfaceRondier.obtenirNomRondeTransmis () [private]

Méthode retournant le nom de la ronde passé entre les différentes activités.

Cette méthode récupère dans les fichiers partagés entre les différentes activités le nom de la ronde en cours.

Renvoie:

Si une ronde est bien inscrite, retourne une string contenant le nom de la ronde. Sinon, retourne null.

void com.project.rondierprojet.InterfaceRondier.onCreate (Bundle *savedInstanceState*) `[protected]`

Méthode appelée au lancement de l'activité.

La méthode onCreate est une méthode appelée par le système Android au lancement de l'activité associée. Cette méthode est ici utilisée pour récupérer l'agent utilisant l'application à partir de la dernière activité ainsi que la ronde en cours aussi récupérée via la dernière activité. par la suite, la classe **InterfaceRondier** crée dynamiquement la liste des pointeaux à scanner, les interfaces de prise de photo et de saisie de main courante (au début vides). La méthode finit par créer la barre de naviguation permettant de changer la vue (liste mains courantes, liste pointeaux ou liste photos) et le bouton donnant la possibilité à l'utilisateur de terminer la ronde manuellement.

Paramètres:

savedInstanceState	Etat antérieur de l'activité.

void com.project.rondierprojet.InterfaceRondier.onDestroy () `[protected]`

Méthode appelée automatiquement lorsque l'activité est détruite par le système Android.

La méthode onDestroy est une méthode appelée par le système Android lorsque ce dernier détruit l'activité (besoin de ressources système, application fermée, etc). Cette méthode définie ici que plus aucune ronde n'est en cours.

void com.project.rondierprojet.InterfaceRondier.onResume () `[protected]`

Méthode appelée automatiquement après l'initialisation de la classe.

La méthode onResume est une méthode appelée par le système Android après l'initialisation de la classe. Cette méthode est ici utilisée pour récupérer le dernier pointeau scannée via **MainActivity**. Si le pointeau scanné est reconnu et que c'est le premier pointeau scannée depuis le lancement de l'application, les interfaces de prise de photo et de saisie de main courante sont actualisées. Le pointeau est ensuite vérifié dans la base de donnée et le passage y est enregistré.

void com.project.rondierprojet.InterfaceRondier.redemarrerApplication () `[private]`

Méthode permettant de redémarrer l'application.

Cette méthode permet de redémarrer l'application et ainsi de supprimer toute modification sur l'interface de l'application lors de son utilisation (Liste de pointeaux, affichage de mains courantes saisies, affichage des photos prises, etc).

void com.project.rondierprojet.InterfaceRondier.supprimerTxtViewAucunPointeauScanne () `[private]`

Méthode supprimant les interfaces créées avec la fonction creerTxtViewAucunPointeauScanne.

Cette méthode supprime les messages d'alertes des interfaces de prise de photo ainsi que de saisie de main courante.

boolean com.project.rondierprojet.InterfaceRondier.verificationPointDansRonde (int *index*) `[private]`

Méthode vérifiant si le dernier pointeau scanné est égale au pointeau indiqué en paramètre.

Cette méthode vérifie si le dernier pointeau scanné est le même que le pointeau passé en paramètre grâce à son index.

Paramètres:

index	L'index dans la liste des pointeaux vérifié par la fonction.

Renvoie:

Si les pointeaux sont identiques, retourne true. Sinon, retourne false.

23.4.1.7 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.MAINACTIVITY

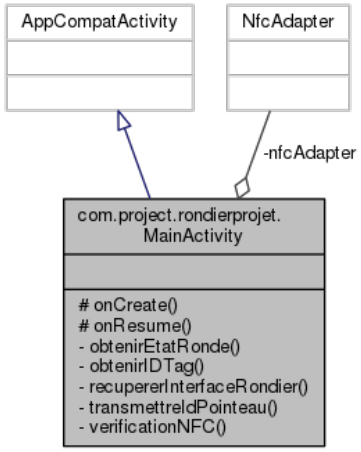
FONCTIONS MEMBRES PROTÉGÉES

void **onCreate** (Bundle savedInstanceState)

Méthode appelée au lancement de l'activité.

void **onResume** ()

Méthode appelée automatiquement après l'initialisation de la classe.



FONCTIONS MEMBRES PRIVÉES

boolean **obtenirEtatRonde** ()

Méthode retournant si une ronde est en cours ou non.

String **obtenirIDTag** (Tag tagMifare)

Méthode retournant l'ID d'un tag en fonction du tag passé en paramètre.

void **recupererInterfaceRondier** ()

Méthode récupérant l'interface de réalisation d'une ronde.

void **transmettreIdPointeau** (String tagID)

Méthode transmettant l'ID du badge de l'agent identifié.

void **verificationNFC** ()

Méthode vérifiant l'état du module NFC.

ATTRIBUTS PRIVÉS

NfcAdapter **nfcAdapter**

DOCUMENTATION DES FONCTIONS MEMBRES

boolean com.project.rondierprojet.MainActivity.obtenirEtatRonde () `[private]`

Méthode retournant si une ronde est en cours ou non.

Cette méthode récupère dans les fichiers partagés entre les différentes activités si une ronde est en cours ou non.

Renvoie:

Retourne true si une ronde est en cours. Sinon, retourne false.

String com.project.rondierprojet.MainActivity.obtenirIDTag (Tag *tagMifare*) `[private]`

Méthode retournant l'ID d'un tag en fonction du tag passé en paramètre.

Cette méthode récupère l'ID du tag passé en paramètre et le convertit ensuite en une string.

Paramètres:

tagMifare	L'objet Tag utilisé par l'API Android pour caractériser un tag MiFare.

Renvoie:

La string contenant l'ID du tag.

void com.project.rondierprojet.MainActivity.onCreate (Bundle *savedInstanceState*) `[protected]`

Méthode appelée au lancement de l'activité.

La méthode onCreate est une méthode appelée par le système Android au lancement de l'activité associée. Cette méthode est ici utilisée pour lancer la vérification des prérequis au bon fonctionnement de l'application.

Paramètres:

savedInstanceState	Etat antérieur de l'activité.

void com.project.rondierprojet.MainActivity.onResume () `[protected]`

Méthode appelée automatiquement après l'initialisation de la classe.

La méthode onResume est une méthode appelée par le système Android après l'initialisation de la classe. Cette méthode est ici utilisée pour gérer l'évènement de découverte de tag par le module NFC. Lorsqu'un tag est découvert, la méthode réoriente l'utilisateur vers la prochaine activité. Si une ronde est en cours, l'utilisateur est réorienté vers l'interface de réalisation de ronde. Dans le cas contraire, l'utilisateur doit choisir la ronde qu'il souhaite effectuer.

void com.project.rondierprojet.MainActivity.recupererInterfaceRondier () `[private]`

Méthode récupérant l'interface de réalisation d'une ronde.

Cette méthode appelle l'activité de réalisation de ronde sans réinitialiser son état. Les modifications précédemment apportées à cette dernière sont alors conservés et l'utilisateur retrouve l'interface comme il l'a laissé en sortant de l'application.

void com.project.rondierprojet.MainActivity.transmettreIdPointeau (String *tagID*) `[private]`

Méthode transmettant l'ID du badge de l'agent identifié.

Cette méthode transmet aux autres activités l'ID sous forme de string de l'agent identifié via les fichiers partagés.

Paramètres:

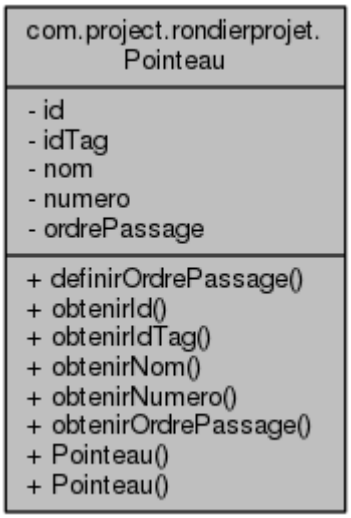
tagID	L'ID du tag MiFare transféré.

void com.project.rondierprojet.MainActivity.verificationNFC () `[private]`

Méthode vérifiant l'état du module NFC.

Cette méthode vérifie si le NFC est disponnible sur le smartphone sur lequel est lancé l'application. Si la fonctionnalité NFC est disponnible, cette méthode vérifie ensuite si cette dernière est bien activée.

23.4.1.8 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.POINTEAU



FONCTIONS MEMBRES PUBLIQUES

void **definirOrdrePassage** (int **ordrePassage**)

Définir un ordre de passage d'un pointeau.

int **obtenirId** ()

Obtenir lI'd d'un pointeau.

String **obtenirIdTag** ()

Obtenir l'idTag d'un pointeau.

String **obtenirNom** ()

Obtenir le nom d'un pointeau.

int **obtenirNumero** ()

Otebnir le numéro d'un pointeau.

int **obtenirOrdrePassage** ()

Obtenir l'ordre de passag d'un pointeau.

Pointeau (int **id**, String **idTag**, String **nom**, int **numero**, int **ordrePassage**)

*Constructeur **Pointeau** Permet de créer un pointeau en fournissant son id, son idTag, son nom, son numero ainsi que son ordre de passage.*

Pointeau (int **id**, String **idTag**, String **nom**, int **numero**)

*Constructeur **Pointeau** Permet de créer un pointeau en fournissant son id, son idTag, son nom ainsi que son numero.*

ATTRIBUTS PRIVÉS

int **id**
String **idTag**
String **nom**
int **numero**
int **ordrePassage**

DOCUMENTATION DES CONSTRUCTEURS ET DESTRUCTEUR

com.project.rondierprojet.Pointeau.Pointeau (int *id*, String *idTag*, String *nom*, int *numero*, int *ordrePassage*)

Constructeur **Pointeau** Permet de créer un pointeau en fournissant son id, son idTag, son nom, son numero ainsi que son ordre de passage.

Paramètres:

id	L'ID du pointeau.
idTag	L'ID du tag MiFare associé au pointeau.
nom	Le nom du pointeau.
numero	Le numéro du pointeau.
ordrePassage	L'ordre de passage du pointeau.

com.project.rondierprojet.Pointeau.Pointeau (int *id*, String *idTag*, String *nom*, int *numero*)

Constructeur **Pointeau** Permet de créer un pointeau en fournissant son id, son idTag, son nom ainsi que son numero.

Paramètres:

id	L'ID du pointeau.
idTag	L'ID du tag MiFare associé au pointeau.
nom	Le nom du pointeau.
numero	Le numéro du pointeau.

DOCUMENTATION DES FONCTIONS MEMBRES

void com.project.rondierprojet.Pointeau.definirOrdrePassage (int *ordrePassage*)

Définir un ordre de passage d'un pointeau.

Permet de définir un ordre de passage associé à un pointeau.

Paramètres:

ordrePassage	L'ordre de passage associé au pointeau.

int com.project.rondierprojet.Pointeau.obtenirId ()

Obtenir l''id d'un pointeau.

Permet de obtenir l''id associé à un pointeau.

Renvoie:

L'ID du pointeau.

String com.project.rondierprojet.Pointeau.obtenirIdTag ()

Obtenir l'idTag d'un pointeau.

Permet d'obtenir l'idTag associé à un pointeau.

Renvoie:

L'ID du tag associé au pointeau.

String com.project.rondierprojet.Pointeau.obtenirNom ()

Obtenir le nom d'un pointeau.

Permet de obtenir le nom associé à un pointeau.

Renvoie:

Le nom du pointeau.

int com.project.rondierprojet.Pointeau.obtenirNumero ()

Otebnir le numéro d'un pointeau.

Permet d'obtenir le numéro associé à un pointeau.

Renvoie:

Le numéro associé au pointeau.

int com.project.rondierprojet.Pointeau.obtenirOrdrePassage ()

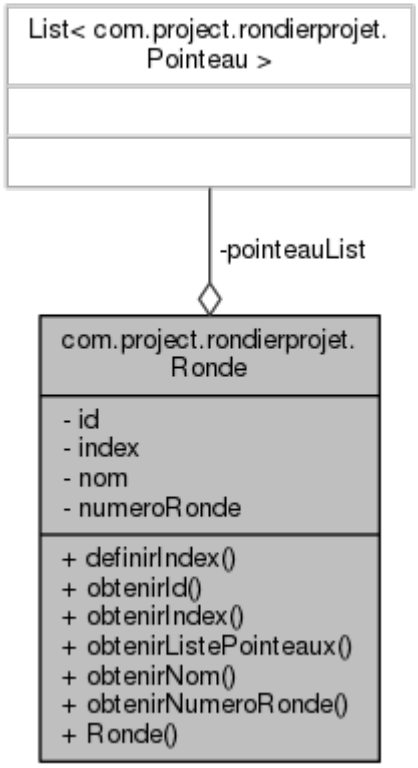
Obtenir l'ordre de passag d'un pointeau.

Permet d'obtenir l'ordre de passage associé à un pointeau.

Renvoie:

L'ordre de passage du pointeau.

23.4.1.9 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.RONDE



FONCTIONS MEMBRES PUBLIQUES

void **definirIndex** (int **index**)
Définir l'index d'une ronde.

int **obtenirId** ()
Obtenir l'ID d'une ronde.

int **obtenirIndex** ()
Obtenir l'index d'une ronde.

List< **Pointeau** > **obtenirListePointeaux** ()
Obtenir la liste des pointeaux d'une ronde.

String **obtenirNom** ()
Obtenir le nom d'une ronde.

int **obtenirNumeroRonde** ()
Obtenir le numero d'une ronde.

Ronde (int **id**, String **nom**, List< **Pointeau** > **pointeauList**, int **numeroRonde**)
*Constructeur **Ronde** Permet de créer une ronde en fournissant son id et son nom.*

ATTRIBUTS PRIVÉS

int **id**
int **index** = 0
String **nom**
int **numeroRonde**
List< **Pointeau** > **pointeauList**

DOCUMENTATION DES CONSTRUCTEURS ET DESTRUCTEUR

com.project.rondierprojet.Ronde.Ronde (int *id*, String *nom*, List< Pointeau > *pointeauList*, int *numeroRonde*)

Constructeur **Ronde** Permet de créer une ronde en fournissant son id et son nom.

Paramètres:

id	L'ID de la ronde.
nom	Le nom de la ronde.

DOCUMENTATION DES FONCTIONS MEMBRES

void com.project.rondierprojet.Ronde.definirIndex (int *index*)

Définir l'index d'une ronde.

Permet de définir l'index d'une ronde.

Paramètres:

index	L'index présent dans la ronde.

int com.project.rondierprojet.Ronde.obtenirId ()

Obtenir l'ID d'une ronde.

Permet d'obtenir l'ID attribué à une ronde.

Renvoie:

L'ID de la ronde.

int com.project.rondierprojet.Ronde.obtenirIndex ()

Obtenir l'index d'une ronde.

Permet d'obtenir l'index d'une ronde.

Renvoie:

L'index de la ronde.

List<Pointeau> com.project.rondierprojet.Ronde.obtenirListePointeaux ()

Obtenir la liste des pointeaux d'une ronde.

Permet d'obtenir une liste de pointeaux que comporte une ronde.

Renvoie:

La liste de pointeaux associés à cette ronde.

String com.project.rondierprojet.Ronde.obtenirNom ()

Obtenir le nom d'une ronde.

Permet d'obtenir le nom attribué à une ronde.

Renvoie:

Le nom de la ronde.

int com.project.rondierprojet.Ronde.obtenirNumeroRonde ()

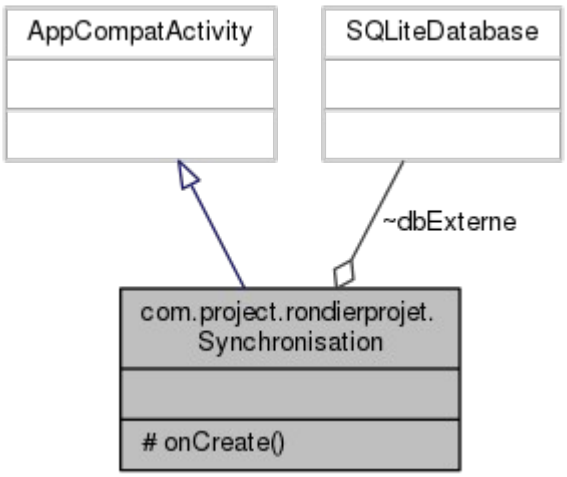
Obtenir le numero d'une ronde.

Permet d'obtenir le numero attribué à une ronde.

Renvoie:

Le numéro de la ronde.

23.4.1.10 - RÉFÉRENCE DE LA CLASSE COM.PROJECT.RONDIERPROJET.SYNCHRONISATION



FONCTIONS MEMBRES PROTÉGÉES

void **onCreate** (Bundle savedInstanceState)

Méthode appelée automatiquement après l'instanciation de la classe.

DOCUMENTATION DES FONCTIONS MEMBRES

void com.project.rondierprojet.Synchronisation.onCreate (Bundle *savedInstanceState*)

`[protected]`

Méthode appelée automatiquement après l'instanciation de la classe.

La méthode onCreate est une méthode appelée par le système Android à l'instanciation de la classe. Cette méthode est ici utilisée pour synchroniser les tables entre les différentes base de données. Cette synchronisation ne se fait que si l'appel de la classe se fait avec l'intent "android.intent.action.SYNC".

Paramètres:

savedInstanceState	Etat antérieur de l'activité.

23.4.2 - Documentation des fichiers

23.4.2.1 - RÉFÉRENCE DU FICHIER COM/PROJECT/RONDIERPROJET/AGENT.JAVA

Définition de la classe de gestion d'un agent.

CLASSES

class **com.project.rondierprojet.Agent**

PAQUETAGES

package **com.project.rondierprojet**

DESCRIPTION DÉTAILLÉE

Définition de la classe de gestion d'un agent.

Classe modélisant un agent de sécurité

23.4.2.2 - RÉFÉRENCE DU FICHIER COM/PROJECT/RONDIERPROJET/CAMERA.JAVA

Définition de la classe de gestion de la caméra.

CLASSES

class **com.project.rondierprojet.Camera**

PAQUETAGES

package **com.project.rondierprojet**

DESCRIPTION DÉTAILLÉE
Définition de la classe de gestion de la caméra.

Classe modélisant la caméra créée à partir de l'API de yaraki (`https://github.com/google/cameraview`) Cette classe permet de créer l'aperçu d'une caméra ainsi que d'enregistrer une photo prise et ce en prenant compte automatiquement de la version d'Android utilisée.

23.4.2.3 - RÉFÉRENCE DU FICHIER COM/PROJECT/RONDIERPROJET/CHOIXRONDE.JAVA

Définition de la classe choix de ronde.

CLASSES
class **com.project.rondierprojet.ChoixRonde**

PAQUETAGES
package **com.project.rondierprojet**

DESCRIPTION DÉTAILLÉE
Définition de la classe choix de ronde.

Classe affichant l'interface responsable de la liste et du choix de la liste attribuée à un agent.

23.4.2.4 - RÉFÉRENCE DU FICHIER COM/PROJECT/RONDIERPROJET/GESTIONBDD.JAVA

Définition de la classe de gestion de la base de données.

CLASSES
class **com.project.rondierprojet.GestionBDD**

PAQUETAGES
package **com.project.rondierprojet**

DESCRIPTION DÉTAILLÉE
Définition de la classe de gestion de la base de données.

Classe gérant les requête SQL envoyées sur le fichier sqlite. Cette classe permet d'injecter de nouvelles données ainsi que de récupérer des données précédemment écrites.

23.4.2.5 - RÉFÉRENCE DU FICHIER COM/PROJECT/RONDIERPROJET/INTERFACERONDIER.JAVA

Définition de la classe correspondant à l'interface de réalisation d'une ronde.

CLASSES
class **com.project.rondierprojet.InterfaceRondier**

PAQUETAGES
package **com.project.rondierprojet**

DESCRIPTION DÉTAILLÉE
Définition de la classe correspondant à l'interface de réalisation d'une ronde.

Classe gérant l'interface de réalisation d'une ronde.

23.4.2.6 - RÉFÉRENCE DU FICHIER COM/PROJECT/RONDIERPROJET/MAINACTIVITY.JAVA

Définition de la classe lancée au démarrage de l'application.

CLASSES
class **com.project.rondierprojet.MainActivity**

PAQUETAGES
package **com.project.rondierprojet**

DESCRIPTION DÉTAILLÉE
Définition de la classe lancée au démarrage de l'application.

Classe gérant le module NFC et la récupération des informations des badges via ce dernier.

23.4.2.7 - RÉFÉRENCE DU FICHIER COM/PROJECT/RONDIERPROJET/POINTEAU.JAVA

Définition de la classe de gestion d'un pointeau.

CLASSES
class **com.project.rondierprojet.Pointeau**

PAQUETAGES
package **com.project.rondierprojet**

DESCRIPTION DÉTAILLÉE
Définition de la classe de gestion d'un pointeau.

Classe modélisant un pointeau.

23.4.2.8 - RÉFÉRENCE DU FICHIER COM/PROJECT/RONDIERPROJET/RONDE.JAVA

Définition de la classe de gestion de ronde.

CLASSES
class **com.project.rondierprojet.Ronde**

PAQUETAGES
package **com.project.rondierprojet**

DESCRIPTION DÉTAILLÉE
Définition de la classe de gestion de ronde.

Classe modélisant une ronde.

23.4.2.9 - RÉFÉRENCE DU FICHIER COM/PROJECT/RONDIERPROJET/SYNCHRONISATION.JAVA

Définition de la classe de synchronisation de la base de données.

CLASSES

class **com.project.rondierprojet.Synchronisation**

PAQUETAGES

package **com.project.rondierprojet**

DESCRIPTION DÉTAILLÉE

Définition de la classe de synchronisation de la base de données.

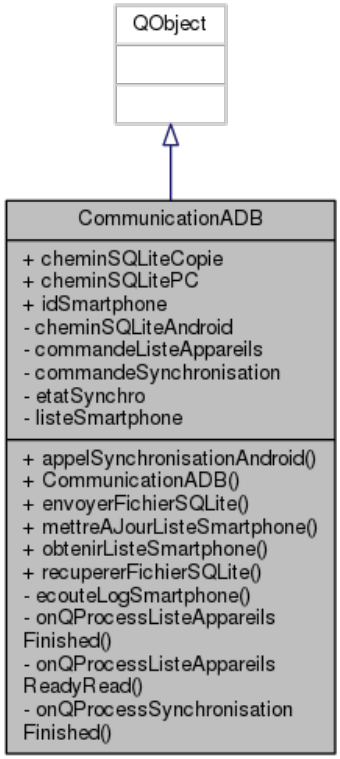
Classe synchronisant la base de données avec le poste de supervision. Cette classe n'est jamais appelée dans l'application elle-même. Elle n'a pour but que d'être appelée via adb, commande à faire sur un ordinateur : adb shell am start -n "com.project.rondierprojet/com.project.rondierprojet.Synchronisation" -a android.intent.action.SYNC. La classe Synchronisation utilise la base de données du téléphone présente dans l'application et gérée par GestionBDD ainsi que la base de données envoyée par le poste de supervision dans /sdcard/Android/data/com.project.rondierprojet/files/Documents.

23.5 - Doxygen de la synchronisation de l'application supervision

23.5.1 - Documentation des classes

23.5.1.1 - RÉFÉRENCE DE LA CLASSE COMMUNICATIONADB

`#include <communicationadb.h>`
Graphe de collaboration de CommunicationADB:



23.5.1.2 - FONCTIONS MEMBRES PUBLIQUES

void **appelSynchronisationAndroid** ()

CommunicationADB::appelSynchronisationAndroid *Cette méthode d'appeler la classe Synchronisation"du smartphone sélectionné dans la liste déroulante.*

CommunicationADB ()

CommunicationADB::CommunicationADB *Constructeur de la classe Communication ADB. Ce constructeur s'occupe de déclarer les QProcess et de connecter les signaux de ces derniers aux slots de la classe.*

void **envoyerFichierSQLite** ()

CommunicationADB::envoyerFichierSQLite *Cette méthode permet d'envoyer le fichier SQLite créé pour la copie des données entre le poste de supervision et le smartphone sur le smartphone.*

void **mettreAJourListeSmartphone** ()

CommunicationADB::mettreAJourListeSmartphone *Cette méthode permet de relancer la détection des smartphones connectés.*

QList< QString > **obtenirListeSmartphone** ()

CommunicationADB::obtenirListeSmartphone *Cette méthode permet de retourner la liste des smartphones connectés au PC.*

void **recupererFichierSQLite** ()

CommunicationADB::recupererFichierSQLite *Cette méthode permet de récupérer le fichier SQLite créé pour la copie des données entre le poste de supervision et le smartphone sur le poste de supervision.*

23.5.1.3 - ATTRIBUTS PUBLICS

QString **cheminSQLiteCopie**
QString **cheminSQLitePC**
QString **idSmartphone**

23.5.1.4 - CONNECTEURS PRIVÉS

void **onQProcessListeAppareilsFinished** ()

CommunicationADB::onQProcessListeAppareilsFinished *Ce slot permet de terminer le QProcess de détection des smartphones connectés au poste de supervision une fois que se dernier a fini de s'éxecuter.*

void **onQProcessListeAppareilsReadyRead** ()

CommunicationADB::onQProcessListeAppareilsReadyRead *Ce slot permet de traiter le résultat de la détection des smartphones connectés faite via le QProcess. J'utilise ici une expression régulière pour ne récupérer que l'ID du smartphone.*

void **onQProcessSynchronisationFinished** ()

CommunicationADB::onQProcessSynchronisationFinished *Ce slot permet de terminer le QProcess qui s'occupe d'appeler la classe Synchronisation"du smartphone.*

23.5.1.5 - ATTRIBUTS PRIVÉS

QString **cheminSQLiteAndroid**
QProcess * **commandeListeAppareils**
QProcess * **commandeSynchronisation**
int **etatSynchro**
QList< QString > **listeSmartphone**

23.5.1.6 - DOCUMENTATION DES CONSTRUCTEURS ET DESTRUCTEUR

COMMUNICATIONADB::COMMUNICATIONADB ()

CommunicationADB::CommunicationADB Constructeur de la classe Communication ADB. Ce constructeur s'occupe de déclarer les QProcess et de connecter les signaux de ces derniers aux slots de la classe.

23.5.1.7 - DOCUMENTATION DES FONCTIONS MEMBRES

VOID COMMUNICATIONADB::APPELSYNCHRONISATIONANDROID ()

CommunicationADB::appelSynchronisationAndroid Cette méthode d'appeler la classe "Synchronisation" du smartphone sélectionné dans la liste déroulante.

VOID COMMUNICATIONADB::ENVOYERFICHIERSQLITE ()

CommunicationADB::envoyerFichierSQLite Cette méthode permet d'envoyer le fichier SQLite créé pour la copie des données entre le poste de supervision et le smartphone sur le smartphone.

VOID COMMUNICATIONADB::METTREAJOURLISTESMARTPHONE ()

CommunicationADB::mettreAJourListeSmartphone Cette méthode permet de relancer la détection des smartphones connectés.

QLIST< QSTRING > COMMUNICATIONADB::OBTENIRLISTESMARTPHONE ()

CommunicationADB::obtenirListeSmartphone Cette méthode permet de retourner la liste des smartphones connectés au PC.

Renvoie:

La liste des smartphones.

VOID COMMUNICATIONADB::ONQPROCESSLISTEAPPAREILSFINISHED () `[PRIVATE], [SLOT]`

CommunicationADB::onQProcessListeAppareilsFinished Ce slot permet de terminer le QProcess de détection des smartphones connectés au poste de supervision une fois que se dernier a fini de s'éxecuter.

VOID COMMUNICATIONADB::ONQPROCESSLISTEAPPAREILSREADYREAD () `[PRIVATE], [SLOT]`

CommunicationADB::onQProcessListeAppareilsReadyRead Ce slot permet de traiter le résultat de la détection des smartphones connectés faite via le QProcess. J'utilise ici une expression régulière pour ne récupérer que l'ID du smartphone.

VOID COMMUNICATIONADB::ONQPROCESSSYNCHRONISATIONFINISHED () `[PRIVATE], [SLOT]`

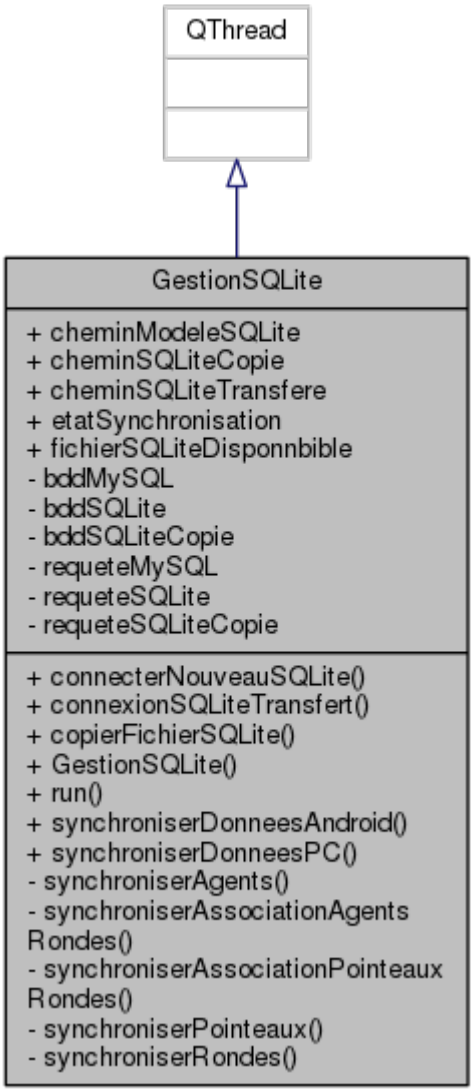
CommunicationADB::onQProcessSynchronisationFinished Ce slot permet de terminer le QProcess qui s'occupe d'appeler la classe "Synchronisation" du smartphone.

VOID COMMUNICATIONADB::RECUPERERFICHIERSQLITE ()

CommunicationADB::recupererFichierSQLite Cette méthode permet de récupérer le fichier SQLite créé pour la copie des données entre le poste de supervision et le smartphone sur le poste de supervision.

23.5.2 - Référence de la classe GestionSQLite

`#include <gestionsqlite.h>`
Graphe de collaboration de GestionSQLite:



23.5.2.1 - SIGNAUX

void **nouveauLog** (QString texte, bool gras)

23.5.2.2 - FONCTIONS MEMBRES PUBLIQUES

void **connecterNouveauSQLite** ()

GestionSQLite::connecterNouveauSQLite *Cette méthode permet, une fois appelée, de connecter la base de données qui a été récupérée sur le smartphone.*

void **connexionSQLiteTransfert** ()

GestionSQLite::connexionSQLiteTransfert *Cette méthode permet, une fois appelée, de connecter la base de données qui sera envoyée sur le smartphone.*

bool **copierFichierSQLite** ()

GestionSQLite::copierFichierSQLite *Cette méthode permet de supprimer les fichiers issues d'une ancienne synchronisation et de copier le fichier modèle pour en avoir une version propre.*

GestionSQLite ()

GestionSQLite::GestionSQLite *Constructeur de la classe* ***GestionSQLite****. Ce constructeur est responsable de la connexion des différentes base de données et de la déclaration des queries.*

void **run** ()

GestionSQLite::run *Cette méthode est une surcharge de la fonction* ***run()*** *de QThread. Elle permet qu'une fois que le thread de* ***GestionSQLite*** *est lancé, les données se synchronisent automatiquement en fonction de l'avancement de la synchronisation avec le smartphone.*

void **synchroniserDonneesAndroid** ()

GestionSQLite::synchroniserDonneesAndroid *Cette méthode permet de synchroniser les données réçus du smartphone dans la base de données du poste de supervision. Les données ici copiées sont celles de la table MainCourante et HistoriquePointeau. Une manipulation spécifique a dû être faite car les mains courantes possèdent comme base d'association l'ID de l'historique du pointeau associé. Il a donc été nécessaire de traiter les différents IDs de façon à ce qu'ils ne se chevauchent pas avec ceux déjà existant.*

void **synchroniserDonneesPC** ()

GestionSQLite::synchroniserDonneesPC *Cette méthode permet de lancer la synchronisation des Agents, Pointeaux, Rondes, Associations Agents/Rondes et Assoications Pointeaux/Rondes vers le fichier qui sera envoyé sur le smartphone.*

23.5.2.3 - ATTRIBUTS PUBLICS

QString **cheminModeleSQLite**
QString **cheminSQLiteCopie**
QString **cheminSQLiteTransfere**
int **etatSynchronisation**
bool **fichierSQLiteDisponnbible**

23.5.2.4 - FONCTIONS MEMBRES PRIVÉES

void **synchroniserAgents** ()

GestionSQLite::synchroniserAgents *Cette méthode permet de synchroniser les données concernant la table Agents entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.*

void **synchroniserAssociationAgentsRondes** ()

GestionSQLite::synchroniserAssociationAgentsRondes *Cette méthode permet de synchroniser les données concernant la table AssociationAgentsRondes entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.*

void **synchroniserAssociationPointeauxRondes** ()

GestionSQLite::synchroniserAssociationPointeauxRondes *Cette méthode permet de synchroniser les données concernant la table AssociationPointeauxRondes entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.*

void **synchroniserPointeaux** ()

GestionSQLite::synchroniserPointeaux *Cette méthode permet de synchroniser les données concernant la table Pointeaux entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.*

void **synchroniserRondes** ()

GestionSQLite::synchroniserRondes *Cette méthode permet de synchroniser les données concernant la table Rondes entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.*

23.5.2.5 - ATTRIBUTS PRIVÉS

QSqlDatabase **bddMySQL**
QSqlDatabase **bddSQLite**
QSqlDatabase **bddSQLiteCopie**
QSqlQuery * **requeteMySQL**
QSqlQuery * **requeteSQLite**
QSqlQuery * **requeteSQLiteCopie**

23.5.2.6 - DOCUMENTATION DES CONSTRUCTEURS ET DESTRUCTEUR

GESTIONSQLITE::GESTIONSQLITE ()

GestionSQLite::GestionSQLite Constructeur de la classe **GestionSQLite**. Ce constructeur est responsable de la connexion des différentes base de données et de la déclaration des queries.

23.5.2.7 - DOCUMENTATION DES FONCTIONS MEMBRES

VOID GESTIONSQLITE::CONNECTERNOUVEAUSQLITE ()

GestionSQLite::connecterNouveauSQLite Cette méthode permet, une fois appelée, de connecter la base de données qui a été récupérée sur le smartphone.

VOID GESTIONSQLITE::CONNEXIONSQLITETRANSFERT ()

GestionSQLite::connexionSQLiteTransfert Cette méthode permet, une fois appelée, de connecter la base de données qui sera envoyée sur le smartphone.

BOOL GESTIONSQLITE::COPIERFICHIERSQLITE ()

GestionSQLite::copierFichierSQLite Cette méthode permet de supprimer les fichiers issues d'une ancienne synchronisation et de copier le fichier modèle pour en avoir une version propre.

Renvoie:

Retourne true si le fichier a bien été créé, sinon retourne false.

VOID GESTIONSQLITE::NOUVEAULOG (QSTRING *TEXTE*, BOOL *GRAS*) [SIGNAL]

VOID GESTIONSQLITE::RUN ()

GestionSQLite::run Cette méthode est une surcharge de la fonction **run()** de QThread. Elle permet qu'une fois que le thread de **GestionSQLite** est lancé, les données se synchronisent automatiquement en fonction de l'avancement de la synchronisation avec le smartphone.

VOID GESTIONSQLITE::SYNCHRONISERAGENTS () [PRIVATE]

GestionSQLite::synchroniserAgents Cette méthode permet de synchroniser les données concernant la table Agents entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.

VOID GESTIONSQLITE::SYNCHRONISERASSOCIATIONAGENTSRONDES () [PRIVATE]

GestionSQLite::synchroniserAssociationAgentsRondes Cette méthode permet de synchroniser les données concernant la table AssociationAgentsRondes entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.

VOID GESTIONSQLITE::SYNCHRONISERASSOCIATIONPOINTEAUXRONDES () [PRIVATE]

GestionSQLite::synchroniserAssociationPointeauxRondes Cette méthode permet de synchroniser les données concernant la table AssociationPointeauxRondes entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.

VOID GESTIONSQLITE::SYNCHRONISERDONNEESANDROID ()

GestionSQLite::synchroniserDonneesAndroid Cette méthode permet de synchroniser les données réçus du smartphone dans la base de données du poste de supervision. Les données ici copiées sont celles de la table MainCourante et HistoriquePointeau. Une manipulation spécifique a dû être faite car les mains courantes possèdent comme base d'association l'ID de l'historique du pointeau associé. Il a donc été nécessaire de traiter les différents IDs de façon à ce qu'ils ne se chevauchent pas avec ceux déjà existant.

VOID GESTIONSQLITE::SYNCHRONISERDONNEESPC ()

GestionSQLite::synchroniserDonneesPC Cette méthode permet de lancer la synchronisation des Agents, Pointeaux, Rondes, Associations Agents/Rondes et Assoications Pointeaux/Rondes vers le fichier qui sera envoyé sur le smartphone.

VOID GESTIONSQLITE::SYNCHRONISERPOINTEAUX () [PRIVATE]

GestionSQLite::synchroniserPointeaux Cette méthode permet de synchroniser les données concernant la table Pointeaux entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.

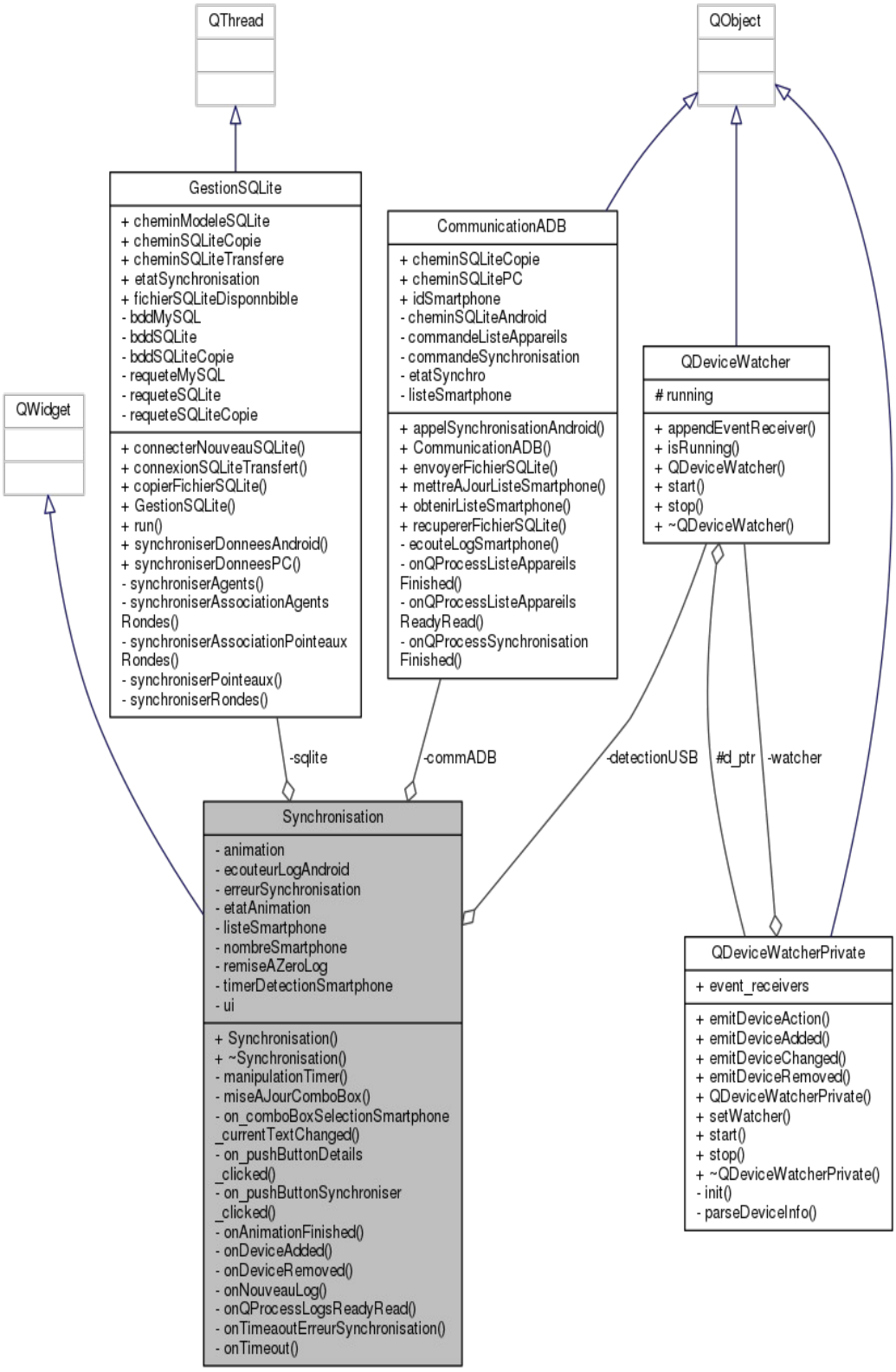
VOID GESTIONSQLITE::SYNCHRONISERRONDES () [PRIVATE]

GestionSQLite::synchroniserRondes Cette méthode permet de synchroniser les données concernant la table Rondes entre la base de données du poste de supervision et le fichier qui sera transféré au smartphone.

23.5.3 - Référence de la classe Synchronisation

`#include <synchronisation.h>`

Graphe de collaboration de Synchronisation:



23.5.3.1 - FONCTIONS MEMBRES PUBLIQUES

Synchronisation (QWidget *parent=0)

Synchronisation::Synchronisation *Constructeur de la classe* ***Synchronisation****. Le constructeur s'occupe ici de lancer toutes les classes secondaires, la détection des smartphones ainsi que l'initialisation de l'interface.*

~Synchronisation ()

Synchronisation::~Synchronisation *Destructeur de la classe* ***Synchronisation****. Le destructeur permet de stoper tous les processus lancés par l'application, permettant de ne pas avoir de message d'erreur à la fermeture de l'application supervision.*

23.5.3.2 - CONNECTEURS PRIVÉS

void **on_comboBoxSelectionSmartphone_currentTextChanged** (const QString &arg1)

Synchronisation::on_comboBoxSelectionSmartphone_currentTextChanged *Ce slot permet d'activer ou de désactiver le bouton de synchronisation de l'interface si un smartphone est sélectionné ou non.*

void **on_pushButtonDetails_clicked** ()

Synchronisation::on_pushButtonDetails_clicked *Ce slot permet d'afficher ou de cacher les détails de la synchronisation en fonction de son état précédent.*

void **on_pushButtonSynchroniser_clicked** ()

Synchronisation::on_pushButtonSynchroniser_clicked *Ce slot permet, quand le bouton de synchronisation est pressé, de préparer les données de synchronisation et de les envoyer sur le smartphone.*

void **onAnimationFinished** ()

Synchronisation::onAnimationFinished *Ce slot permet, lorsque l'animation en cours de la progress bar est terminé, de lancer l'animation suivante. Cela permet une progression de la bar continue et fluide.*

void **onDeviceAdded** ()

Synchronisation::onDeviceAdded *Ce slot permet de lancer la détection de smartphone lorsqu'un périphérique est branché en USB sur le poste de supervision.*

void **onDeviceRemoved** ()

Synchronisation::onDeviceRemoved *Ce slot permet de mettre à jour la liste des smartphones lorsqu'un périphérique USB est débranché du poste de supervision.*

void **onNouveauLog** (QString texte, bool gras)

Synchronisation::onNouveauLog *Ce slot permet de récupérer les messages de la classe* ***GestionSQLite*** *et de les afficher dans les détails de l'interface.*

void **onQProcessLogsReadyRead** ()

Synchronisation::onQProcessLogsReadyRead *Ce slot permet de lire les logs envoyés par le smartphone et donc de détecter lorsque ce dernier a terminé sa partie de la synchronisation. Le slot lance ensuite le transfert des données du fichier récupéré vers la base de données du poste de supervision.*

void **onTimeaoutErreurSynchronisation** ()

Synchronisation::onTimeaoutErreurSynchronisation *Ce slot permet d'identifier si le smartphone en cours de synchronisation a eu un problème. Si le timer a le temps de se terminer et donc de lancer ce slot, la synchronisation en cours est considérée comme échouée.*

void **onTimeout** ()

Synchronisation::onTimeout *Ce slot permet de lancer une détection des smartphones connectés dans un rapport cyclique ce qui permet de retrouver un téléphone qui ne serait pas détecté à la première détection.*

23.5.3.3 - FONCTIONS MEMBRES PRIVÉES

void **manipulationTimer** ()

Synchronisation::manipulationTimer *Cette méthode permet d'arrêter ou de relancer la détection par cycle des smartphones connectés s'il y a au moins un smartphone branché.*

void **miseAJourComboBox** ()

Synchronisation::miseAJourComboBox *Cette méthode permet de mettre à jour la liste déroulante de liste des smartphones de l'interface lorsqu'un smartphone est ajouté ou enlevé. Cette manipulation permet de ne pas remettre à zéro la sélection lorsque la liste des smartphones connectés est modifiée.*

23.5.3.4 - ATTRIBUTS PRIVÉS

QPropertyAnimation * **animation**
CommunicationADB * commADB
QDeviceWatcher * detectionUSB
QProcess * **ecouteurLogAndroid**
QTimer * **erreurSynchronisation**
int **etatAnimation**
QList< QString > **listeSmartphone**
int **nombreSmartphone**
QProcess * **remiseAZeroLog**
GestionSQLite * sqlite
QTimer * **timerDetectionSmartphone**
Ui::Synchronisation * **ui**

23.5.3.5 - DOCUMENTATION DES CONSTRUCTEURS ET DESTRUCTEUR

SYNCHRONISATION::SYNCHRONISATION (QWIDGET * *PARENT* = 0) [EXPLICIT]

Synchronisation::Synchronisation Constructeur de la classe **Synchronisation**. Le constructeur s'occupe ici de lancer toutes les classes secondaires, la détection des smartphones ainsi que l'initialisation de l'interface.

SYNCHRONISATION::~SYNCHRONISATION ()

Synchronisation::~Synchronisation Destructeur de la classe **Synchronisation**. Le destructeur permet de stoper tous les processus lancés par l'application, permettant de ne pas avoir de message d'erreur à la fermeture de l'application supervision.

23.5.3.6 - DOCUMENTATION DES FONCTIONS MEMBRES

VOID SYNCHRONISATION::MANIPULATIONTIMER () [PRIVATE]

Synchronisation::manipulationTimer Cette méthode permet d'arrêter ou de relancer la détection par cycle des smartphones connectés s'il y a au moins un smartphone branché.

VOID SYNCHRONISATION::MISEAJOURCOMBOBOX () [PRIVATE]

Synchronisation::miseAJourComboBox Cette méthode permet de mettre à jour la liste déroulante de liste des smartphones de l'interface lorsqu'un smartphone est ajouté ou enlevé. Cette manipulation permet de ne pas remettre à zéro la sélection lorsque la liste des smartphones connectés est modifiée.

VOID SYNCHRONISATION::ON_COMBOBOXSELECTIONSMARTPHONE_CURRENTTEXTCHANGED (CONST QSTRING & *ARG1*) [PRIVATE], [SLOT]

Synchronisation::on_comboBoxSelectionSmartphone_currentTextChanged Ce slot permet d'activer ou de désactiver le bouton de synchronisation de l'interface si un smartphone est sélectionné ou non.

VOID SYNCHRONISATION::ON_PUSHBUTTONDETAILS_CLICKED () [PRIVATE], [SLOT]

Synchronisation::on_pushButtonDetails_clicked Ce slot permet d'afficher ou de cacher les détails de la synchronisation en fonction de son état précédent.

VOID SYNCHRONISATION::ON_PUSHBUTTONSYNCHRONISER_CLICKED () [PRIVATE], [SLOT]

Synchronisation::on_pushButtonSynchroniser_clicked Ce slot permet, quand le bouton de synchronisation est pressé, de préparer les données de synchronisation et de les envoyer sur le smartphone.

VOID SYNCHRONISATION::ONANIMATIONFINISHED () [PRIVATE], [SLOT]

Synchronisation::onAnimationFinished Ce slot permet, lorsque l'animation en cours de la progress bar est terminé, de lancer l'animation suivante. Cela permet une progression de la bar continue et fluide.

VOID SYNCHRONISATION::ONDEVICEADDED () [PRIVATE], [SLOT]

Synchronisation::onDeviceAdded Ce slot permet de lancer la détection de smartphone lorsqu'un périphérique est branché en USB sur le poste de supervision.

VOID SYNCHRONISATION::ONDEVICEREMOVED () [PRIVATE], [SLOT]

Synchronisation::onDeviceRemoved Ce slot permet de mettre à jour la liste des smartphones lorsqu'un périphérique USB est débranché du poste de supervision.

VOID SYNCHRONISATION::ONNOUVEAULOG (QSTRING *TEXTE*) [PRIVATE], [SLOT]

Synchronisation::onNouveauLog Ce slot permet de récupérer les messages de la classe **GestionSQLite** et de les afficher dans les détails de l'interface.

Paramètres:

texte	Texte comportant le message envoyé par GestionSQLite à afficher dans les détails.

VOID SYNCHRONISATION::ONQPROCESSLOGSREADYREAD () [PRIVATE], [SLOT]

Synchronisation::onQProcessLogsReadyRead Ce slot permet de lire les logs envoyés par le smartphone et donc de détecter lorsque ce dernier a terminé sa partie de la synchronisation. Le slot lance ensuite le transfert des données du fichier récupéré vers la base de données du poste de supervision.

VOID SYNCHRONISATION::ONTIMEAOUTERREURSYNCHRONISATION () [PRIVATE], [SLOT]

Synchronisation::onTimeaoutErreurSynchronisation Ce slot permet d'identifier si le smartphone en cours de synchronisation a eu un problème. Si le timer a le temps de se terminer et donc de lancer ce slot, la synchronisation en cours est considérée comme échouée.

VOID SYNCHRONISATION::ONTIMEOUT () [PRIVATE], [SLOT]

Synchronisation::onTimeout Ce slot permet de lancer une détection des smartphones connectés dans un rapport cyclique ce qui permet de retrouver un téléphone qui ne serait pas détecté à la première détection.