Cas EDF : Développement Android - Concepts avancés – Partie 4

Cette publication comporte cinq parties dont l'ordre est dicté par la logique du développement.
Les parties 2 et 3 sont facultatives.

Partie 1 : Gestion des clients
Partie 2 : Géolocalisation de l'agent et géocodage du client sélectionné
Partie 3 : Signature Client
- **Partie 4 : Communication avec le serveur**

Partie 5 : Identification, import et export des données.

Description du thème

Propriétés	Description
Intitulé long	**Cas EDF : Développement Androïd - Concepts avancés - Partie 4 : Communication avec le serveur**
Formation concernée	BTS Services Informatiques aux Organisations
Matière	SLAM 4
Présentation	Développement permettant d'aborder des concepts de la programmation Android d'une application embarquée, communiquant avec un serveur. Il aborde les notions : ➢ d'affichage de liste / d'adapter, ➢ de GEOLOCALISATION / GEOCODER, ➢ de graphisme (canvas) et d'encodage JPG, ➢ d'échange avec un serveur WEB (THREAD / JSON / GSON), ➢ d'utilisation d'un SGBDO DB4o.
Notions	Savoirs • D4.1 - Conception et réalisation d'une solution applicative • D4.2 - Maintenance d'une solution applicative Savoir-faire • Programmer un composant logiciel • Exploiter une bibliothèque de composants • Adapter un composant logiciel • Valider et documenter un composant logiciel • Programmer au sein d'un framework
Transversalité	SLAM5
Pré-requis	Développement d'une application Android sous un environnement Eclipse. (Exemple : Cas AMAP Jean-Philippe PUJOL)
Outils	Eclipse, DB4o, OME, Gson, Google play services, Apache, Mysql
Mots-clés	Application mobile, Android, SGBDO, DB4o, Géolocalisation, Géocodage, Thread, json, Gson, MVC, canvas, encodage JPG
Durée	24 heures (8,4,4,4,4) (Temps divisé par 2 si utilisation du squelette application)
Auteur	Pierre François ROMEUF avec la relecture et les judicieux conseils de l'équipe CERTA
Version	v 1.0
Date de publication	Juin 2014

Contexte

Application embarquée sur une solution technique d'accès (STA) sous Android, permettant à un agent EDF d'effectuer sa tournée journalière de relevés des compteurs EDF.

Les principales fonctionnalités sont :

- Identification de l'agent sur le device avec contrôle sur un serveur web,
- Import des clients depuis un serveur web,
- Affichage des clients,
- Saisie des informations clients,
- Aide au déplacement via géolocalisation de la position de l'agent EDF et géocodage de l'adresse client,
- Enregistrement de la signature client validant les informations saisies,
- Export des données sur le serveur web.

Le SGBD embarqué est un SGBDO DB4o. Le serveur distant est un serveur de type LAMP installé sur la ferme de serveurs ou via un hébergement gratuit (ex : http://www.hostinger.fr/)

Cette application peut être dérivée pour de multiples besoins : ceux des livreurs, des commerciaux, des visiteurs, des contrôleurs ...

Le code fourni en annexe nécessite de votre part une compréhension.
Il représente normalement votre travail de programmeur, de fouille sur internet, avec tests, compréhension et modifications du code.

Il vous est fourni afin que ce développement ne représente qu'un travail raisonnable et pour vous présenter les différentes facettes du développement sur Android.

Conseils : consultez notamment developer.android.com, le tutoriel du zéro (http://uploads.siteduzero.com/pdf/554364-creez-des-applications-pour-android.pdf), etc.

Communication avec le serveur

Tâches asynchrones

Une *Activity* Android ne peut se mettre en attente d'un résultat long au regard des différents états d'une *Activity* car le système peut, à n'importe quel moment, reprendre la main pour assurer le bon fonctionnement de l'appareil (exemple : appel téléphonique).

Android impose que toutes les opérations potentiellement lentes (accès réseau, fichiers, accès base de données, calculs complexes,...) s'exécutent de manière asynchrone.

La représentation la plus classique de tâches asynchrones est le *thread*.

Il est important de préciser qu'un *thread* n'est pas un processus. En effet, les processus vivent dans des espaces virtuels isolés alors que les *threads* sont des traitements qui vivent ensemble au sein d'un même processus.

Les threads partagent la même mémoire contrairement aux processus.

Un *thread* est donc une portion de code capable de s'exécuter en parallèle à d'autres traitements. Ils servent à maintes choses par exemple :

- faire des traitements en tâche de fond, c'est le cas de la coloration syntaxique des éditeurs ;
- exécuter plusieurs instances d'un même code pour accélérer le traitement, pour de longs traitements n'utilisant pas les mêmes ressources ;
- s'adresser de manière personnelle à plusieurs clients simultanément comme les serveurs HTTP ou les chats.

Les *threads* ne s'exécutent pas en même temps, mais en temps partagé, c'est pour cette raison qu'il est important pour un *thread* de toujours laisser une chance aux autres de s'exécuter.

Par défaut, une application Android s'exécute dans un processus unique, le processus dit "principal" (aussi appelé *UI thread*).

Dans le processus principal, le système crée un *thread* d'exécution pour l'application : le *thread* principal (UI). Il est, entre autres, responsable de l'interface graphique et des messages/notifications/événements entre composants graphiques.

Cependant, il est interdit d'effectuer des opérations sur l'interface graphique en dehors du *thread* principal, ce qui se résume dans la documentation sur les threads par deux règles :

- *Do not block the UI thread*
- *Do not access the Android UI toolkit from outside the UI thread.*

Android fournit des méthodes pour résoudre le problème précédemment évoqué. Il s'agit de créer des objets exécutables dont la partie affectant l'interface graphique n'est pas exécutée, mais déléguée à l'UI thread pour exécution ultérieure.

Cette communication inter-threads pourra se faire à l'aide de la classe Handler qui sera capable d'envoyer des messages à notre UI Thread.

Possibilité de traitements asynchrones

Thread

La classe *Thread* du package java.lang est celle qui doit impérativement être dérivée pour qu'une classe puisse être considérée comme un *thread* et donc, exécutable en parallèle.

Cette classe concrète implémente l'interface *Runnable*.

Le *thread* peut avoir quatre états différents :

- Nouveau : C'est l'état initial après l'instanciation du *thread*. À ce stade, le *thread* est opérationnel, mais celui-ci n'est pas encore actif. Un *thread* prend cet état après son instanciation.

- Exécutable : Un *thread* est dans un état exécutable à partir du moment où il a été lancé par la méthode *start()* et le reste tant qu'il n'est pas sorti de la méthode *run()*.
  Dès que le système le pourra, il donnera du temps d'exécution à votre *thread*.

- Attente : Un *thread* en attente est un thread qui n'exécute aucun traitement et ne consomme aucune ressource CPU. Il existe plusieurs manières de mettre un thread en attente. Par exemple :
  - appeler la méthode *thread.sleep* (temps en millisecondes) ;
  - appeler la méthode *wait()* ;
  - accéder à une ressource bloquante (flux, accès en base de données, etc.) ;
  - accéder à une instance sur laquelle un verrou a été posé.

  Un thread en attente reste considéré comme exécutable.

- Mort : Un *thread* dans un état mort est un *thread* qui est sorti de sa méthode *run()* soit de manière naturelle, soit de manière subite.

Exemple : S'il s'agit de faire une tâche répétitive qui ne consomme que très peu de temps CPU à intervalles réguliers, une solution ne nécessitant pas de processus à part consiste à programmer une tâche répétitive à l'aide d'un *TimerTask*. (Toutes les x secondes faire…).

Services

Un service est une activité sans interface graphique qui sert à effectuer des opérations ou des calculs en dehors de l'interaction utilisateur.

Exemples d'utilisation :

- Écoute un fichier audio mp3, pendant la navigation sur le web ;
- Les coordonnées GPS sont envoyées régulièrement ;
- Communication régulière avec un serveur web pour recevoir des notifications (à l'inverse de la méthode *push*).

La structure d'une classe de service ressemble à une activité.
Pour réaliser un service, on hérite de la classe Service et on implémente les méthodes de création/démarrage/arrêt du service.

Classes spécifiques Android

En plus des classes de programmation concurrentes de Java (*Thread*, *Executor*, *ThreadPoolExecutor*, *FutureTask*, *TimerTask*, etc.), Android fournit quelques classes supplémentaires pour programmer des tâches concurrentes (*ASyncTask*, *IntentService*, etc.).

La classe *AsyncTask* permet d'exécuter des opérations en tâche de fond et de publier les résultats dans l'interface graphique sans avoir à manipuler de *thread* ou de *handler*.
Le handler peut être considéré comme un pointeur (poignée) vers un objet de l'activity principal qui démarre le thread, permettant au thread d'interagir avec cetobjet. Principalement l'objet est un objet graphique; exemple barre d'avancement, compteur...

Les méthodes *d'AsyncTask*

- *onPreExecute()* : invoquée au démarrage. Cette étape est normalement utilisée pour initialiser la tâche, par exemple en montrant une barre de progression à l'interface.
- *doInBackground(Params...)* : invoquée juste après la fin de *onPreExecute()*.
  Cette étape est utilisée pour effectuer l'action en tâche de fond à proprement parler et peut prendre beaucoup de temps. Les paramètres de la tâche asynchrone sont passés à cette étape. Le résultat de l'opération doit être retourné par cette étape et sera passé à la dernière étape. Cette étape peut aussi utiliser *publishProgress(Progress...)* pour publier une ou plusieurs unités de progression. Ces valeur sont publiées au thread de l'UI, à l'étape *onProgressUpdate(Progress...)*.
- *onProgressUpdate(Progress...)* : invoqué par le thread de l'UI après un appel à la méthode *publishProgress(Progress...)*.
  Le *timing* de l'exécution est indéfini. Cette méthode est utilisée pour afficher n'importe quelle forme de progression dans l'UI pendant que la tâche de fond est toujours en cours d'exécution, par exemple pour animer une barre de progression ou pour montrer un log dans un champ texte.
- *onPostExecute(Result)*, : invoquée dans le *thread* de l'UI après la fin de l'exécution de la tâche.
  Le résultat de l'opération de tâche de fond est passé à cette étape en paramètre.

Une *AsyncTask* doit obligatoirement implémenter la méthode ***doInBackground***. C'est elle qui réalisera le traitement de manière asynchrone dans un *Thread* séparé.

Les méthodes ***onPreExecute*** (appelée avant le traitement), ***onProgressUpdate*** (appelée lorsque vous souhaitez afficher sa progression) et ***onPostExecute*** (appelée après le traitement) sont optionnelles.

Un appel à la méthode ***publishProgress*** permet la mise à jour de la progression. On ne doit pas appeler la méthode *onProgressUpdate* directement.

Attention, ces trois méthodes (*onPreExecute*, *onProgressUpdate* et *onPostExecute*) s'exécutent depuis *l'**UI Thread***. C'est d'ailleurs grâce à cela qu'elles peuvent modifier l'interface. On ne doit donc pas y effectuer de traitements lourds.

Communication avec un serveur web

Android contient le package réseau standard *Java java.net* qui peut être utilisé pour accéder aux ressources du net.

Android contient également la bibliothèque *HttpClient Apache*.

La classe de base pour l'accès au réseau via HTTP dans le package *java.net* est la classe de *HttpURLConnection*.
La meilleure façon d'accéder à l'Internet selon Google est la classe *HttpURLConnection*, maintenue et améliorée par Google.
Cette classe est très simple d'utilisation.

Pour savoir si le réseau est actif :

```
public boolean isNetworkAvailable() {
    ConnectivityManager cm = (ConnectivityManager)
      getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = cm.getActiveNetworkInfo();
    if (networkInfo != null && networkInfo.isConnected()) {
        return true;
    }
    return false;
}
```

JSON

JSON (*JavaScript Object Notation* – Notation Objet issue de JavaScript) est un format léger d'échange de données. Il est facile à lire ou à écrire pour des humains. Il peut être aisément analysé ou généré. JSON est un format texte complètement indépendant de tout langage.

JSON se base sur deux structures :

- Une collection de couples nom/valeur. Divers langages la réifient par un objet, un enregistrement, une structure, un dictionnaire, une table de hachage, une liste typée ou un tableau associatif.
- Une liste de valeurs ordonnées. La plupart des langages la réifient par un tableau, un vecteur, une liste ou une suite.

Ces structures de données sont universelles.

En JSON, elles prennent les formes suivantes :

- Un objet, qui est un ensemble de couples nom/valeur non ordonnés. Un objet commence par { (accolade gauche) et se termine par } (accolade droite). Chaque nom est suivi de : (deux-points) et les couples nom/valeur sont séparés par , (virgule).
- Un tableau est une collection de valeurs ordonnées. Un tableau commence par [(crochet gauche) et se termine par] (crochet droit). Les valeurs sont séparées par , (virgule).
- Une valeur peut être soit une *chaîne de caractères* entre guillemets, soit un *nombre*, soit true ou false ou null, soit un objet soit un tableau. Ces structures peuvent être imbriquées.
- Une chaîne de caractères est une suite de zéro ou plus caractères Unicode, entre guillemets, et utilisant les échappements avec *antislash*. Un caractère est représenté par une chaîne d'un seul caractère.

Exemple d'un objet liste de course qui contient 2 objets fruits et légumes :
"fruits" est un objet de type tableau, premier élément un objet de 3 couples nom/valeurs, deuxième élément un objet d'un couple nom/valeur.
"légume" est un objet dont la valeur est un objet de 3 couples nom/valeur :

```
{
 "fruits": [
   { "kiwis": 3,
     "mangues": 4,
     "pommes": null
   },
   { "panier": true },
 ],
 "legumes":
   { "patates": "amandine",
     "figues": "de barbarie",
     "poireaux": false
   }
}
```

Activity Identification

- Créer *l'Activity Identification* qui pour l'instant ne contient qu'un bouton d'id *testcon* qui lance *l'asynctask Connexion*

Afin de pouvoir accéder à Internet vous devez rajouter dans le *manifest* :
<uses-permission android:name="android.permission.INTERNET" />

- Modifiez la *MainActivity* pour pouvoir appeler *l'Activity Identification* sur le clic de l'image identification.

Code

- Ajoutez les variables :

```
private Button mBouton = null;
private AsyncTask<String, String, Boolean> mThreadCon = null;
```

- Ajoutez sur le clic du bouton *testcon* le code :

```
String[] mesparams = { "monidentifiant", "monpass","monurl" };
mThreadCon = new Connexion(Identification.this).execute(mesparams);// la
classe Connexion est la classe héritant de AsyncTask que nous allons créer
juste après
// le .execute permet de passer à la méthode connexion des paramètres qui
seront reçus via la méthode doInBackground
```

- Ajoutez la méthode permettant d'afficher les messages à partir de la classe connexion car cette dernière ne peut accéder directement à L'UI

```
public void alertmsg(String title, String msg) {
   AlertDialog.Builder adb = new AlertDialog.Builder(this);
   adb.setTitle(title);
   adb.setPositiveButton("Ok", null);
   adb.setMessage(msg);
   adb.show();
}
```

- Ajoutez la méthode permettant de gérer le retour après la fin de l'*asynctask*

```
public void retourIdentification(StringBuilder sb)
{
      alertmsg("retour import", sb.toString());
}
```

Classe Connexion

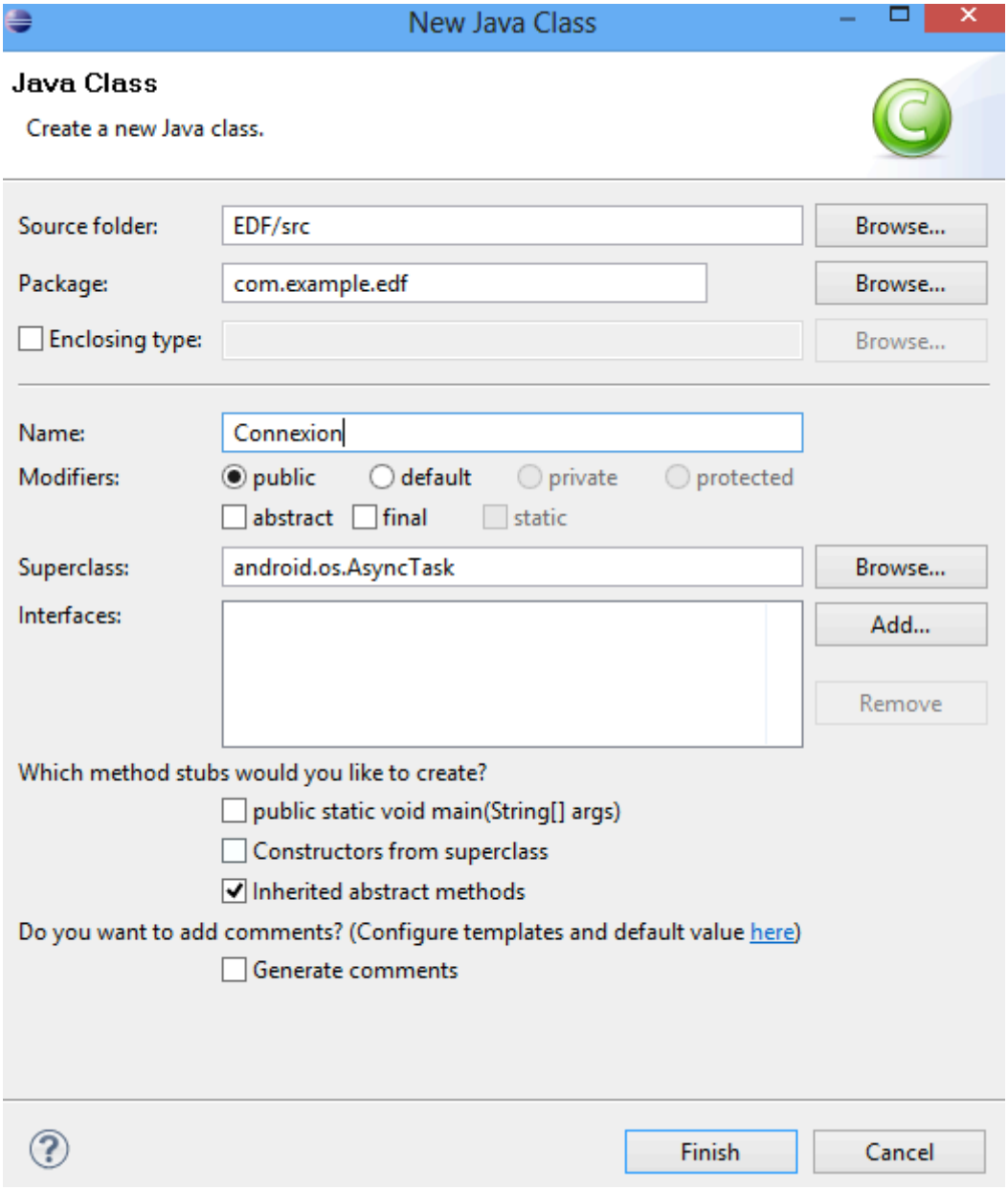
On ne va pas utiliser directement *AsyncTask*, mais plutôt créer une classe qui en dérivera. Cependant, il ne s'agit pas d'un héritage évident puisqu'il faut préciser trois paramètres :

- Le paramètre *Params* permet de définir le typage des objets sur lesquels on va faire une opération.
- Le deuxième paramètre, *Progress*, indique le typage des objets qui indiqueront l'avancement de l'opération.
- Enfin, *Result* est utilisé pour symboliser le résultat de l'opération.

Ce qui donne dans le contexte :

```
public class MaClasse extends AsyncTask<Params, Progress, Result>
```

- Créez la classe *Connexion*



Cette *asynctask* est faite de telle façon qu'elle pourra être appelée par n'importe quelle *Activity* en lui passant des paramètres différents selon l'activité.

- Modifiez la classe connexion à l'image de celle fournie en annexe afin d'avoir une classe générique qui nous servira aussi dans les *Activity Import* et *Export*.

- Testez l'appel à la classe *Connexion*

Nous allons essayer de voir si les paramètres envoyés à la classe *Connexion* ont bien été reçus.

- Modifiez le code de `doInBackground` afin de recevoir les paramètres lors de l'appel

```
String vid = "", vpass = "", vurl = "";
if (vclassactivity.contains("Identification")) {
   vid = params[0];
   vpass = params[1];
   vurl = params[2];
}
```

- Modifiez le code de `doInBackground` afin d'afficher les paramètres reçus via l'appel à `publishProgress`
- Testez l'appel à la classe *Connexion* et le passage des paramètres

Connexion à un serveur internet

Test de la connexion

- Créez la base de données à partir de edf.sql (fichier fourni) ;
- Lancez un serveur apache qui simulera le vrai serveur de EDF (ici Wamp, mais il serait préférable de créer un serveur Apache sur la ferme, ou via un hébergeur gratuit) ;
- Modifiez le *httpd.conf* de Apache afin d'inscrire un répertoire dans lequel vous aurez créé le dossier EDF

```
<Directory "c:/?????/EDF/">
    Options Indexes FollowSymLinks MultiViews
    AllowOverride all
        Order allow,deny
    Allow from all
</Directory>
```

- Créez un fichier *connect.php* sous le répertoire spécifié

```
<?php
echo 'test serveur ok';
?>
```

- Récupérez l'adresse IP de votre serveur et tester la connexion à http://**xxx.xxx.xxx.xxx**/edf/connect.php à partir de votre STA (tablette ou téléphone)

  Si non autorisé dans *httpd.conf*, commentez la ligne
  ServerName localhost

```
#
# Deny access to the entirety of your server's filesystem. You must
# explicitly permit access to web content directories in other
# <Directory> blocks below.
#
<Directory />
   AllowOverride All
#    Require all granted
</Directory>
```

Asyntask et HttpURLConnection

- Modifiez la méthode *doInBackgroung* avec le code qui permet de se connecter à un serveur (cf. annexe) en remplaçant la simulation de la réception d'un serveur
- Modifiez *connect.php* :

Avec réception des paramètres JSON et envoi du résultat. Exemple : une requête SQL encodée en JSON avec, pour chaque ligne, un encodage en utf8 pour les accents.

Attention : pour les dates, ici le format est yyyy-mm-dd.

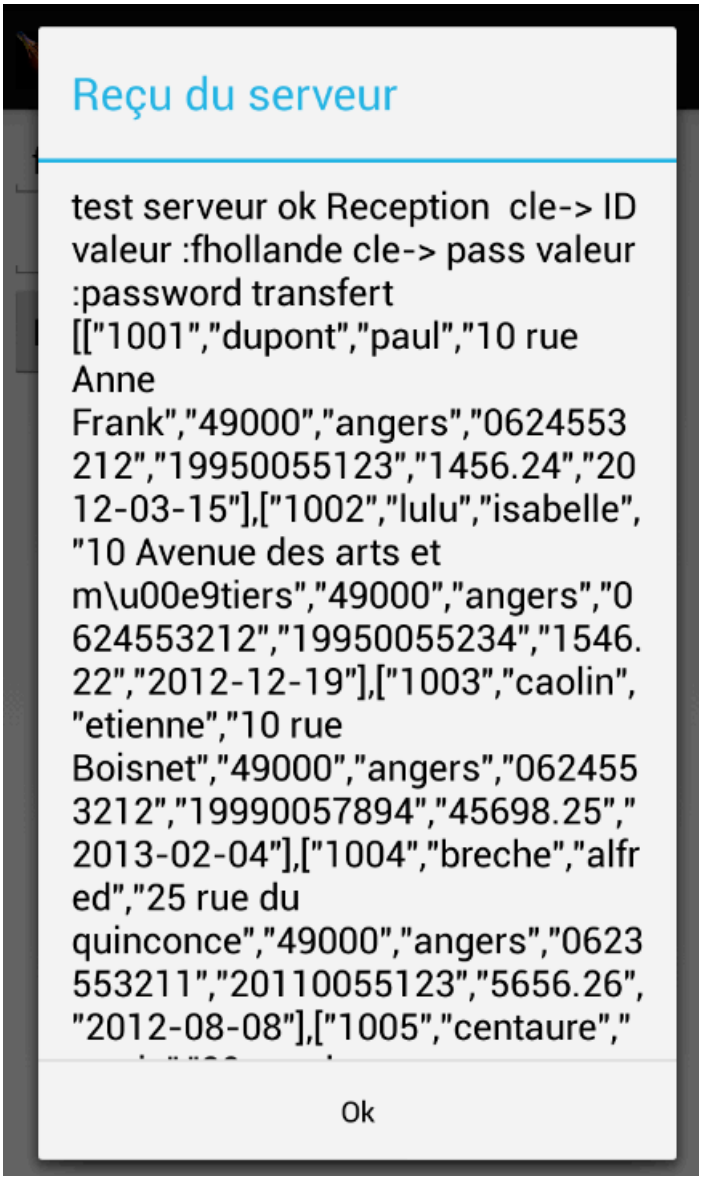
```
<?php
function utf8($n) {
        return (utf8_encode($n));
        }

$data = json_decode(file_get_contents("php://input"),true);
$recu='test serveur ok Reception ';
foreach ($data as $key => $value){
$recu=$recu.' cle-> '.$key.' valeur :'.$value;
            }
$recu=$recu.' transfert ';
$id=$data['ID'];
$pass=$data['pass'];
$host="localhost";
$username="root";
$password="";
$db_name="EDF";
$con=mysql_connect("$host", "$username", "$password")or die("cannot
connect");
mysql_select_db("$db_name") or die ("cannot select DB");
$sql = "SELECT
identifiant,nom,prenom,adresse,codePostal,ville,telephone,idCompteur,ancien
Releve,dateAncienReleve FROM client ,controleur where idcontroleur=id and
id='".$id."' and mp=md5('".$pass."')";
$result = mysql_query($sql);
$json = array();
if(mysql_num_rows($result)){
while($row= mysql_fetch_assoc ($result)){
$json[]=array_map('utf8',$row);
}
}
mysql_close($con);
$recu=$recu.json_encode($json);
echo $recu;
?>
```

- Modifiez les paramètres d'appel de l'activity identification par

String[] mesparams = { "fhollande", "password","http://***xxx.xxx.xxx.xxx***/edf/connect.php" };

- Testez votre application

Exemple de résultat obtenu :



Ici, réception d'une chaine de caractères (objet JSON) composé de cinq éléments JSON.

ANNEXE : Exemple de code

Connexion avec le serveur

Exemple de classe générique héritant de AsyncTask

```
import java.lang.ref.WeakReference;
import android.app.Activity;
import android.os.AsyncTask;
import android.widget.Toast;

public class Connexion extends AsyncTask<String, String, Boolean> {
      // Référence à l'activité qui appelle
      private WeakReference<Activity> activityAppelante = null;
      private String classActivityAppelante;
      private StringBuilder stringBuilder = new StringBuilder();

    public Connexion (Activity pActivity) {
      activityAppelante = new WeakReference<Activity>(pActivity);
      classActivityAppelante = pActivity.getClass().toString();
      }

    @Override
    protected void onPreExecute () {// Au lancement, on envoie un message à
l'appelant
      if (activityAppelante.get() != null)
                  Toast.makeText(activityAppelante.get(), "Thread on démarre",
                              Toast.LENGTH_SHORT).show();
    }

    @Override
    protected void onPostExecute (Boolean result) {
      if (activityAppelante.get() != null) {
            if (result) {
                        Toast.makeText(activityAppelante.get(), "Fin ok",
                                    Toast.LENGTH_SHORT).show();
//pour exemple on appelle une méthode de l'appelant qui va gérer la fin ok du thread
                  if (classActivityAppelante.contains("Identification"))
                  {
                        ((Identification)
activityAppelante.get()).retourIdentification (stringBuilder);
                  }
            }
            else
                  Toast.makeText(activityAppelante.get(), "Fin ko",
                                    Toast.LENGTH_SHORT).show();         }
    }

    @Override
    protected Boolean doInBackground (String... params) {// Exécution en arrière plan
      // on simule ici une activité de 2 sec ne sert à rien
      try {
            Thread.sleep(2000);
      }catch(InterruptedException e) {
        return false;
      }

      stringBuilder.append("Retour vers l'activity appelante");//simule une réception
d'un serveur
      String[] vstring={ "Thread","appel du doInBackground" };
      publishProgress(vstring);
        return true;
    }
```

```
    @Override
    protected void onProgressUpdate (String... param) {
      // utilisation de on progress pour afficher des message pendant le
doInBackground
      // ici pour exemple on appelle une méthode de l'appelant qui peut par exemple
modifier son layout
      if (classActivityAppelante.contains("Identification"))
      {
            ((Identification) activityAppelante.get()).alertmsg(param[0],param[1]);
      }
    }

    @Override
    protected void onCancelled () {
      if(activityAppelante.get() != null)
        Toast.makeText(activityAppelante.get(), "Annulation",
Toast.LENGTH_SHORT).show();
    }
}
```

Connexion au serveur avec *HttpURLConnection* et passage de paramètres en JSON

```
      HttpURLConnection urlConnection = null;
            try {
                  URL url = new URL(vUrl);
                  urlConnection = (HttpURLConnection) url.openConnection();
                  urlConnection
                              .setRequestProperty("Content-Type",
"application/json");
                  urlConnection.setRequestProperty("Accept", "application/json");
                  urlConnection.setRequestMethod("POST");
                  urlConnection.setDoOutput(true);
                  urlConnection.setConnectTimeout(2000);
                  OutputStreamWriter out = new OutputStreamWriter(
                              urlConnection.getOutputStream());
      // selon l'activity appelante on peut passer des paramètres en JSON exemple
      if (classActivityAppelante.contains("Identification"))
      {
            // Création objet jsonn clé valeur
            JSONObject jsonParam = new JSONObject();
            // Exemple Clé valeur utiles à notre application
            jsonParam.put("ID", vid);
            jsonParam.put("pass", vpass);
            out.write(jsonParam.toString());
            out.flush();
      }
            out.close();
            // récupération du serveur
            int HttpResult = urlConnection.getResponseCode();
                  if (HttpResult == HttpURLConnection.HTTP_OK) {
                        BufferedReader br = new BufferedReader(new InputStreamReader(
                                    urlConnection.getInputStream(), "utf-8"));
                        String line = null;
                        while ((line = br.readLine()) != null) {
                              stringBuilder.append(line);
                        }
                        br.close();
                        String[] vstring0 = { "Reçu du serveur",
                                    stringBuilder.toString() };

                        publishProgress(vstring0);
                  } else {
                        String[] vstring0 = { "Erreur",
                                    urlConnection.getResponseMessage() };
```

```
            publishProgress(vstring0);
      }
} catch (MalformedURLException e) {

      String[] vstring0 = { "Erreur", "Pbs url" };
      publishProgress(vstring0);
      return false;
} catch (java.net.SocketTimeoutException e) {
      String[] vstring0 = { "Erreur", "temps trop long" };
      publishProgress(vstring0);
      return false;
} catch (IOException e) {

      String[] vstring0 = { "Erreur", "Pbs IO" };
      publishProgress(vstring0);
      return false;
} catch (JSONException e) {
      String[] vstring0 = { "Erreur", "Pbs json" };
      publishProgress(vstring0);
      return false;
} finally {
      if (urlConnection != null)
            urlConnection.disconnect();
}
```