GUIDE
ÉTAPE 3 : ENREGISTREMENT EN LOCAL PAR SÉRIALISATION OU SQLITE

Contenu

1. Enregistrement en local par sérialisation

Pour le moment, lorsque l'application est fermée, les données saisies sont perdues. Il existe plusieurs méthodes pour enregistrer les données afin de les réutiliser ultérieurement. On parle alors de "persistance" des données : la sérialisation, la base de données locale, l'enregistrement distant (par exemple dans une base de données distante type MySQL ou MariaDB). Toutes ces méthodes vont être abordées.

Dans un premier temps, vous allez découvrir la sérialisation, qui consiste à enregistrer directement le contenu d'objets, au format binaire, donc dans un fichier. Le fichier sera directement dans le téléphone.

1A. La classe Serializer

Il est préférable de ranger les classes techniques dans un package séparé : sous Android Studio, dans le package général ("com.example.coach") et donc au même niveau que les 3 autres packages, créez le package "outils". Vous avez maintenant 4 packages.

Vous allez utiliser une classe de sérialisation qui a déjà été écrite, et dont on va détailler le contenu. Prenez le fichier Serializer.java qui se trouve dans les "sources Android" mis à votre disposition, et copiez-le dans le dossier "Coach\app\src\main\java\com\example\coach\outils\" de votre projet (en passant par l'explorateur de fichier et non par Android Studio). Si vous regardez dans l'arborescence du projet, sous Android Studio, vous devriez voir maintenant ce nouveau fichier, dans le package outils.

Double-cliquez sur Serializer pour l'ouvrir et voir le code. Normalement, si vous avez respecté toutes les étapes de création du projet, la première ligne de code qui contient le nom du package, doit être correcte. Si la ligne est soulignée en rouge, alors il faut corriger le nom du package.

Cette classe ne contient que 2 méthodes statiques :

- serialize : permet d'enregistrer l'objet reçu en paramètre, dans le fichier dont le nom est aussi passé en paramètre, au niveau du contexte passé aussi en paramètre (car on a besoin d'un contexte pour sérialiser) ;
- deSerialize : retourne l'objet qui a été sérialisé dans le fichier dont le nom est passé en paramètre, au niveau du contexte passé en paramètre.

Prenez le temps de regarder plus en détail le contenu des 2 méthodes pour mieux comprendre le fonctionnement.

Cette classe pourra éventuellement vous servir pour d'autres projets.

1B. Sauvegarde de l'objet

Vous l'avez compris : on ne peut sérialiser que des objets. On va donc sérialiser le profil que l'on a créé grâce à la classe métier Profil.

Attention cependant, pour qu'un objet puisse être sérialisé, il faut que la classe correspondante soit sérialisable : donc, au niveau de la classe Profil, faites en sorte qu'elle soit sérialisable en la faisant implémenter l'interface Serializable.

À partir de là, c'est bien sûr encore une fois le contrôleur qui va s'occuper de tout gérer.

Dans la classe Controle, commencez par déclarer en static une propriété nomFic de type String et initialisez-là directement avec "saveprofil". Ce sera le nom du fichier binaire qui va mémoriser la sérialisation du profil. Évidemment vous pourriez mettre le nom que vous voulez.

À quel moment sauvegarder le profil ? Tout simplement au moment où il est créé. Dans la classe Controle, méthode creerProfil, après avoir créé le profil, vous allez le sérialiser en appelant la méthode statique serialize de la classe outils Serializer en mettant les bons paramètres… et c'est là que vous avez un problème pour le dernier paramètre. Il est demandé un contexte (une Activity) alors que vous êtes dans le contrôleur. Du coup, faites en sorte d'ajouter un paramètre de type Context à la méthode creerProfil pour pouvoir utiliser ce paramètre. Il faut alors aussi modifier l'appel de la méthode creerProfil dans la classe MainActivity pour envoyer le contexte (il suffit d'envoyer this en dernier paramètre, car this représente l'instance de MainActivity, donc du contexte).

Pour le moment cela ne sert à rien de tester. Il faut d'abord voir comment récupérer un objet sérialisé.

1C. Récupération de l'objet

Il ne reste plus qu'à récupérer l'objet. C'est quelque chose qu'il faut faire dès le démarrage de l'application, et il faut en profiter pour valoriser les objets graphiques. Car effectivement, le but est d'afficher les informations précédemment saisies.

Puisqu'il y a plusieurs traitements à faire, autant les mettre dans une méthode. Toujours dans la classe Controle, créez la méthode privée et statique recupSerialize qui reçoit en paramètre un contexte et qui ne retourne rien. Écrivez le code de la méthode qui doit juste valoriser la propriété profil avec le résultat de la désérialisation (donc en lui affectant l'appel de la méthode statique deserialize de la classe Serializer, avec les bons paramètres). La ligne de code est soulignée en rouge : c'est normal. La méthode deSerialize retourne un objet type Object : vous devez donc le transtyper en Profil avant l'affectation. Si aucun objet n'a été sérialisé, la propriété profil va recevoir null.

Cette méthode doit être appelée au moment du démarrage de l'application, donc lors de la création de l'instance de Controle : dans la méthode getInstance, juste après avoir créé l'instance (donc une ligne après le new, dans le if), appelez la méthode recupSerialize… mais il faut un contexte ! Donc ajoutez en paramètre de getInstance un contexte de type Context, et apportez les modifications nécessaires (ajoutez this lors de l'appel de getInstance, dans MainActivity).

Il faut aussi que le contrôleur soit capable de retourner les informations du profil. Donc, en suivant la même logique que les méthodes getIMG et getMessage (dans la classe Controle), écrivez les méthodes qui permettent de récupérer la taille, le poids, l'âge et le sexe. Mais attention, il ne faut pas retourner une information du profil si le profil est null. Donc, dans chacune des 4 méthodes, testez si le profil est null : dans ce cas retournez null, sinon retournez l'information désirée.

Il faut maintenant exploiter tout ça dans la classe MainActivity.

Dans la classe MainActivity, créez la méthode privée recupProfil() qui ne reçoit aucun paramètre et qui ne retourne rien. Cette méthode doit commencer à tester une des propriétés du profil (par exemple, testez la taille récupérée par le contrôleur). Dans le cas où la propriété n'est pas nulle, il suffit de

valoriser chaque objet graphique par les informations récupérées grâce aux getters qui sont dans la classe Controle. Pour cela, utilisez la méthode setText sur l'objet correspondant. Attention, si vous mettez en paramètre de setText, une variable de type Integer, vous allez avoir des problèmes. Il suffit de transformer cet Integer en texte : une méthode très simple consiste à concaténer en premier une chaine vide. Voici un exemple :

```
txtTaille.setText(""+controle.getTaille());
```

Vous pouvez aussi appliquer la méthode toString() qui transforme n'importe quoi en chaîne.

Attention, pour le sexe, c'est la propriété setChecked qu'il va falloir utiliser sur l'objet rdHomme. Le problème se situe aussi sur le bouton radio de la femme qui doit être sélectionné si le sexe est à 0. Donc, comme vous l'avez fait pour rdHomme, rajoutez en haut de la classe, la propriété rdFemme, sans oublier de la valoriser dans la méthode init avant de l'utiliser dans recupProfil.

Une fois les 4 informations utilisées pour remplir les objets graphiques (taille, poids, âge et sexe), il faut simuler le clic sur le bouton Calcul. Pour cela, il suffit d'appliquer la méthode performClick() sur btnCalc (la propriété privée liée au bouton dans l'interface).

La méthode recupProfil doit être appelée au chargement : à la fin de la méthode init et surtout après l'appel de ecouteCalcul() qui initialise l'écoute du bouton calcul (sinon, le performClick() va poser problème).

Il est temps de tester : lancez l'application, faites un premier calcul. Stoppez l'application. Relancez l'application. Si tout marche bien, cette fois les zones ne sont pas vides et contiennent les informations saisies la fois précédente. Le résultat de l'IMG doit aussi s'afficher.

Ici on s'est contenté d'enregistrer un objet. On pourrait enregistrer de la même façon une collection d'objets. Mais c'était juste pour vous montrer que ce genre d'enregistrement existe.

Pensez à faire un "commit and push" pour l'enregistrement sur Bitbucket.

2. Enregistrement en local avec Sqlite

Vous venez de le voir : la sérialisation marche très bien pour mémoriser des objets. Cependant, il peut être parfois plus pratique d'enregistrer les informations dans une base de données. La base de données offre en particulier l'avantage de pouvoir utiliser le langage SQL pour exploiter les informations qu'elle contient.

Dans cette étape, voyons comment enregistrer dans une base de données locale au téléphone. Le SGBDR correspondant s'appelle Sqlite et permet des opérations minimales mais suffisantes pour la création et l'exploitation d'une base de données locale.

Mais avant tout, il faut enlever les effets de la sérialisation qui ne servira plus : mettez en commentaire dans la classe Controle, l'appel de recupSerialize dans getInstance et la demande de sérialisation dans creerProfil. Ne touchez pas au reste qui va nous servir.

2A. Mémorisation de plusieurs mesures

Pour le moment, un seul lot de valeurs (poids, taille, âge, sexe) est enregistré. Pour qu'il y ait plusieurs tuples dans l'unique table Profil que l'on va bientôt créer, le but va être de mémoriser plusieurs valeurs, qui seront identifiées par la date et l'heure de la mesure.

2A1. Ajout de la date

Dans la classe Profil, ajoutez la propriété dateMesure de type Date (en récupérant l'import java.util.Date), avec le getter correspondant.

Dans le constructeur de la classe Profil, ajoutez le paramètre dateMesure et utilisez-le pour valoriser la propriété correspondante.

Remarquez le message en rouge "2 related problems" au dessus de la méthode. Si vous cliquez dessus, vous pouvez voir, en bas, où se situent les problèmes. En effet, 2 appels du constructeur Profil existent (un dans la classe Controle, et un dans ProfilTest). Si vous double cliquez en bas à gauche, sur la ligne :

```
profil = new Profil(poids, taille, age, sexe);
```

vous allez arriver sur la ligne de l'erreur, dans la classe Controle. Dans la classe Controle, une erreur apparait au niveau de l'instanciation de la classe Profil. C'est normal puisqu'il manque un paramètre. Ajoutez le paramètre manquant en envoyant la date du jour (qui s'obtient tout simplement en instanciant la classe Date du package java.util). Enregistrez. En revenant sur la classe Profil, ll n'y a plus qu'une erreur. En cliquant dessus, vous allez être directement redirigé vers la classe ProflTest. Corrigez aussi l'erreur ici.

2B. La classe SQLiteOpenHelper

Commencez par récupérer la classe MySQLiteOpenHelper qui se trouve dans les sources, et copiez-la dans le dossier outils (en passant par l'explorateur, pas par Android Studio). Vérifiez que la classe apparait sous Android Studio (cliquez juste sur "outils" pour qu'elle apparaisse) et pensez à modifier la première ligne (le package) si vous avez une erreur (ce qui ne devrait normalement pas être le cas).

Regardez le contenu de cette classe qui est très courte. La classe hérite de SQLiteOpenHelper, qui permet de créer et/ou accéder à une base SQLite dans le mobile.

Le constructeur se contente d'appeler le constructeur de la classe mère en lui envoyant essentiellement les paramètres reçus (le contexte le nom de la base de données et le numéro de version car il est possible de mémoriser plusieurs versions d'une même base).

Deux méthodes sont automatiquement redéfinies (onCreate et onUpgrade). Seule onCreate nous intéresse. Cette méthode n'est appelée que si le nom de la base reçu dans le constructeur n'existe pas encore. Donc elle ne s'exécute qu'une seule fois par base de données, contrairement au constructeur qui est appelé à chaque fois que l'on crée un objet de type MySQLiteOpenHelper qui va aussi permettre d'accéder à la base de données.

Vous remarquerez que la classe est déjà en partie remplie : une propriété de type String est déjà présente, contenant la requête de création de la table profil dont on va avoir besoin, et cette requête est exécutée dans le onCreate.

Suivant les projets que vous pourrez créer sous Android Studio (cliquez juste sur "outils" pour qu'elle apparaisse), si vous voulez manipuler une BDD Sqlite, il suffira de récupérer cette classe et de l'adapter à vos besoins, entre autres en modifiant la requête de création de la ou des tables nécessaires.

2C. SQLite (rapide présentation)

Avant de poursuivre, voici une rapide présentation de SQLite.

C'est en fait juste un moteur de base de données relationnel, qui s'intègre obligatoirement directement dans un programme et qui est basé sur le standard SQL92 (en plus léger). Il n'y a donc pas de SGBDR dissocié qui offrirait des outils indépendants de gestion des données (pour voir par exemple la structure de la base ou le contenu des tables). Cependant, il existe des outils indépendants qui permettent de consulter le contenu d'une base SQLite. On va le voir plus loin.

Une base de données SQLite n'offre pas la possibilité d'être interrogée à distance. Cependant plusieurs programmes locaux peuvent accéder à la même base de données, mais il n'y a pas de possibilité de gestion de droits d'accès.

Le gros avantage de SQLite est sa légèreté, d'où son intérêt pour les technologies embarquées, et le fait qu'elle soit directement intégrée dans le mobile. Cela ne veut cependant pas dire que son utilisation se limite à ce type de technologies.

2C1. Types de données

Les types utilisables sous SQLite sont limités mais flexibles (si un format ne fonctionne pas, un autre plus adapté est adopté) :

TEXT, NUMERIC, INTEGER, REAL

Donc une date sera soit enregistrée en texte, soit en integer avec dans les 2 cas des conversions nécessaires.

2C2. Contraintes

Contraintes possibles sur une colonne : NOT NULL, CHECK, DEFAULT, AUTOINCREMENT

Contraintes possibles sur plusieurs colonnes : PRIMARY KEY, UNIQUE, FOREIGN KEY

Autres opérations :

Il est possible de mémoriser des déclencheurs, de créer des vues et même de gérer des transactions. Cependant, par défaut, les opérations sont atomiques et autocommitées.

2D. Classe d'accès à la base de données

Le but est de regrouper dans une classe tout ce qui concerne l'accès à la base locale. Pour simplifier, plutôt que de faire un nouveau package, on va placer cette classe dans le package "modele" : dans ce package, créez la classe "AccesLocal".

Dans cette classe, on trouvera en plus du constructeur, dans un premier temps, une méthode qui permet d'ajouter un profil dans la table, et une méthode qui permet de récupérer le dernier profil ajouté. C'est tout ce dont on a besoin pour le moment dans notre application. Vous allez alors penser : pourquoi enregistrer plusieurs profils si on n'en récupère qu'un seul ? Il est surtout question ici de vous montrer le fonctionnement de SQLite. Dans un second temps, avec l'accès à une base distante, on manipulera plusieurs profils.

2D1. Les propriétés

Dans cette classe, déclarez trois propriétés pour mémoriser le nom de la base de type String (nomBase contenant "bdCoach.sqlite"), la version de type Integer (versionBase contenant 1) et enfin une propriété accesBD du type de la nouvelle classe (MySQLiteOpenHelper) qui permettra d'accéder à la base.

2D2. Le constructeur

Écrivez le constructeur qui doit recevoir un contexte en paramètre : en effet, on en aura besoin pour instancier la classe MySQLiteOpenHelper.

Dans le constructeur, valoriser la propriété accesBD en lui affectant une instance de la classe MySQLiteOpenHelper avec les bons paramètres.

2D3. Ajout d'un profil

Il va être nécessaire de pouvoir ajouter un profil dans la table profil. Écrivez la méthode publique ajout qui reçoit en paramètre un objet de type Profil et qui ne retourne rien.

Pour accéder à la base locale, vous avez besoin de déclarer une propriété bd de type SQLiteDatabase : déclarez cette propriété en plus des 3 propriétés précédentes.

Dans la méthode ajout, valorisez la propriété bd en lui affectant l'appel de la méthode getWritableDatebase() sur l'objet accesBD. Ainsi vous avez un accès en écriture sur la base de données.

Une première méthode pour créer et exécuter la requête d'insertion serait dans un premier temps de construire la requête dans une chaîne, en concaténant les valeurs provenant de l'objet profil :

```
String req = "insert into profil (datemesure, poids, taille, age, sexe) values ";
req += "(\"" + profil.getDateMesure() + "\","+ profil.getPoids() + "," + profil.
getTaille() + "," + profil.getAge() + "," + profil.getSexe() + ")";
```

puis d'exécuter cette requête :

```
bd.execSQL(req);
```

Mais cette technique, qui fonctionne, ne permet pas d'éviter les les injections SQL. Ceci dit, précisément dans cette application, il ne peut pas y avoir d'injection SQL car les seules valeurs saisies sont des nombres. Mais autant prendre de bonnes habitudes.

Vous allez donc déclarer et instancier un objet values de type ContentValues. Il va permettre de mémoriser les valeurs pour chaque attribut. Il suffit ensuite d'appliquer la méthode put sur cet objet. Cette méthode attend en premier paramètre, le nom de l'attribut dans la table, et en second paramètre, la valeur à affecter à l'attribut. Par exemple :

```
values.put("poids", profil.getPoids());
```

Il faut donc le faire pour chaque attribut. Pensez à convertir en String (avec toString) la propriété dateMesure qui est au format Date.

Ensuite, pour gérer l'insert, il faut appeler la méthode insert sur l'objet bd, Mettez le nom de la table en premier paramètre, null en second paramètre et values en troisième paramètre, pour envoyer les valeurs à insérer.

Vous pouvez enfin fermer la connexion avec la méthode close sur l'objet bd.

2D4. Récupération du dernier profil

Avant de commencer cette méthode, sachez que cette fois vous allez devoir manipuler un curseur. Voici les instructions accessibles sur un curseur en SQLite :

```
void moveToFirst() /* se positionne sur le premier tuple */
void moveToLast() /* se positionne sur le dernier tuple */
void moveToNext() /* se positionne sur le tuple suivant */
Boolean isAfterLast() /* vrai si après le dernier atteint */
Boolean isBeforeFirst() /* vrai si avant le premier atteint */
int getInt(int indice) /* récupère le contenu de type int du champ de la ligne
active dont l'indice est passé en paramètre (l'indice commence à 0) */
float getFloat(int indice) /* idem avec le type float */
String getString(int indice) /* idem avec le type string */
void close() /* ferme le curseur */
```

Vous allez maintenant écrire la méthode recupDernier qui ne reçoit aucun paramètre et qui retourne un objet de type Profil.

Commencez par déclarer dans la méthode, un objet profil de type Profil en l'initialisant à null. Plus loin, si on arrive à récupérer un profil dans la base de données, cet objet sera valorisé avec les informations récupérées.

Valorisez la propriété bd comme vous l'aviez fait dans la méthode précédente, mais cette fois en utilisant la méthode getReadableDatabase() car on n'a juste besoin d'accéder en lecture à la base de données.

Créez une variable req de type String et affectez-lui la requête select qui permet de récupérer les profils.

Cette fois, il va falloir utiliser un curseur pour accéder au résultat de la requête. Toujours dans la méthode, déclarez un objet curseur de type Cursor, et affectez-lui le résultat de l'appel de la méthode rawQuery sur l'objet bd. Cette méthode attend 2 paramètres : le premier est la requête précédemment écrite, mettez null pour le second.

Positionnez-vous sur la dernière ligne du curseur (référez-vous aux méthodes données dans l'encadré ci-dessus).

Il faut maintenant contrôler qu'il y a bien au moins une ligne dans le curseur. Faites le test nécessaire (avec par exemple isAfterLast en contrôlant qu'il est à faux, donc qu'on n'est pas après le dernier). Si le test est correct, récupérez dans des variables locales toutes les informations de la ligne du curseur, sachant que datemesure est à l'indice 0, poids à l'indice 1, etc. Vous allez avoir un problème avec datemesure qui est de type String et que vous devez convertir en Date. Pour le moment vous ne savez pas faire et ce n'est pas si simple. Dans un premier temps, nous n'avons pas besoin de la date, donc contentez-vous d'affecter la date du jour (en gardant du coup un type Date).

Une fois les informations récupérées, valorisez l'objet profil en lui affectant une instance de la classe Profil avec les bons paramètres.

Après avoir fermé le if, fermez le curseur.

Il ne reste plus qu'à retourner le profil.

2E. Travail avec le contrôleur

Comme d'habitude, c'est le contrôleur qui va s'occuper de gérer les échanges entre la vue et le modèle.

Dans la classe Controle, ajoutez la propriété statique accesLocal de type AccesLocal.

Dans la méthode getInstance, juste après avoir créé l'instance, instanciez l'objet accesLocal. Puis, valorisez l'objet profil en lui affectant l'appel de la méthode recupDernier sur l'objet accesLocal (au cas où il y aurait un profil à récupérer). Vérifiez que l'appel de recupSerialize est bien mis en commentaire.

Dans la méthode creerProfil, si vous ne l'avez pas encore fait, mettez en commentaire la ligne de sérialisation et ajoutez la ligne de code qui permet d'ajouter le nouveau profil dans la base locale (en utilisant la méthode ajout sur l'objet accesLocal).

Faites le test : lancez l'application, saisissez un profil, calculez puis fermez l'application. Normalement le profil a dû s'enregistrer dans la base de données. Relancez l'application, le profil enregistré doit apparaître automatiquement.

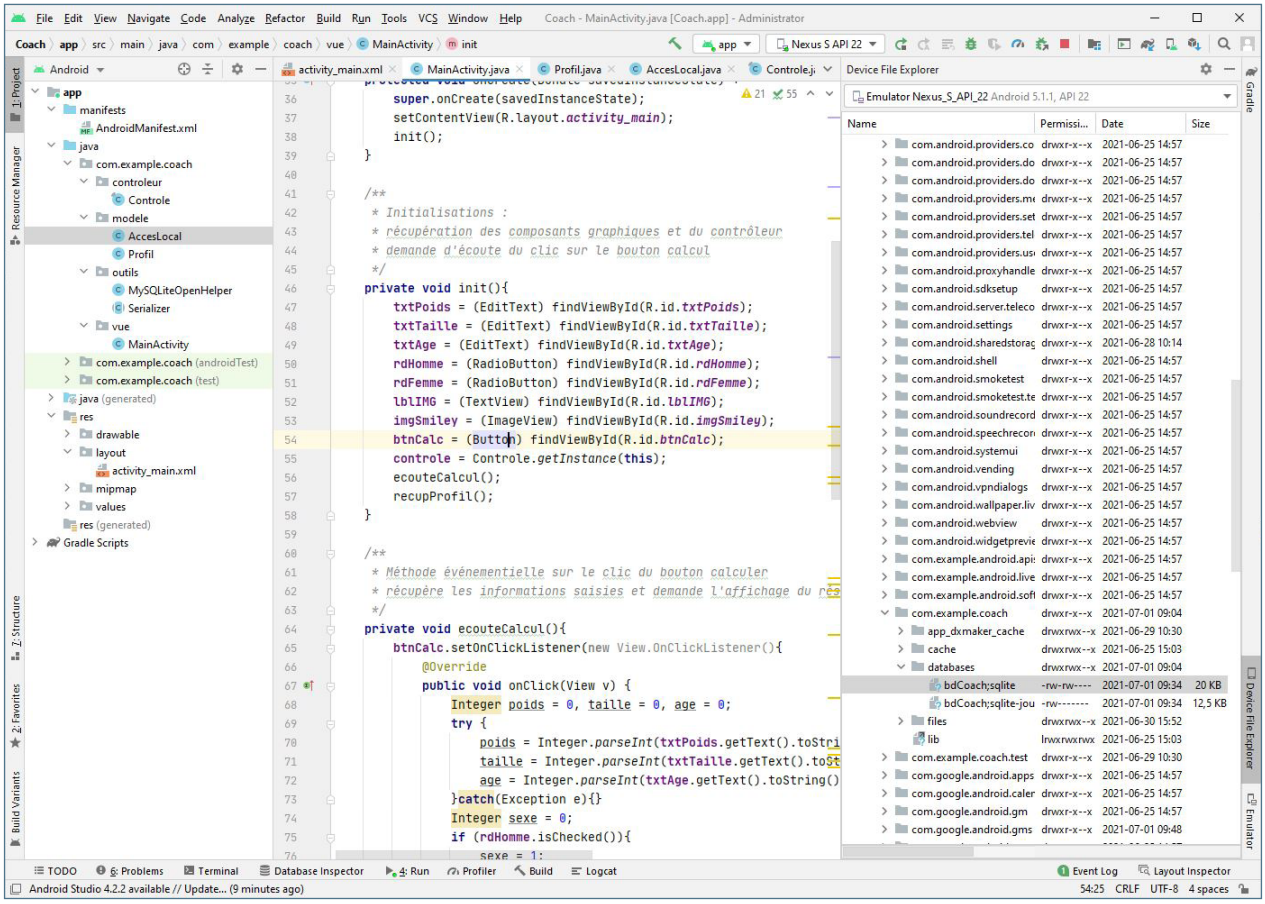
Si ça ne marche pas, n'hésitez pas à utiliser le débogueur et aussi Log.d pour faire des affichages consoles en particulier dans la méthode de récupération du profil, pour mieux situer l'erreur. Lisez aussi la partie suivante "Base de données dans le téléphone" qui peut vous aider à contrôler le contenu de la base de données.

2F. Base de données dans le téléphone

Il est possible de contrôler que la base de données est bien enregistrée dans le téléphone.

Sous Android Studio, cliquez en bas à droite sur "Device File Explorer" (écrit verticalement). La partie "Device File Explorer s'ouvre à droite et juste en dessous du titre, vous pouvez voir un combo qui contient

les différents émulateurs. Normalement vous êtes positionné sur le bon (celui que vous venez d'utiliser) mais sinon, il suffit de sélectionner l'émulateur sur lequel vous avez fait le test de la BDD. Il faut bien sûr que l'émulateur soit ouvert (ou connecté). Ouvrez "data > data > com.example.coach > databases". Vous devriez voir bdCoach.sqlite comme la capture ci-dessous :



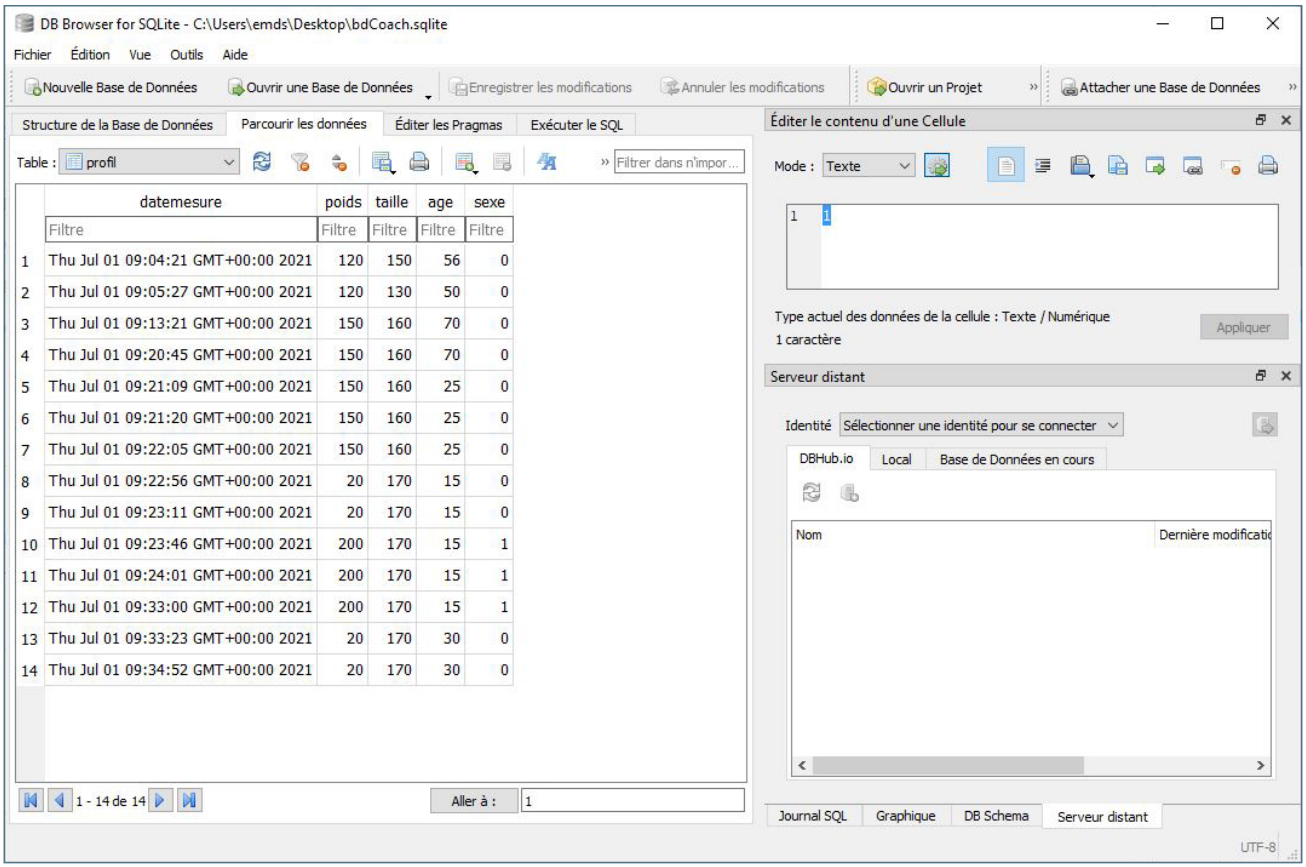
C'est à cet endroit que vous pourrez, si vous le voulez, supprimer la base de données (en faisant un clic droit sur le fichier bdCoach.sqlite et en sélectionnant "delete", idem pour bdCoach.sqlite-journal). Cette manipulation est bien utile quand on veut changer la structure de la base de données, sachant que la création de la structure ne se fait qu'une fois, à la création (onCreate) de la base de données.

Pour le moment, sélectionnez bdCoach.sqlite, faites un clic droit et choisissez "save as" pour enregistrer le fichier où vous voulez sur votre ordinateur (excepté dans le dossier du projet).

Il faut maintenant un logiciel qui soit capable de voir le contenu du fichier. Allez sur le site "http://sqlitebrowser.org/" et récupérez "DB Browser for SQLite" correspondant à votre système (en allant dans la partie "download) puis installez-le.

Une fois installé, lancez le logiciel.

Allez dans le menu "Fichier > ouvrir une base de données". Sélectionnez le fichier téléchargé. Vous allez pouvoir voir la structure et le contenu de la base de données. Allez dans l'onglet "Parcourir les données" et, dans le combo du haut à gauche permettant de sélectionner une table, sélectionnez "profil". Vous allez voir la liste des profils enregistrés.



Cela va vous permettre de contrôler que des enregistrements ont bien été effectués dans la base de données.

Remarquez au passage qu'il a enregistré 2 fois les mêmes mesures : cela vient du fait qu'au lancement de l'application, on récupère le dernier profil et on clique à nouveau sur calcul, ce qui provoque un nouvel enregistrement du profil. Pour le moment ce n'est pas un problème, mais vous voyez qu'utiliser un browser permet de mieux analyser ce qui s'est passé dans la base de données.

2G. Formatage de la date

On avait laissé de côté le formatage de la date. Ceci ne se faisant pas en une seule ligne, il est judicieux de créer une méthode qui permet de convertir un String en Date. Mais où mettre cette méthode ? Ces méthodes "outils" devraient logiquement être rassemblées dans des classes positionnées dans le package "outil". Et c'est d'ailleurs tout à fait le genre d'outils qui pourront être réutilisés dans d'autres applications. Dans ce package, créez la classe abstraite MesOutils. Pourquoi abstraite ? Parce qu'on ne va jamais instancier cette classe : on veut juste directement accéder à ces méthodes pour les utiliser comme des outils. Donc toutes les méthodes seront statiques.

Dans cette classe, créez la méthode convertStringToDate qui reçoit en paramètre uneDate de type String et qui va retourner un objet de type Date.

Dans cette méthode, mettez le code suivant :

```
String expectedPattern = "EEE MMM dd hh:mm:ss 'GMT+00:00' yyyy";
SimpleDateFormat formatter = new SimpleDateFormat(expectedPattern);
try {
    Date date = formatter.parse(uneDate);
    return date;
} catch (ParseException e) {
    e.printStackTrace();
}
return null;
```

Le but est d'utiliser un objet de type SimpleDateFormat pour formater la chaine afin de générer une date. Cette opération pouvant échouer (si la chaine ne contient pas une date), voilà pourquoi il est nécessaire d'utiliser un try/catch.

Toujours dans la même classe, autant créer tout de suite la méthode qui fait le contraire. Écrivez la méthode convertDateToString et copiez le code suivant :

```
SimpleDateFormat date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
return date.format(uneDate);
```

Dans la classe AccesLocal, méthode recupDernier, remplacez :

```
Date dateMesure = new Date();
```

Par la récupération de la date dans la table (qui est au format String dans la première colonne d'indice 0) après l'avoir convertie avec la méthode convertStringToDate.

Comme la date n'est pour le moment pas affichée, faites juste un Log.d (un affichage console) après cette ligne, pour afficher dateMesure dans la console et voir si le format est correct. Pour voir le Log.d, pensez à ouvrir la console Logcat (en bas), lancez l'application une première fois, saisissez et calculez un img, fermez l'application et relancez-là. Vous devriez obtenir un affichage console qui ressemble à ceci (par exemple pour la date 01/07/2021 10h44m55s) :

```
Thu Jul 01 10:44:55 GMT+00:00 2021
```

Donc au format :

```
EEE MMM dd hh:mm:ss 'GMT+00:00' yyyy
```

Si vous avez bien tout observé, c'est le format qui a été utilisé pour la conversion. Si vous remarquez une différence dans le format (par exemple, si vous obtenez un formatage différent, alors modifiez la fonction convertStringToDate en conséquence).

Pour le moment, les informations du dernier profil sont à nouveau enregistrées dans la base locale puisque l'enregistrement se fait à chaque clic sur le bouton calculer, et justement on provoque un clic au chargement. Ce n'est pas très grave.

Si vous vous demandez pourquoi on s'est fatigué à mémoriser la date alors qu'elle ne va pas nous servir, c'est parce qu'on avait besoin d'un identifiant dans la base de données. Et rassurez-vous, elle va servir par la suite.

Pensez à faire un "commit and push" pour l'enregistrement sur Bitbucket.

CNED, BP 60200, 86980 Futuroscope Chasseneuil Cedex, France

87D22TDWB1B21