PARTIE 1: INSTALLATION ET PARAMETRAGE DE L'ENVIRONNEMENT ANDROID

1. PRE-REQUIS

1.1. RECUPERATION DE ECLIPSE + ADT ET DU SDK ANDROID

- Aller sur le site http://developer.android.com/sdk/index.html pour récupérer le zip.
- Ce zip contient le SDK Android, et Eclipse avec le plugin ADT

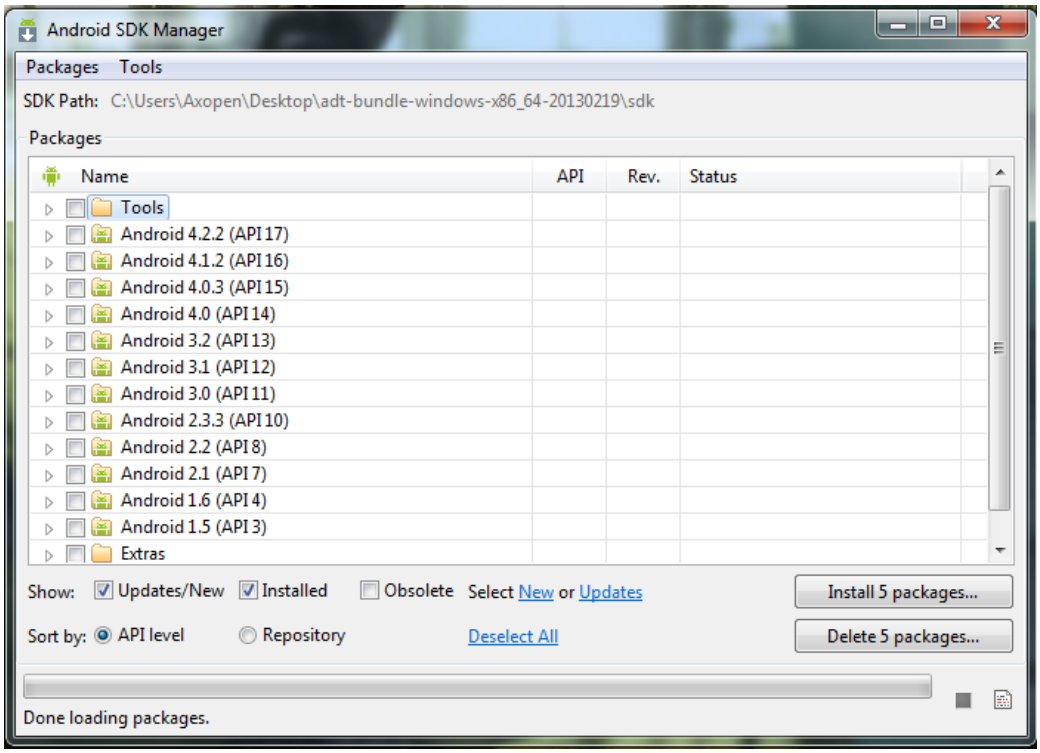
1.2. INSTALLATION DU JAVA DEVELOPPEMENT KIT

- Récupérer la version 6 du JDK. Le JDK (Java Development Kit), qui contient le JRE (afin d'exécuter les applications Java), mais aussi un ensemble d'outils pour compiler et déboguer votre code

2. PARAMETRAGE

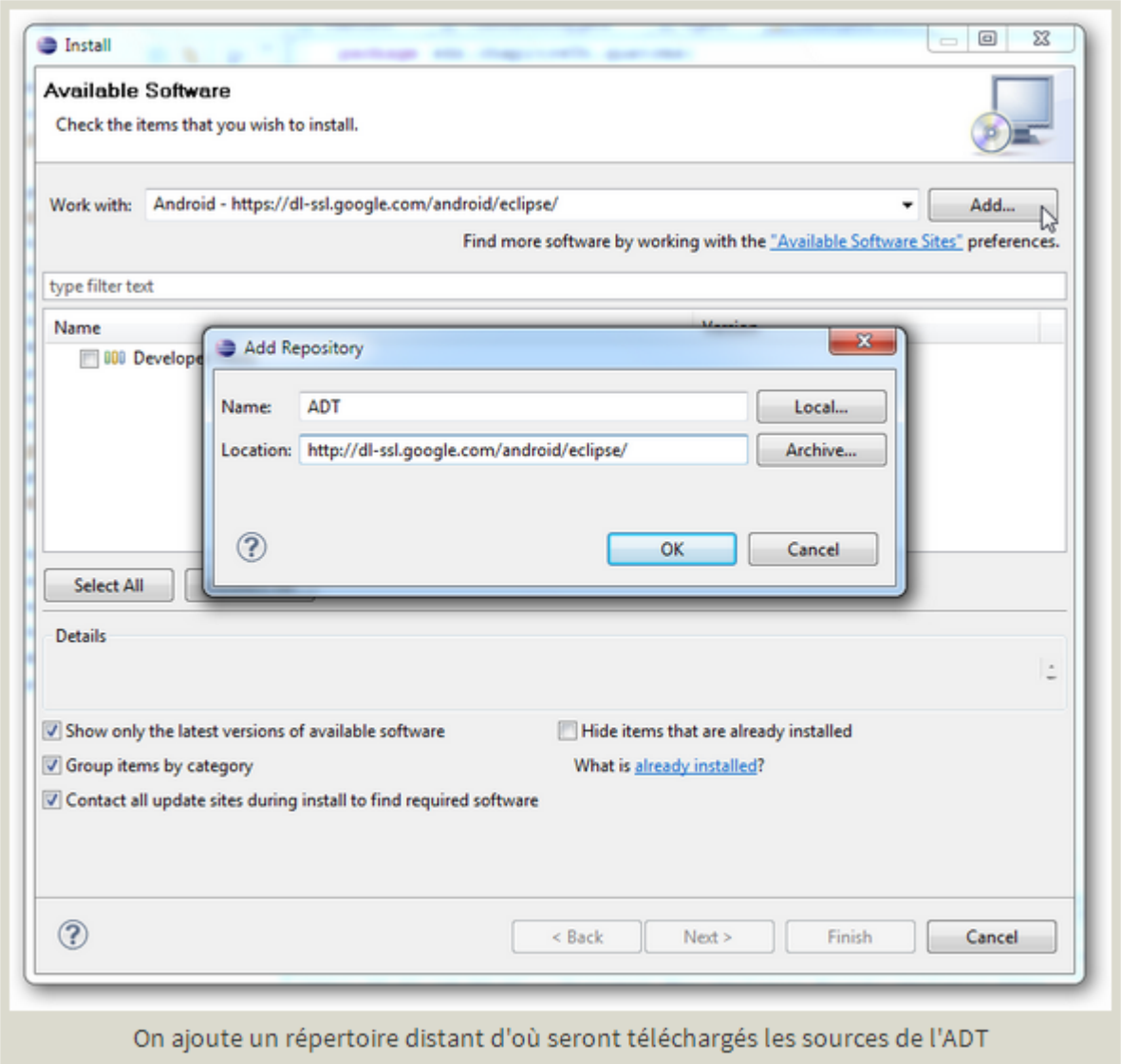
2.1. LE SDK MANAGER

- Celui-ci vous permet de télécharger ce dont vous avez besoin
- Sélectionnez les Tools, les Extras, et la version pour laquelle vous voulez développer votre application
- (Les API dont le numéro est compris entre 11 et 13 sont théoriquement destinées aux tablettes graphiques. En théorie, vous n'avez pas à vous en soucier, les applications développées avec les API numériquement inférieures fonctionneront, mais il y aura des petits efforts à fournir en revanche en ce qui concerne l'interface graphique)



2.2. L'IDE ECLIPSE

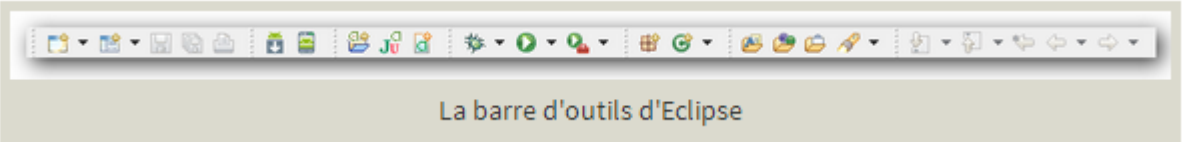
- Nous choisissons Eclipse : tout simplement parce qu'il est gratuit, puissant et recommandé par Google dans la documentation officielle d'Android. Vous pouvez aussi opter pour d'autres IDE compétents tels que IntelliJ IDEA, NetBeans avec une extension ou encore MoSync. Il existe d'ailleurs un nouvel IDE développé par Google disponible ici : http://developer.android.com/sdk/installing/studio.html
- Si vous avez déjà Eclipse, il vous faut ADT :
  Allez dans Help puis dans Install New Softwares… (installer de nouveaux programmes). Au premier encart intitulé Work with: cliquez sur le bouton Add… qui se situe juste à côté. On va définir où télécharger ce nouveau programme. Dans l'encart Name écrivez par exemple ADT et, dans location, mettez cette adresse https://dl-ssl.google.com/android/eclipse/, comme à la figure suivante. Avec cette adresse, on indique à Eclipse qu'on désire télécharger de nouveaux logiciels qui se trouvent à cet emplacement, afin qu'Eclipse nous propose de les télécharger. Cliquez ensuite sur OK



On ajoute un répertoire distant d'où seront téléchargés les sources de l'ADT

2.3. L'EMULATEUR DE TELEPHONE : ANDROID VIRTUAL DEVICE

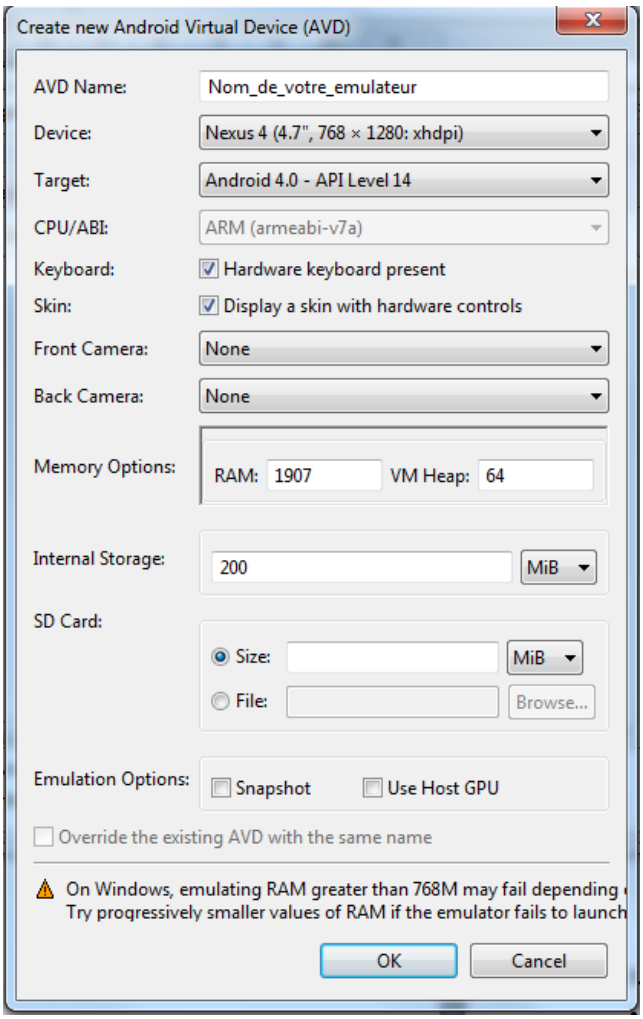
- Cet outil est utile si vous n'avez pas de terminal Andoid pour développer
- Il vous permet d'en émuler un et de tester vos applications
- Ouvrez Eclipse repérez tout d'abord où se trouve la barre d'outils, visible à la figure suivante



La barre d'outils d'Eclipse

- Dans la figure ci-dessous l'icône de gauche permet d'ouvrir les outils du SDK et celui de droite permet d'ouvrir l'interface de gestion d'AVD. Cliquez dessus puis sur New… pour ajouter un nouvel AVD

- Dans cette nouvelle fenêtre vous devez définir les paramètres de votre ADV
- Tout d'abord le nom, puis quelle téléphone vous voulez émuler (vous pouvez aussi choisir une taille d'écran pour avoir une généricité), et enfin la version d'Android sur laquelle tournera votre ADV



2.4. POUR UN VRAI TERMINAL

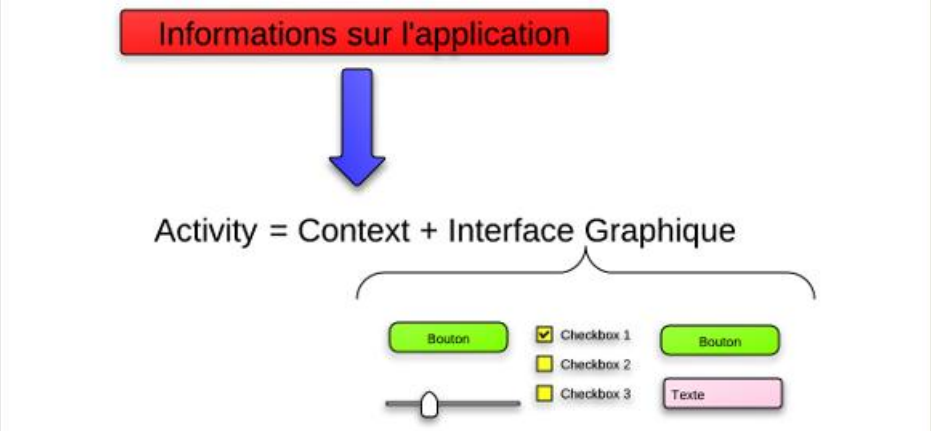
- Tout naturellement, vous devez configurer votre téléphone comme on a configuré l'émulateur. En plus, vous devez indiquer que vous acceptez les applications qui ne proviennent pas du Market dans Configuration > Application > Source inconnue.
- Pour Windows vous devez télécharger les drivers, pour Mac vous n'avez rien à faire, pour Linux cela dépend des terminaux

PARTIE 2: LE DEVELOPPEMENT SOUS ANDROID

1. ACTIVITE ET VUE

1.1 ACTIVITE

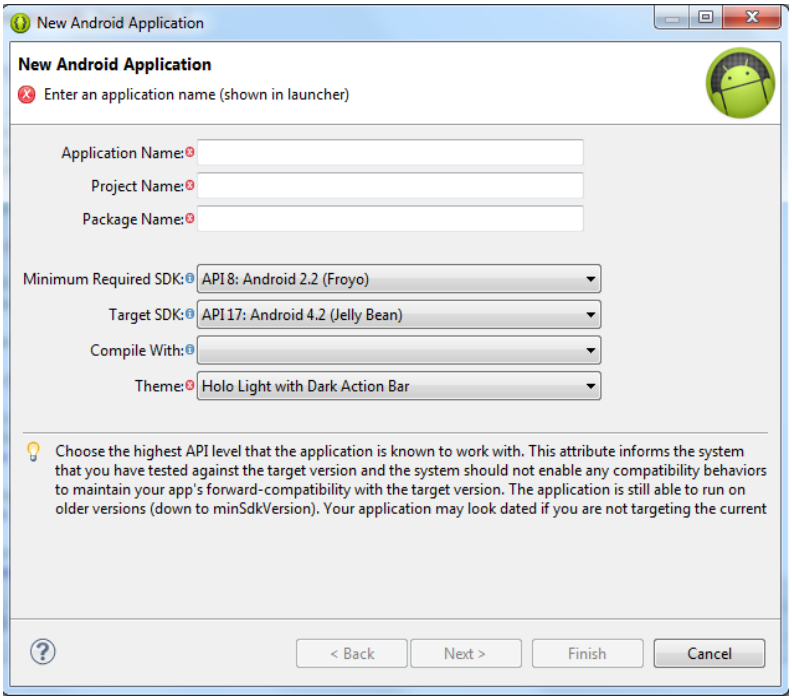
- Une activité est pour expliquer simplement : une page
- Il n'y a qu'une seule activité par pages, chaque activité est donc relié à un écran (voir plus bas)
- Une activité contient des informations sur l'état actuel de l'application : ces informations s'appellent le context. Ce context constitue un lien avec le système Android ainsi que les autres



activités de l'application, comme le montre la figure suivante

2. CREATION D'UNE APPLICATION

2.1. CREATION D'UN PROJET



- Tout d'abord, vous pouvez choisir le nom de votre application avec Application name. Il s'agit du nom qui apparaîtra sur l'appareil et sur Google Play pour vos futures applications
- Project name est le nom de votre projet pour Eclipse. Ce champ n'influence pas l'application en elle-même, il s'agit juste du nom sous lequel Eclipse la connaîtra
- Il faudra ensuite choisir dans quel package ira votre application
- La liste suivante, Minimum Required SDK, est un peu plus subtile. Elle vous permet de définir à partir de quelle version d'Android votre application sera visible sur le marché d'applications. Ce n'est pas parce que vous compilez votre application pour l'API 7 que vous souhaitez que votre application fonctionne sous les téléphones qui utilisent Android 2.1, vous pouvez très bien viser les téléphones qui exploitent des systèmes plus récents que la 2.2 pour profiter de leur stabilité par exemple, mais sans exploiter les capacités du SDK de l'API 8. De même, vous pouvez très bien rendre disponibles aux utilisateurs d'Android 1.6 vos applications développées avec l'API 7 si vous n'exploitez pas les nouveautés introduites par l'API 7, mais c'est plus complexe
- La liste Target SDK vous permet de viser une version en particulier pour votre application
- La liste Compile With vous permet de choisir pour quelle version du SDK vous allez compiler votre application
- Le thème n'est pas important
- Les pages suivantes sont intuitifs elles servent à la personnalisation de l'application

3. LES ECRANS

3.1. LES RESSOURCES

- Celles-ci sont écris en XML, voir le fonctionnement ci-dessous

```xml
<?xml version="1.0" encoding="utf-8"?>
<bibliotheque>
  <livre style="fantaisie">
    <auteur>George R. R. MARTIN</auteur>
    <titre>A Game Of Thrones</titre>
    <langue>klingon</langue>
    <prix>10.17</prix>
  </livre>
  <livre style="aventure">
    <auteur>Alain Damasio</auteur>
    <titre>La Horde Du Contrevent</titre>
    <prix devise="euro">9.40</prix>
    <recommandation note="20"/>
  </livre>
</bibliotheque>
```

- Les différents types de ressources :

Type	Description
Dessin et image (res/drawable)	On y trouve les images matricielles (les images de type PNG, JPEG ou encore GIF) ainsi que des fichiers XML qui permettent de décrire des dessins simples (par exemple des cercles ou des carrés).
Mise en page ou interface graphique (res/layout)	Les fichiers XML qui représentent la disposition des vues (on abordera cet aspect, qui est très vaste, dans la prochaine partie).
Menu (res/menu)	Les fichiers XML pour pouvoir constituer des menus.
Donnée brute (res/raw)	Données diverses au format brut. Ces données ne sont pas des fichiers de ressources standards, on pourrait y mettre de la musique ou des fichiers HTML par exemple.
Différentes variables (res/values)	Il est plus difficile de cibler les ressources qui appartiennent à cette catégorie tant elles sont nombreuses. On y trouve entre autre des variables standards, comme des chaînes de caractères, des dimensions, des couleurs, etc.

3.1.1. Points particuliers des ressources

- Les shapes à mettre dans res/drawable permettent de définir plusieurs choses, des formes pour des boutons ou autres éléments, des dégradés de couleurs. C'est un peu l'idée du CSS on sépare le design des éléments.
  Plus d'informations sur ce lien :
  http://developer.android.com/guide/topics/resources/drawable-resource.html

- Les res/values : il est très important de définir tous ces Strings ici pour les utiliser dans ses Layouts.
  http://developer.android.com/guide/topics/resources/string-resource.html
- Il est aussi utile d'y ajouter les couleurs pour utiliser un nom et pas un code, pour une meilleure lisibilité. (Dans les dernières versions de l'API il existe un res /color)
  http://developer.android.com/guide/topics/resources/color-list-resource.html
- ET enfin Style, ce fichier permet de définir des styles, par exemple un TextView avec ce style aura toujours la même forme ce qui est utile puisque nous n'avons pas besoin de réécrire les différents paramètres à chaque fois.
  http://developer.android.com/guide/topics/resources/style-resource.html

- Les menus: On place des items qui correspondent aux boutons menus.
  http://developer.android.com/guide/topics/ui/menus.html

3.2. LES LAYOUTS

- Les Layouts sont écrits en XML, puisque ce sont des ressources.
- Ils servent à placer des éléments sur l'écran (boutons, champs de texte, images…)

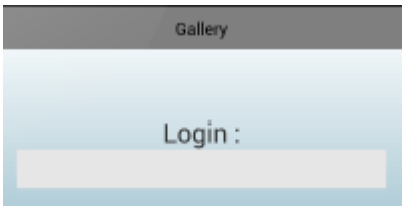
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/background_login"
    android:orientation="vertical" >

    <Gallery
        android:id="@+id/axopenImg"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@drawable/logo_axopen"
        android:gravity="center_horizontal" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="60dip"
        android:gravity="center"
        android:text="Login :"
        android:textSize="25sp" />

    <EditText
        android:id="@+id/login"
        android:layout_width="fill_parent"
        android:layout_height="35dip"
        android:layout_marginLeft="15dip"
        android:layout_marginRight="15dip"
        android:background="@color/fond_gris"
        android:inputType="textNoSuggestions|none" />
```

- Voici par exemple le début de la page Login d'une application, on voit un LinearLayout qui prend toute la page avec un background qui est défini dans une shape dans les drawables.
- Ensuite, nous insérons une image en tant que logo.
- Après vient le champ de texte « Login », celui –ci est définit dans "@string/login".
- Et enfin le champ pour entrer le password.
  (L'attribut android:inputType= "textNoSuggestions|none" sert à cacher le clavier à l'ouverture de l'activité).



Voilà le rendu de ce Layout

3.3. L'ACTVITE

Notre classe Java doit être extends Activity
Voici un exemple de la page de Login vu précédemment :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.login);  //  Cette méthode appelle le Layout fait plus haut afin de
l'afficher

    pseudo = (EditText) findViewById(R.id.login); // Celle-ci permet de d'associer le champs pour
écrire le texte à une variable

    button.setOnClickListener(new View.OnClickListener() { // Si nous avions définit une variable
button, au clique nous aurions effectué le code de cette fonction

        @Override
        public void onClick(View v) {
            Intent intent = new Intent(LoginActivity.this,  ChoixActivity.class);
            startActivity(intent); // Ce code va appeler l'activité ChoixActivity si on clique sur le bouton
        }

    });

}

@Override
public boolean onCreateOptionsMenu(Menu menu) { // Cette méthode va créer le menu «  menumain »
qui est dans res/menu

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menumain, menu);

    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) { // Si on clique sur l'item (qui est
déconnexion) on fait un finsh()

    finish();// Cette instruction sert à fermer une activité, comme on est sur la première on ferme
l'application
    return true;

}
```

4. LE MANIFEST

4.1. VERSION ANDROID

- Nous allons pouvoir modifier ici la version d'android vise et celle requise pour voir l'application sur le Play Store

```
<uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="14" />
```

4.2. LES PERMISSIONS

- Avec ceci nous allons pouvoir ajouter des permissions à notre application afin d'avoir accès par exemple à la carte SD, à internet, à la camera… Si vous utilisez un périphérique interne regardez bien vos permissions si une erreur est affichée

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

4.3. DETAILS DE L'APPLICATION

- La balise application permet de définir un icône pour l'application définit dans les ressources drawables, le nom de l'application définit dans les Strings, un thème pour l'application dans les styles

```
<application
    android:allowBackup="true"
    android:icon="@drawable/axopen"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
```

4.4. ACTIVITE PRINCIPALE

- Ceci correspond à l'activité principale, on peut donc changer le chemin dans le package, définir une orientation obligatoire…

```
<activity
        android:name="com.axopen.mobile.activity.LoginActivity"
        android:screenOrientation="portrait"
        android:label="@string/app_name"
        android:windowSoftInputMode="stateHidden" >
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />

          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```

Attention : chaque activité doit être définit dans le Manifest, sinon le projet ne compile pas.

PARTIE 3 : LES WEB SERVICES

1. CREATION D'UN WEB SERVICE

1.1. APPLICATION

Nous avons besoin d'une classe extends Application pour créer notre web service :

```
@ApplicationPath("webservice") // path de la la web application
public class WebApplication extends Application {

}
```

1.2. LA CLASSE DU WEB SERVICE

```
@Produces("application/json; charset=UTF-8") // ce que le client va envoyer
@Path("/login") // path du web-service
@Stateless    // pas de session
public class LoginWebservice {

    @Inject    // on inject le manager
    private CompteManager mCompteManager;

    @POST // on précise que c'est du post
    @Path(value = "ident") // path de la methode

    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)

    public Response login(UserWSO pUser) { // nous verrons plus tard l'utilité des WSO
        /* code java pour vérifier si le user existe et récupéré ses droits */

                int result[] = { lCompte.getIdCompte(), role };

                return Response.status(200).entity(result).build(); // on renvoie le tableau de valeurs

    }

}
```

2. L'ENVOIE

2.1. UNE ASYNCTASK

```
public class Connection extends AsyncTask<String, Void, Integer[]> { /* le premier type c'est pour l'entrée, le
second pour l'exécution, et le troisième pour le résultat */


  public Connection() {
    /* constructeur */
  }
```

```
    @Override
    protected void onPreExecute() {
    }

    protected void onPostExecute(Integer result) {
        /* ex : afficher un résultat */
    }

    protected Integer[] doInBackground(String... param) {

            /* on récupère les paramètres */
            String pass = param[1];
            String name = param[0];

            /* partie pour la connection en POST */
            HttpClient httpclient = new DefaultHttpClient();

            HttpPost httppost = new HttpPost("adresse-du-serveur/path-de-la-webApplication/path-dans-le-web-
service/path-de-la-method");

            /* passage de paramètres avec des JsonObject */
            JSONObject jsonObject = new JSONObject();

            jsonObject.put("login", name);
            jsonObject.put("password", pass);

            StringEntity se = new StringEntity(jsonObject.toString());

            httppost.setEntity(se);

            httppost.setHeader("Content-type", "application/json");

            /* récupération de la réponse du serveur */
            HttpResponse response = httpclient.execute(httppost);

            final int lStatusCode = response.getStatusLine().getStatusCode();

            if (lStatusCode != HttpStatus.SC_OK) {
                Log.w("", "Error " + lStatusCode + " for URL " + httppost);
                return null;
            } // Execute HTTP Post Request

            Gson lGson = new Gson(); // objet GSON pour caster en ce qu'on veut plus tard

            HttpEntity getResponseEntity = response.getEntity();

            Reader lReader = new InputStreamReader(
                    getResponseEntity.getContent());

            /* on utilise un tableau pour récupérer les réponses du serveur */
            Integer[] lCompte = lGson.fromJson(lReader, Integer[].class);
```

```
            return lCompte;
    }
}
```

2.2. UNE ACTIVITE

```
Connection conn = new Connection(); // on créé une nouvelle connexion

AsyncTask<String, Void, Integer[]> resultat = conn.execute(login,  mdp); // on créé l'asynktask resultat, on execute notre connexion en passant le login et le mot de passé en paramètres.

Integer[] result = resultat.get(); // on récupère le résultat

if(result != null){
        int res 1 = result[0];
        int res 2 = result[1]; // on place les résultats dans des variables
}
```

2.3. COTE SERVEUR

Pré-requis :

- Nous allons créer des Classes WSO
- Les attributs de ces classes auront les mêmes noms et types que ceux mis dans le JsonObject
- Nous allons placer un ObjetWSO en paramètre de notre web service
- Le JsonObject remplira automatiquement les atttributs de notre objet mis en paramètre de notre web service
- Nous pourrons ainsi, avoir accès à ce qui a été envoyé.

2.3.1. Les WSO pour L'envoi de données au client

Côté serveur nous avons besoin de créer des BO spéciaux (WSO) pour les web services.
Pourquoi ?
Car dans un premier temps cela nous permettra d'avoir les deux mêmes BO côté client et côté serveur.
Dans un second temps, ce BO est indépendant des autres et n'aura donc pas besoin d'être modifié.
Et pour finir, étant donné que nous n'avons pas besoin de toutes les données des BO nous allons sélectionner seulement celles qui sont utiles.

BoWSO

```java
public class NdfHorsContratWSO implements Serializable {

    private static final long serialVersionUID = 1L;

    private int idHorsContrat;

    private String libelle;

    public NdfHorsContratWSO() {

    }

    public int getIdHorsContrat() {
        return idHorsContrat;
    }

    public void setIdHorsContrat(int pIdHorsContrat) {
        idHorsContrat = pIdHorsContrat;
    }

    public String getLibelle() {
        return libelle;
    }

    public void setLibelle(String pLibelle) {
        libelle = pLibelle;
    }

}
```

BO normal

```java
@Entity
@Table(name="demande_hors_contrat")
public class DemandeHorsContrat implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_hors_contrat")
    private int idHorsContrat;

    @Column(name="actif_heure_supp")
    private int actifHeureSupp;

    @Column(name="actif_imputation")
    private int actifImputation;

    @Column(name="actif_ndf")
    private int actifNdf;

    private String libelle;

    @OneToMany(mappedBy="horsContrat")
    private List<HeureSupp> heureSupps;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "horsContra
    private List<Imputation> imputations;

    @ManyToMany
    @JoinTable(name = "user_role_hors_contrat_masque", invers
    private List<UserRole> userRoles;

    public DemandeHorsContrat() {
    }

    public int getIdHorsContrat() {
        return this.idHorsContrat;
    }

    public void setIdHorsContrat(int idHorsContrat) {
        this.idHorsContrat = idHorsContrat;
    }

    public int getActifHeureSupp() {
        return this.actifHeureSupp;
    }

    public void setActifHeureSupp(int actifHeureSupp) {
        this.actifHeureSupp = actifHeureSupp;
    }
```

PARTIE 4 : REMARQUES

1. FERMER LES ACTIVITES

1.1. AU LANCEMENT DE L'ACTIVITE

Utiliser :

startActivityForResult(intent, 10);

Le 10 servira à fermer cette activité. On va donner un nombre différent par activité afin de nous y retrouver plus facilement.

Par exemple :

startActivityForResult(intent2, 20);

1.2. AU FINISH

Utiliser :

```
setResult(chiffre);
finish();
```

Le chiffre passé en paramètre de setResult servira à savoir ce que l'on a voulu faire dans l'activité du dessus (dans la pile). Puis avec le Finish on ferme l'activité en cours.

1.3. L'ANALYSE DU RESULTAT

Utiliser :

```
@Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        switch (requestCode) {
        case 10:
            if (resultCode == 1) {
                setResult(1);
                finish();
            } else if (resultCode == 2) {
            }
        case 20:
            if (resultCode == 1) {
                setResult(1);
                finish();
            } else if (resultCode == 2) {
            }
        default:

        } // switch
```

Dans cette partie le code est plus complexe :

- On fait un switch sur le résultat de l'activité d'au-dessus (dans la pile), envoyé avec la méthode startActivityForResult
- Si c'est requestCode 10 on se rappelle que c'est l'activité du 1.1. Si le resultCode est à 1 on envoie 1 à une activité plus basse et on ferme l'activité en cours. Si le resultCode est à 2 on ne fait rien, c'est-à-dire qu'on reste sur cette activité.

Dans quel cas utiliser ceci :
Imaginons que nous avons ouvert plusieurs activités dans l'application et que nous cliquons sur déconnexion. On veut donc retourner à la page de login.
On utilise donc le cas où le resultCode est à 1 jusqu'à arriver à la page login. Autrement dit on va fermer toutes les activités jusqu'à la page de login.
Quand nous sommes sur la page de login nous utilisons la méthode où le resultCode est à 2 pour rester sur cette activité.

(les chiffres utilisés sont utilisés par hasard vous avez entièrement le choix de mettre autre chose, ou même mieux d'utiliser des variables static pour comprendre plus facilement où vous en êtes).

2. PASSER DES CARACTERES SPECIAUX EN JSON

Il faut convertir les caractères ne UTF-8 :
Coté client :

```
public static String convert(String str)
{
    StringBuffer ostr = new StringBuffer();
    for(int i=0; i<str.length(); i++)
    {
        ostr.append("\\u");          // standard unicode format.
            String hex = Integer.toHexString(str.charAt(i) & 0xFFFF);     // Get hex value of the char.
            for(int j=0; j<4-hex.length(); j++) // Prepend zeros because unicode requires 4 digits
                ostr.append("0");
            ostr.append(hex.toLowerCase(Locale.FRENCH));
        //}
    }
return (new String(ostr));      //Return the stringbuffer cast as a string.
}
```

Côté serveur :

```
byte[] imageByteArray = Base64.decode(image);
String lDescription = photo.getDescription();
String str = lDescription.split(" ")[0];
str = str.replace("\\", "");
String[] arr = str.split("u");
String text = "";
for (int i = 1; i < arr.length; i++) {
    int hexVal = Integer.parseInt(arr[i], 16);
    text += (char) hexVal;
}
```

3. CACHER DES BOUTONS OU LES DESACTIVER

3.1. CACHER/AFFICHER

- Cacher :
  Element.setVisibility(View.GONE);
- Afficher :
  Element.setVisibility(View.VISIBLE);

3.2. ACTIVER/DÉSACTIVER

- Activer :
  Element.setEnabled(true);
- Désactiver :
  Element.setEnabled(false);

4. LES ADAPTER (LISTES DE TEXTVIEWS, LISTES DEROULANTES…)

Je vais prendre dans cette partie pour exemple une liste de TextView.

4.1. LE LAYOUT DE BASE

Nous pouvons observer dans ce code que nous ne mettons pas un textview mais une listView qui comportera nos textview, c'est un type un peu plus générique.

```
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" >

    <ListView
      android:id="@+id/listNdf"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent" >
    </ListView>
  </LinearLayout>
```

4.2. LE LAYOUT ASSOCIE A LA LISTEVIEW

Ce Layout sert à placer les éléments dans notre ListView, nous avons donc là 2 textView qui vont servir à afficher du texte, le troisième qui sera caché servira à stocker un id.

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    <TextView
      android:id="@+id/textNdf"
      android:layout_width="wrap_content"
      android:layout_height="match_parent"
      android:textIsSelectable="true" />
```

```
    <TextView
      android:id="@+id/textNdf2"
      android:layout_width="wrap_content"
      android:layout_height="match_parent"
      android:textIsSelectable="true" />

  </LinearLayout>

  <TextView
    android:id="@+id/idNdf"
    android:layout_width="0px"
    android:layout_height="0px"
    android:textIsSelectable="true" />
```

4.3. LE VIEWHOLDER

Le viewHolder va servir à faire le lien entre le java et le xml dans l'adapter.

```
public class ViewHolder {

    private TextView texte;
    private TextView id;
    private TextView text2;

    /* getters, setters et constructeurs */

}
```

4.4. L'ADAPTER

```
public class NdfAdapter extends ArrayAdapter<Ndf> { // il faut extends de ArraAdapter<type de nos données>

    private Ndf[] mData;
    private static LayoutInflater mInflater = null;
    private Context context;

    /* constructeur*/
    public NdfAdapter(Context pContext, Ndf[] pData) {
        super(pContext, 0, 0, pData);
        context = pContext;
        mData = pData;

        mInflater = (LayoutInflater) pContext
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {

        View lView = convertView;
        ViewHolderNDF lHolder; // on crée un objet Holder à partir de celui vu en 4.3
        if (convertView == null) {
```

```
            lHolder = new ViewHolderNDF();
            lView = mInflater.inflate(
                    com.axopen.mobile.R.layout.affichage_liste_ndf, null);
            /* on récupère le layout associé à la liste view */
            lHolder.setId((TextView) lView
                    .findViewById(com.axopen.mobile.R.id.idNdf));
            lHolder.setTexte((TextView) lView
                    .findViewById(com.axopen.mobile.R.id.textNdf));
            lHolder.setText2((TextView) lView
                    .findViewById(com.axopen.mobile.R.id.textNdf2));
            /* on associe les champs du layout au holder */
            lView.setTag(lHolder);
        } else {
            lHolder = (ViewHolderNDF) convertView.getTag();
        }

        Ndf lNdf = (mData[position]);
        String text =  " Date : \n Libellé : \n Statut : \n Montant TTC : \n Categorie : \n Contrat : ";
        lHolder.getTexte().setText(text);

        String text2 = " " + lNdf.getDateNdf() + "\n " + lNdf.getLibelle() + "\n " + lNdf.getStatut().getLibelle()
                + "\n " + lNdf.getFormatedTTC() + "\n " + lNdf.getCategorie() + "\n " + lNdf.getContrat();
        lHolder.getText2().setText(text2);
        /* on insert le text dans les textViews */
        lHolder.getId().setText(String.valueOf(lNdf.getIdNdf()));
        /* on place l'id dans le textview cache */

        return lView;

    }

    /* Cela sert à afficher la note de frais */
    @Override
    public View getDropDownView(int position, View convertView, ViewGroup parent) {
        TextView label = new TextView(context);
        Ndf lType = (mData[position]);
        String lTypeString = lType.getLibelle();
        label.setText("\n\t" + lTypeString + "\n");

        return label;
    }

/* getters, setters */

}
```

4.5. L'ACTIVITE

```
private void listeNdf() {
        try {
            mListView = (ListView) findViewById(R.id.listNdf); // on récupère notre listView

            RecupListeNdf recupLNdf = new RecupListeNdf();

            AsyncTask<Integer, Void, Ndf[]> listNdf = recupLNdf.execute(
                    CompteUser.getIdCompte(), moisImp); // on récupère une liste de NDF avec un web service
donc une asynktask

            Ndf[] Ndf;

            Ndf = listNdf.get(); // fin de la récupération

            if (Ndf != null) {
                mNdfAdapter = new NdfAdapter(this, Ndf); // on créer notre adapter
                mListView.setAdapter(mNdfAdapter); // on associe l'adapter à la listview

            }

        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (ExecutionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```