Androboum : Un projet pour apprendre la programmation Android

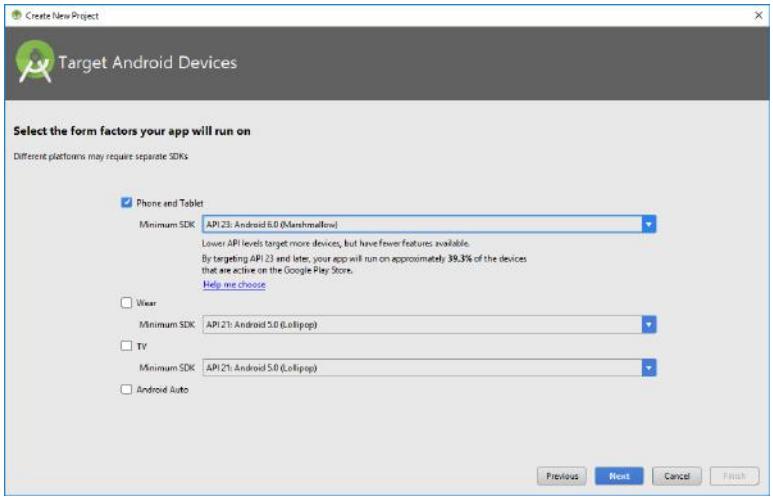
I. Table des matières

II. Mise en place

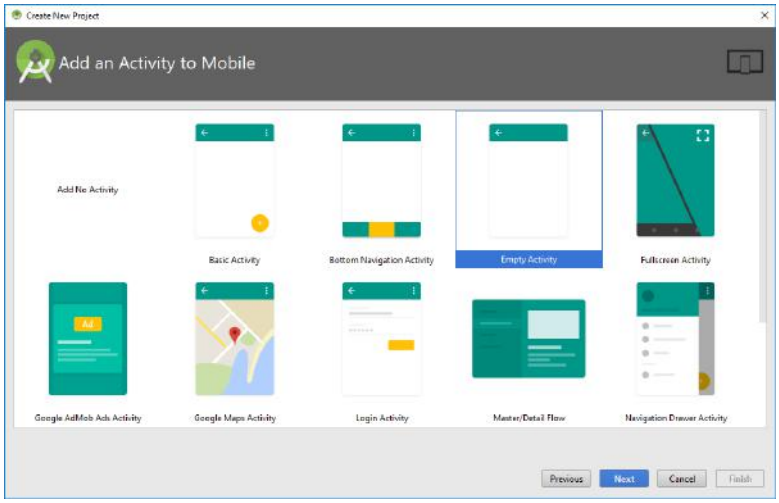
Objectif : Créer un nouveau projet et vérifier son fonctionnement correct, aussi bien dans l'émulateur, que sur un smartphone connecté à la machine de développement.

1. Un nouveau projet

Lancez Android Studio et créez un nouveau projet que vous appellerez « Androboum ». Vérifiez le répertoire de création de ce projet et lors de l'écran de sélection de l'API, choisissez une des dernières versions de l'API android (voir l'écran ci-dessous).

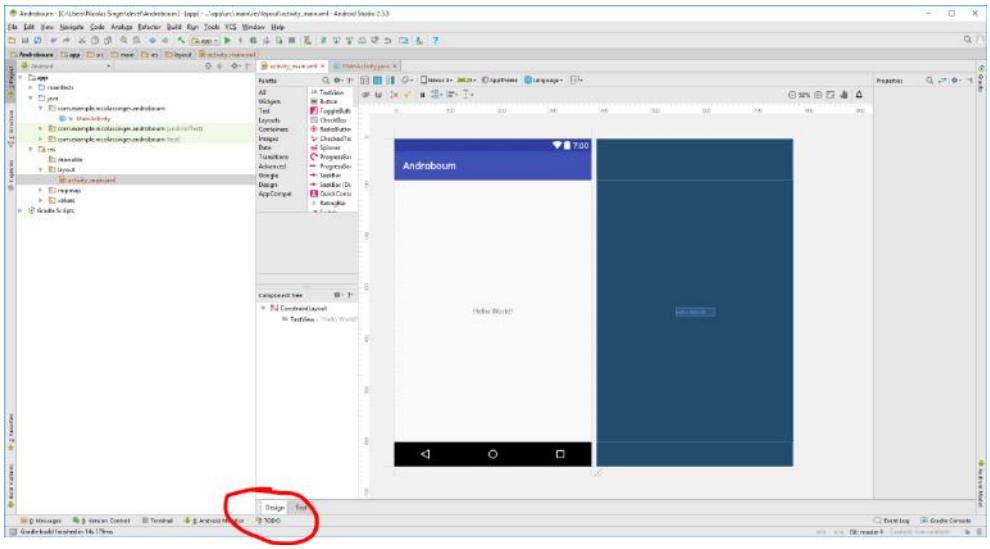


Sur l'écran qui suit choisissez de créer une « Empty Activity » pour démarrer avec une activité vide.



Sur l'écran suivant, laissez le nom de l'activité et du *layout* par défaut et cliquez sur « Finish ».

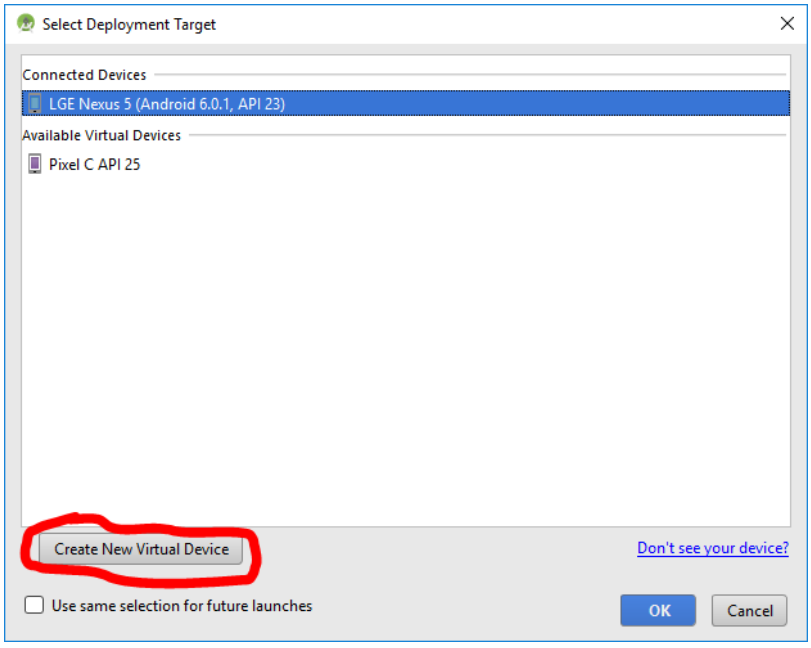
Le projet étant créé, vous pouvez naviguer jusqu'à aller chercher le *layout* de l'activité qui se trouve dans le fichier `app/res/layout/activity_main.xml`. Le mode « design » permet d'observer à quoi ressemblera l'interface de l'activité quand elle sera déployée sur l'émulateur ou un vrai appareil.



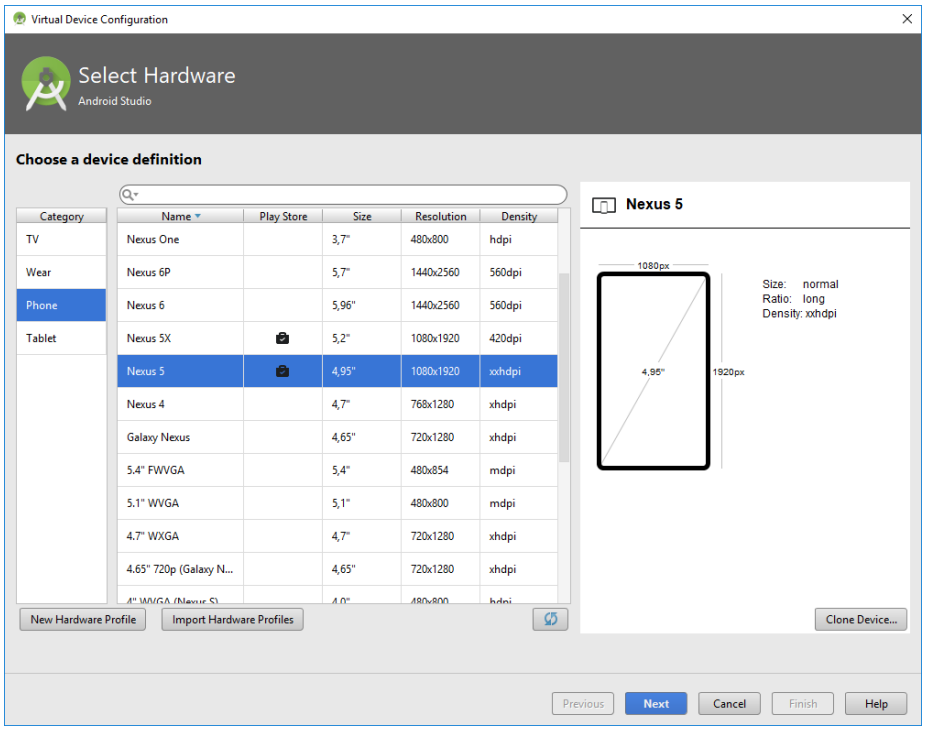
2. Déploiement sur un appareil virtuel (émulateur)

C'est le moment de tester que tout fonctionne en déployant cette application vers l'émulateur.

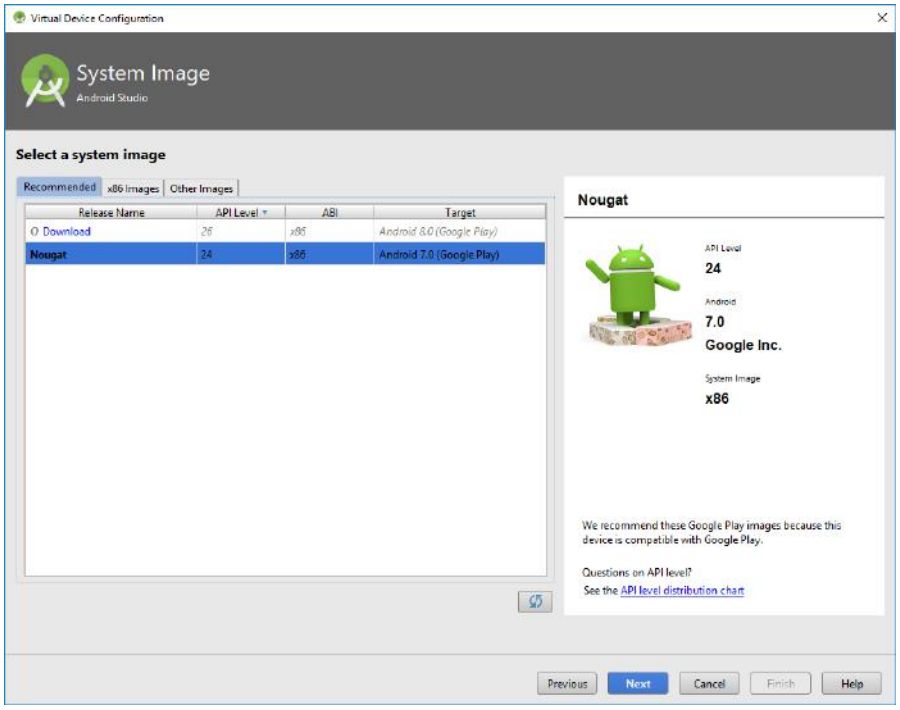
Utilisez le menu « Run/Run app » pour lancer le déploiement. L'écran qui apparait vous propose de choisir l'appareil qui sera la cible du déploiement. Si aucun émulateur n'apparaît, vous devez cliquer sur « Create New Virtual Device » pour en créer un.



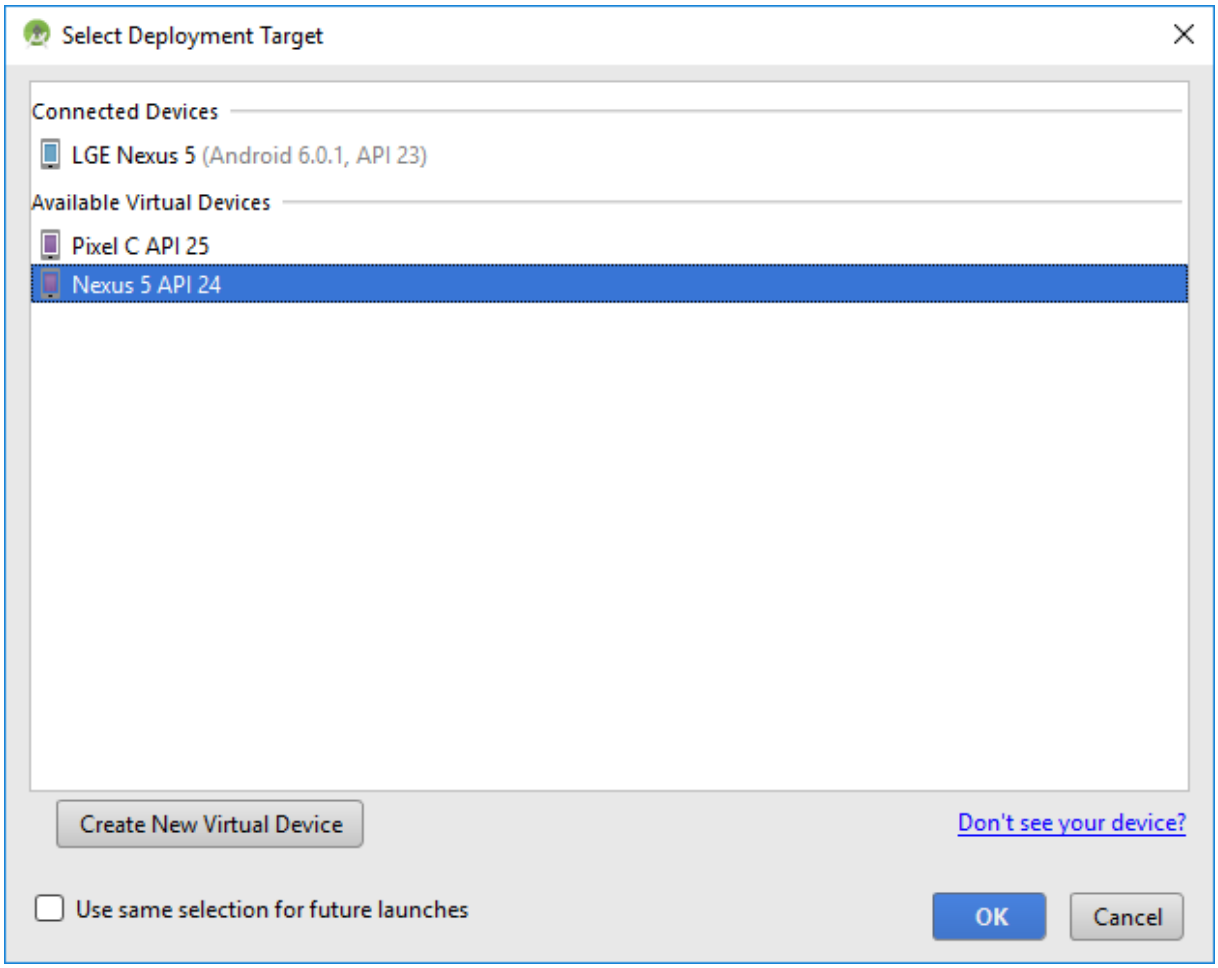
Choisissez d'émuler un appareil avec un écran de taille raisonnable afin de ne pas trop surcharger l'émulateur. Choisissez par exemple un Nexus 5.



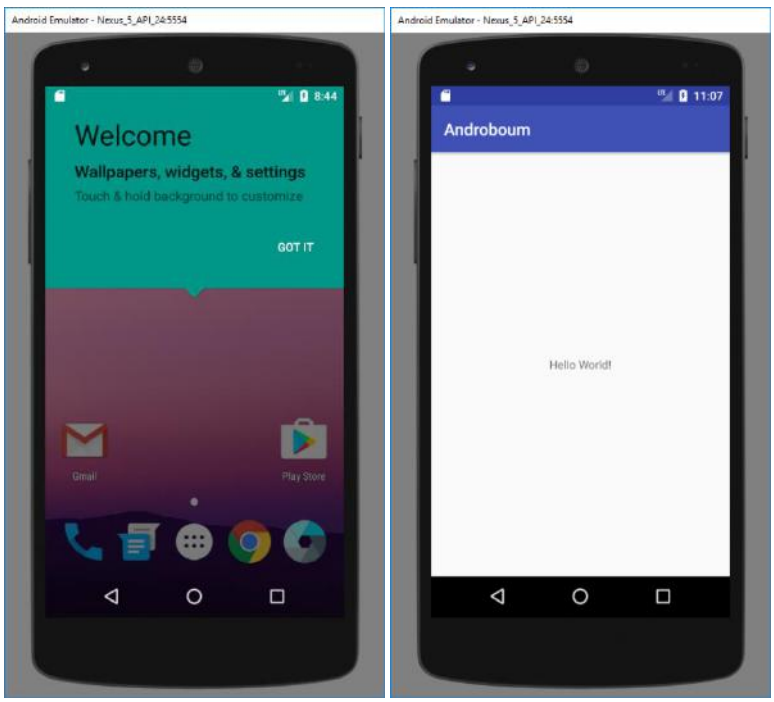
Choisissez ensuite la version d'Android que cet appareil virtuel fera tourner. Cliquez sur un des choix recommandés et si besoin téléchargez l'API retenue en cliquant sur « Download ».



Terminez la création de l'émulateur. Ce dernier doit au final s'afficher dans la liste des appareils virtuels disponibles, comme par exemple dans la photo écran ci-dessous :



Sélectionnez l'appareil virtuel et cliquez sur « ok ». La phase de compilation et de déploiement commence et doit mener au lancement de l'émulateur puis au lancement de l'application sur l'émulateur.

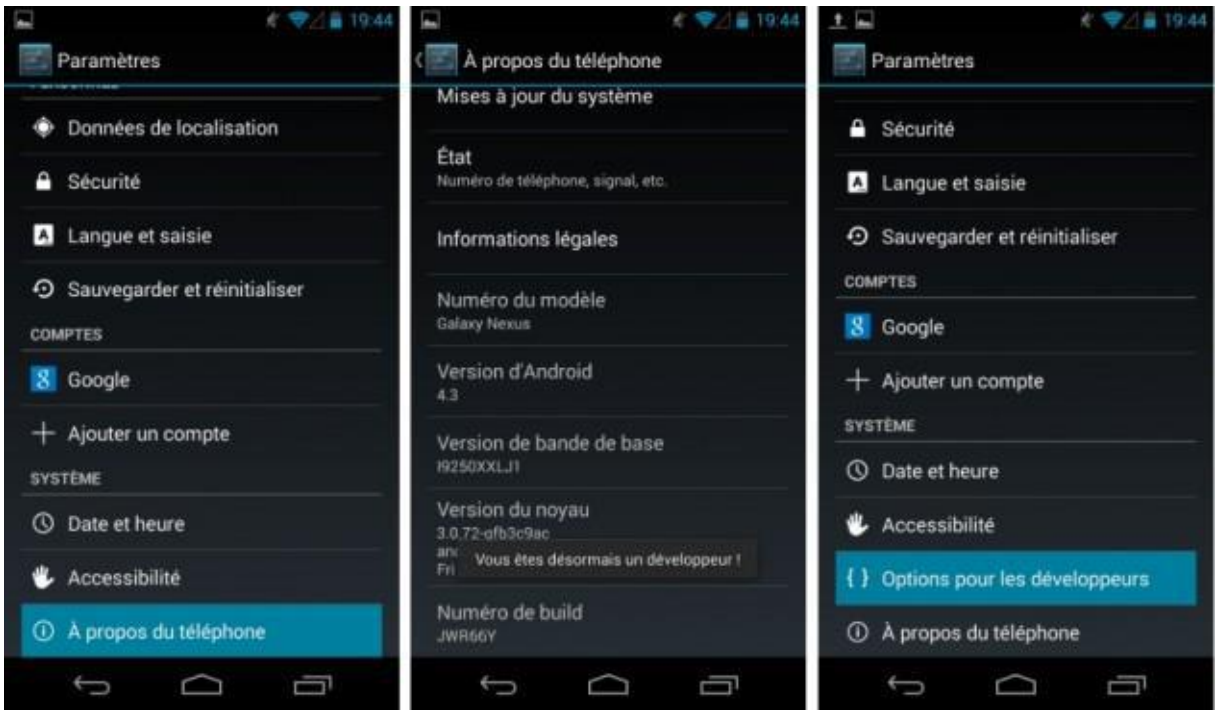


Si tout va bien, vous pouvez utiliser l'émulateur comme si c'était un vrai téléphone. Essayez par exemple de faire des rotations de l'écran, d'accéder à la liste des applications, de lancer d'autres applications parmis les applications préinstallées, voire d'en installer des nouvelles (il faudra vous authentifier si vous voulez accéder au Play Store), etc.

3. Déploiement sur un appareil réel

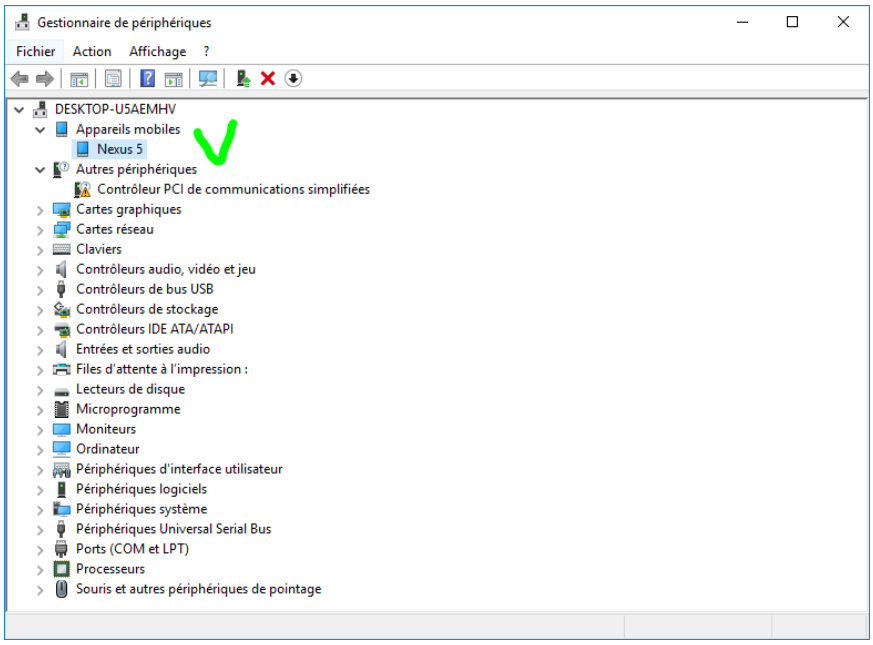
Connectez un téléphone à l'ordinateur qui vous sert d'environnement de développement. Vous aurez besoin pour cela d'un câble USB.

Pour pouvoir utiliser ce téléphone dans le cadre de vos développements Android, vous devez activer un menu caché destiné à cela. Sur les dernières versions d'Android, on active ce menu en cliquant 4 fois sur l'item « Numéro de Build » que l'on trouve dans le menu « A propos du téléphone » des paramètres (voir ci-dessous). Cela fait, un nouveau menu « Options pour les développeurs » apparaît dans la liste des paramètres.



Sélectionnez ce menu et activez la cache à cocher « Débogage USB ». Cochez cette case permet au téléphone de communiquer avec l'ordinateur en mode « Débogage » ce qui permet en particulier le déploiement d'applications.

Après avoir réalisé ces opérations, et si le bon driver est installé sur votre machine, le téléphone doit apparaître dans le gestionnaire de périphériques comme sur l'écran ci-après :



Fermez l'émulateur (s'il est encore ouvert) et refaites un « Run/Run App » de votre projet. Si tout va bien votre téléphone doit apparaître dans la liste des appareils disponibles, éventuellement sous la forme d'un identifiant numérique. Si un identifiant numérique apparaît, il faut autoriser la connexion au niveau du smartphone en cochant « *authorize* » sur la boite de dialogue qui a dû s'ouvrir. Une fois cette connexion autorisée, c'est le nom de votre appareil qui doit apparaître.

Terminez la procédure de déploiement en choisissant votre appareil dans la liste, et si tout va bien l'application doit être déployée et lancée sur votre téléphone (sauf si votre appareil ne permet pas de faire tourner des applications de l'API sélectionnée lors de la création du projet).

III. Ecran d'accueil d'Androboum

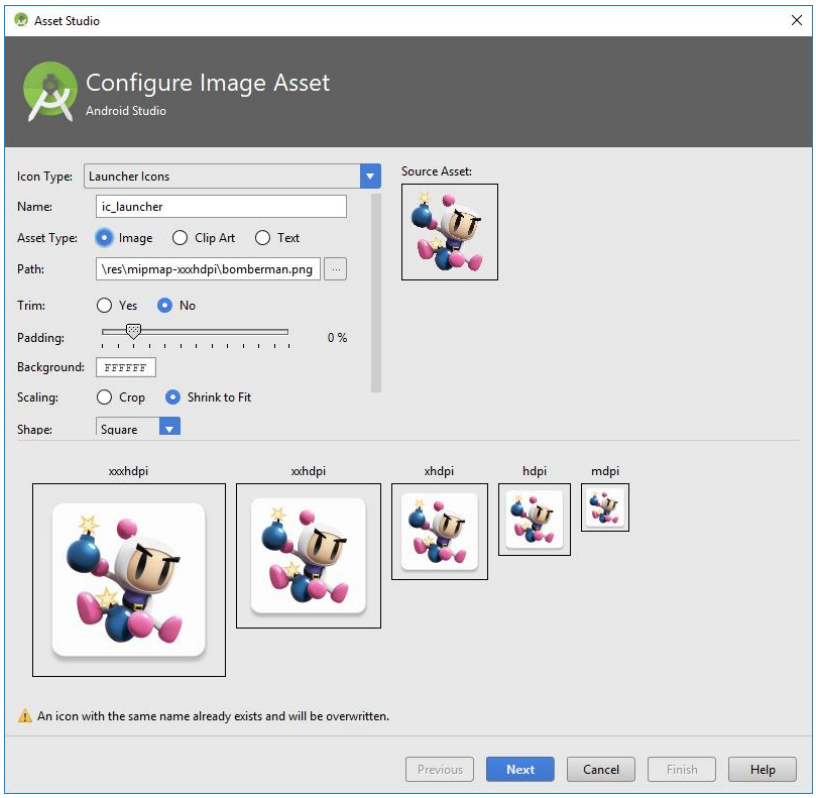
Objectif : Doter l'application d'une icône et d'un écran d'accueil.

1. Icône de lancement

Les applications Android sont représentées par une icône qui sert entre autres à les lancer en cliquant dessus. Cette icône est appelée « *Launcher Icon* » par l'environnement de développement. Par défaut il s'agit d'une icône représentant le robot Android et nous allons la changer immédiatement.

Pour cela, allez chercher sur internet une image que vous souhaitez utiliser pour représenter votre application, et placez-là quelque part sur le disque (mais pas dans le projet Android). Choisissez de préférence une image au format `.png`. Utilisez ensuite le menu « *File/New/Image Asset* » pour lancer le concepteur d'icônes (on peut aussi accéder à ce menu de façon contextuelle par clic droit sur le projet). S'ouvre alors un écran permettant de créer de nouvelles icônes. Cliquez sur le type « *image* » pour l'item « *Asset Type* » et indiquez dans le *Path* l'image que vous avez enregistrée précédemment. L'éditeur vous montre la forme qu'aura votre icône dans les différentes tailles proposées. Faites les réglages nécessaires, et cliquer sur « *Next* » puis « *Finish* » pour remplacer l'icône par défaut par votre image.

Si vous refaites un « *Run* » de votre projet, vous pourrez constater que l'icône de démarrage de votre application a bien été modifiée.



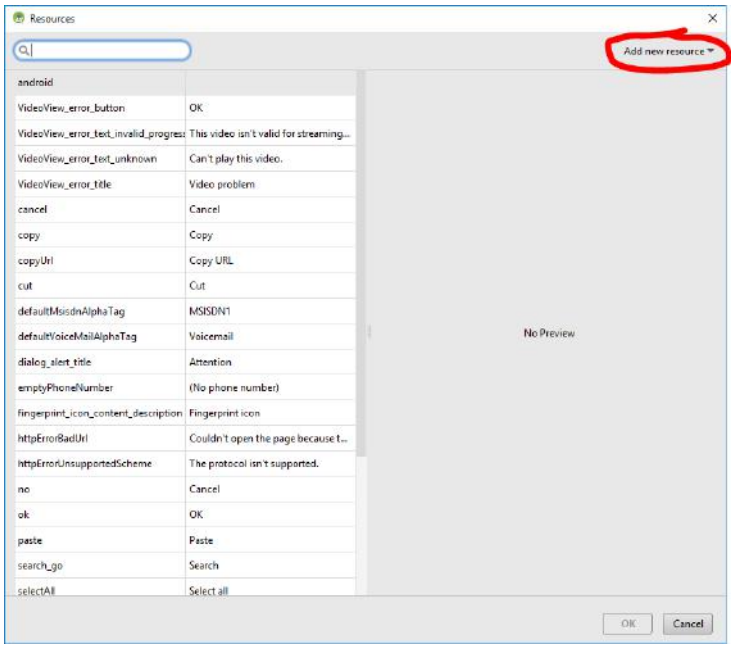
2. Une image pour l'écran d'accueil

Allez chercher une autre image qui sera présentée sur l'écran d'accueil et placez là dans le dossier `app/res/drawable` (en cliquant avec le bouton droit sur le dossier de ressource *`drawable`* vous pouvez choisir le menu *« Show in Explorer »* qui vous permet d'accéder à ce qu'il contient et donc d'y placer des nouveaux fichiers).

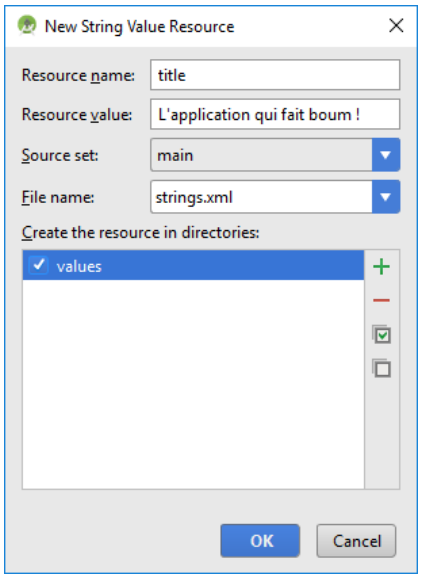
3. Conception de l'écran d'accueil

Reprenez l'édition du fichier `main_activity.xml` en mode « *Design* ». Pour commencer nous allons récupérer le texte « `Hello World` » pour en faire le titre de notre écran d'accueil. Cliquez sur ce texte, et vérifiez qu'un panneau de propriétés s'ouvre sur la droite de l'écran, avec un champ « `Text` » (Si ce n'est pas le cas, vérifiez bien que vous êtes en mode « *Design* »). Ce champ contient le texte qui sera affiché. Nous pourrions le saisir directement mais ce n'est pas conseillé. Les bonnes pratiques Android sont en effet de stocker les différents textes de l'application dans un fichier appelée « `values.xml` » qui se trouve dans le dossier des ressources, et de construire l'interface graphique en faisant référence aux valeurs contenues dans ce fichier. Cela permet en particulier de créer des interface « localisées », c'est-à-dire dont le contenu peut s'adapter à la langue de l'utilisateur. Nous allons mettre cela en application en prévoyant que notre application propose à la fois une version anglaise, et une version française (par défaut).

Par conséquent, au lieu de saisir un texte dans le champ « `Text` » des propriétés de l'item « `Hello World` », cliquez sur les trois petits points à sa droite pour obtenir l'écran ci-après :

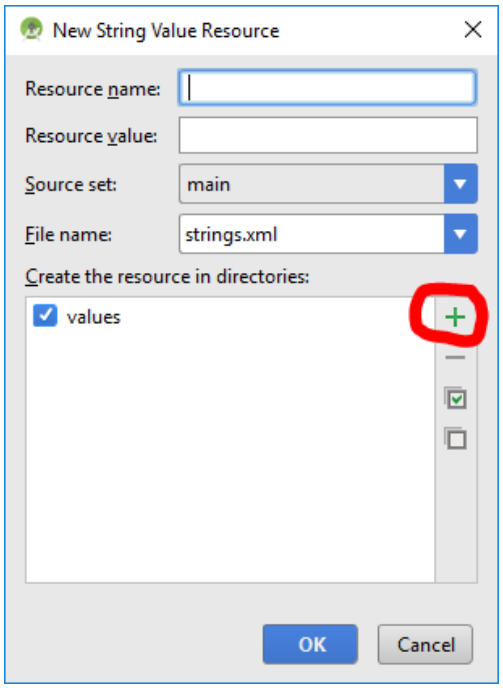


Cliquez sur le bouton « *Add new resource* » en haut à droite puis sur son sous-menu « *New String Value...* ». Complétez le formulaire qui s'ouvre pour donner un titre et une valeur à la ressource comme sur l'écran ci-après :

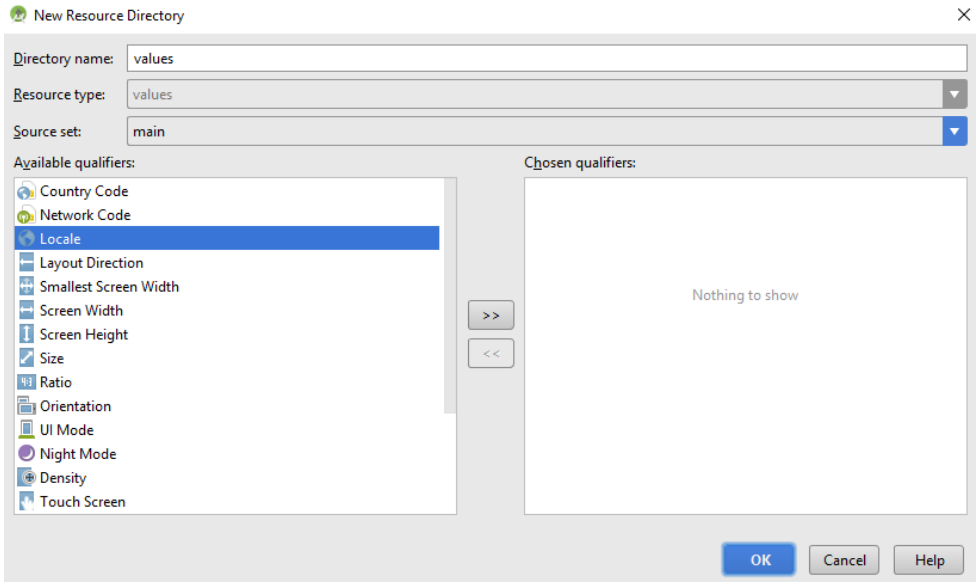


En cliquant sur « ok » vous devez constater que le contenu du texte « `Hello World` » est désormais la phrase que nous avons saisie.

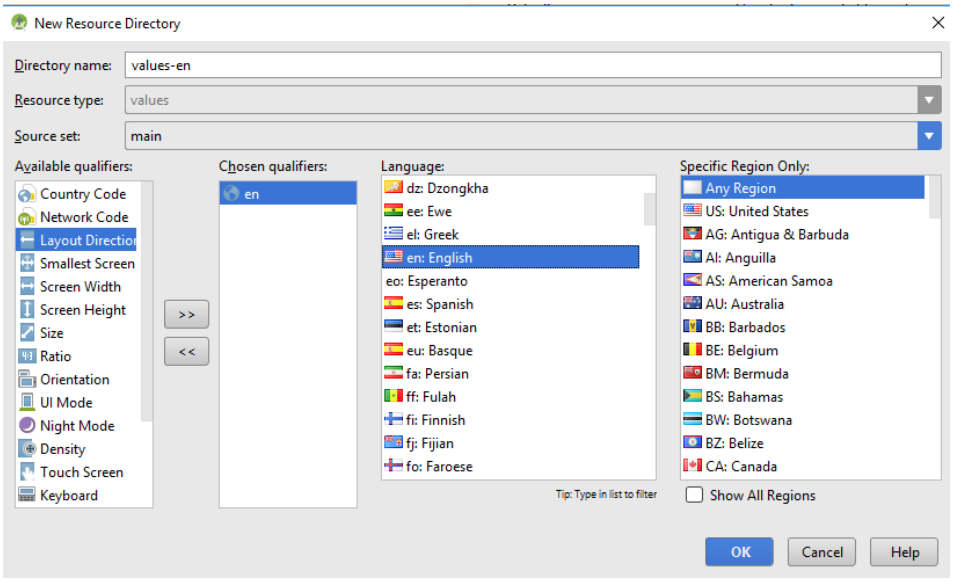
Nous voulons maintenant traduire ce texte en anglais. Pour cela, réouvrez le même écran en cliquant à nouveau sur les trois petits points à droite de la propriété « `Text` ». Recliquez également sur « *Add new resource* », et cette-fois cliquez sur le signe + pour créer un nouveau fichier de valeur qui correspondra à la langue anglaise.



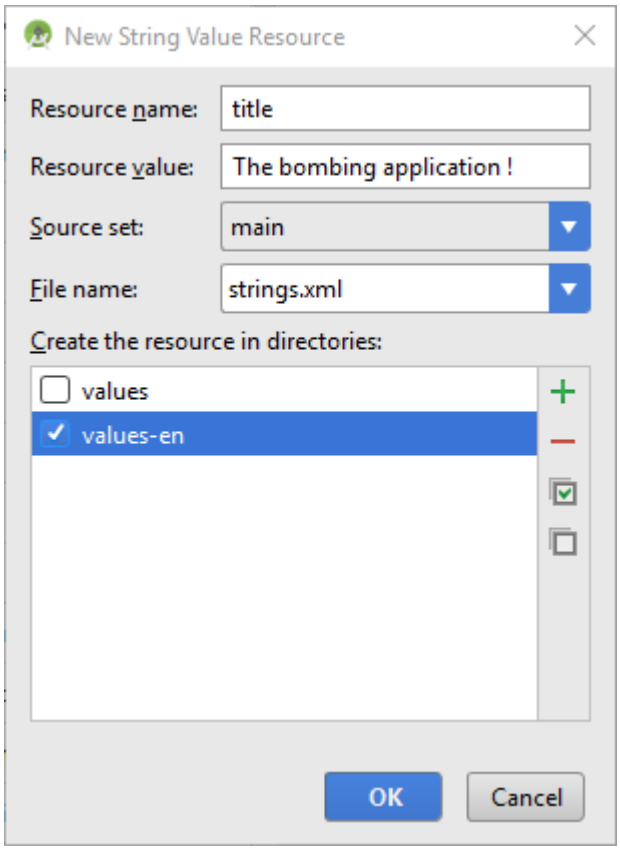
Dans la fenêtre de dialogue qui s'ouvre, sélectionnez l'item `Locale` et faites-le passer dans la partie droite en cliquant sur le symbole double chevron au milieu :



Sélectionnez ensuite « *English* » dans la liste des langues et laissez la région sur « *Any Region* » :

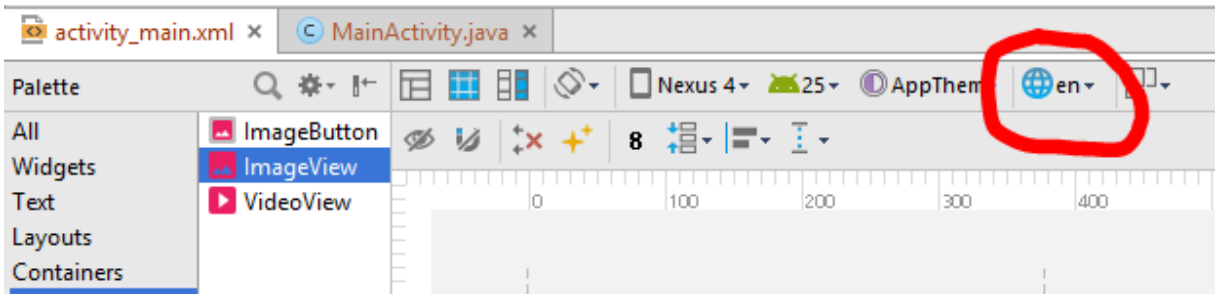


Cliquer sur « *ok* » et remplissez cette fois l'écran de ressource comme ci-dessous (notez-bien que *values-en* est coché) :



Le résultat de cette opération est la création d'un deuxième fichier de ressources appelé « `values-en` » dans lequel l'application ira chercher ses chaines de caractères si l'application est configurée pour être en anglais.

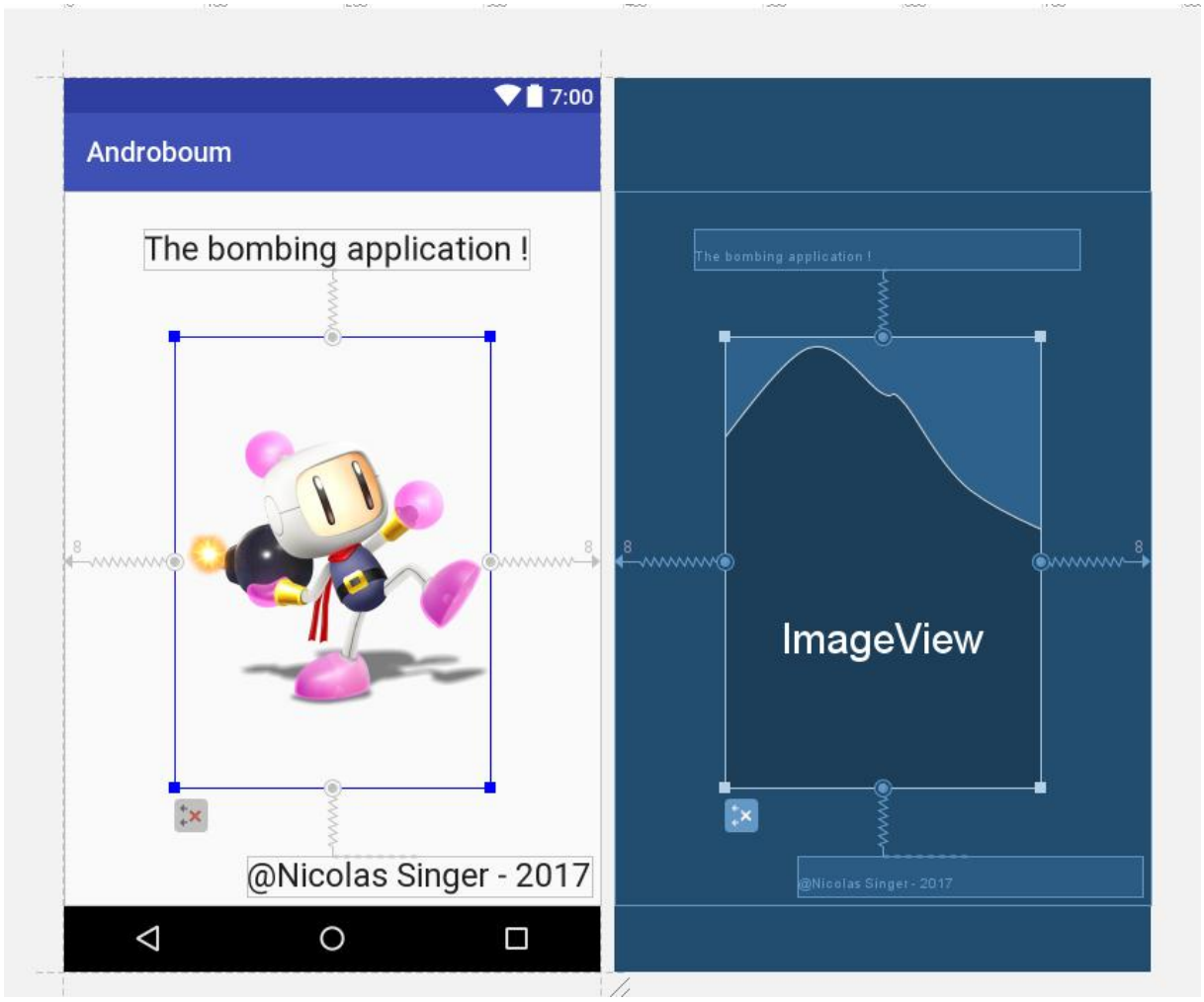
Pour le tester vous pouvez utiliser le menu du designer permettant de modifier la langue d'affichage :



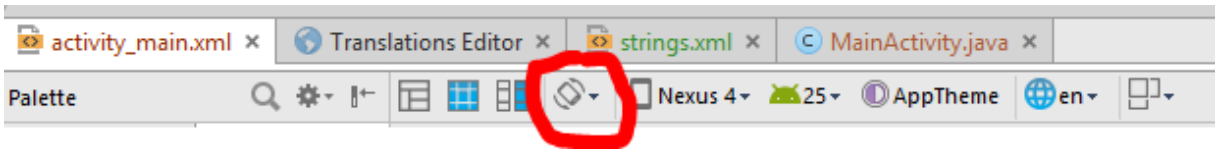
En principe les changements de langues doivent désormais modifier le texte affiché.

Vous pouvez également tester en condition réelle l'application en faisant un « Run » puis en paramétrant l'émulateur ou votre téléphone pour changer la langue par défaut. Là encore le texte doit s'adapter à la langue choisie.

Ajouter à présent une image à votre écran (avec le champ « `ImageView` » de la rubrique « `Images` »), ainsi qu'un deuxième champ texte qui servira à afficher le copyright de l'application. Essayez d'utiliser le designer pour paramétrer les contraintes (liaisons entre les éléments de l'interface) pour que l'image soit centrée horizontalement, ainsi qu'entre le titre et le copyright. Au final votre écran doit ressembler à la copie écran ci-dessous. Notez les « ressorts » qui apparaissent et qui indiquent les contraintes qui pèsent sur l'image :

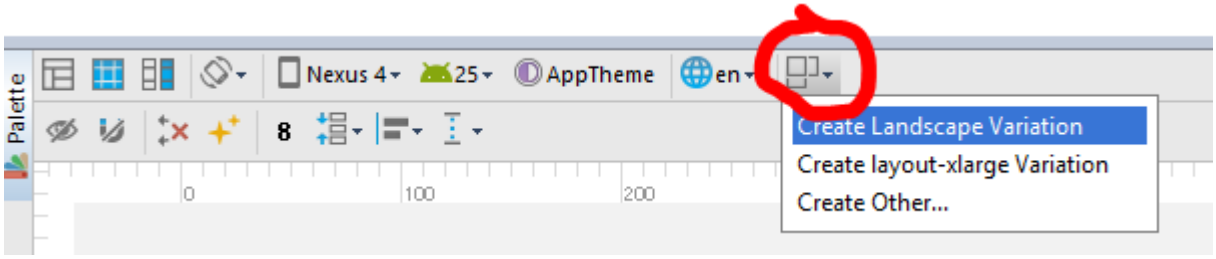


Vous pouvez vérifier que l'application respecte ces contraintes, par exemple en sélectionnant une autre taille d'écran ou simplement en faisant passer le designer en mode paysage :

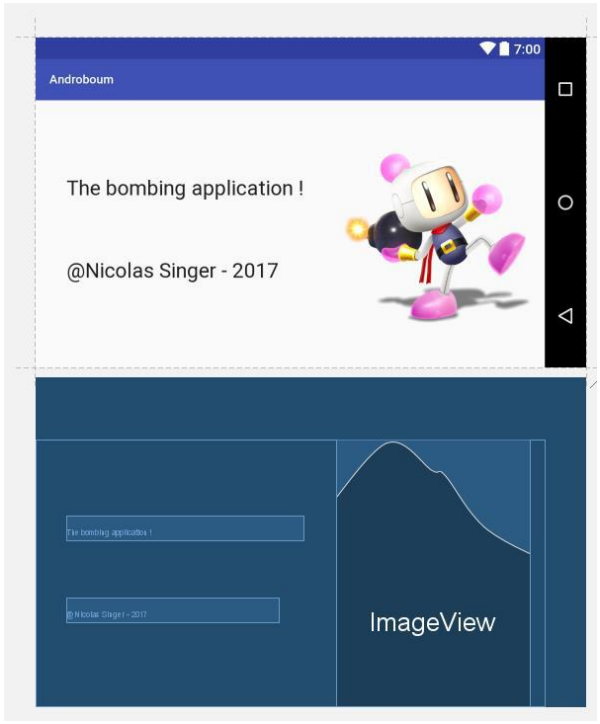


L'écran doit effectuer une rotation, et l'image doit conserver son centrage horizontal et vertical.

Restons sur cette histoire de rotation. Imaginons que nous souhaitions proposer un écran d'accueil différent suivant que le téléphone est en position portrait ou paysage. Gardons par exemple la configuration actuelle pour le mode portrait, mais proposons en une nouvelle pour le mode paysage. Cela est possible en créant deux fichiers de *layout* différents. Un pour le portrait (l'actuel), et un nouveau qui ne sera utilisé que pour le mode paysage. Le designer nous permet de le créer facilement en cliquant sur le bouton de l'interface identifié ci-dessous puis sur « *Create Landscape Variation…* » .



Si vous faites-cela, le designer vous propose une nouvelle vue que vous pouvez modifier mais qui ne sera valable que pour le mode paysage. Par exemple, dans ce mode nous pouvons choisir de mettre l'image à droite, et le texte à gauche suivant le modèle ci-dessous :



Faites-le (il faudra changer les contraintes qui pèse sur l'image pour y arriver), et vérifier que lors de la rotation de l'écran, l'application alterne désormais entre les deux présentations.

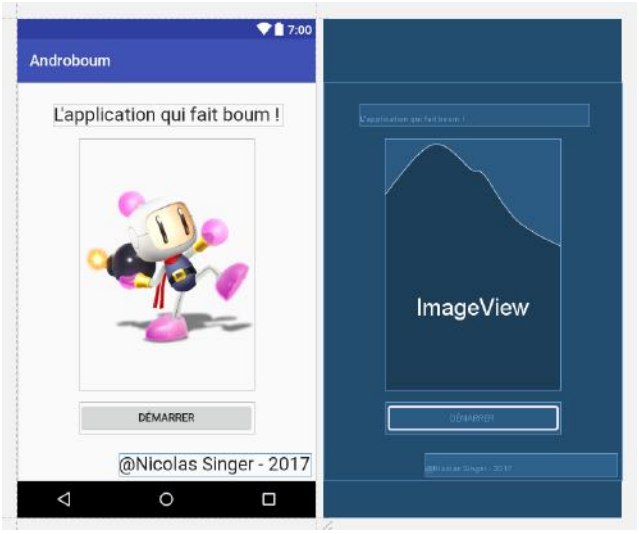
Vérifiez aussi qu'un fichier supplémentaire `activity_main.xml (land)` a bien été créé dans le projet. Vous pouvez consulter son contenu en mode texte (xml) pour le comparer au fichier d'origine.

Enfin ajoutez à votre écran un bouton « Démarrer » (qui devra être traduit en « *Start* » en anglais) qui permettra plus tard de passer à l'écran suivant. Vous trouverez les boutons cliquables dans la catégorie « `Widgets` » puis « `Button` » du designer. Repensez votre interface pour intégrer proprement ce bouton, et n'oubliez pas de le faire aussi pour le mode paysage.

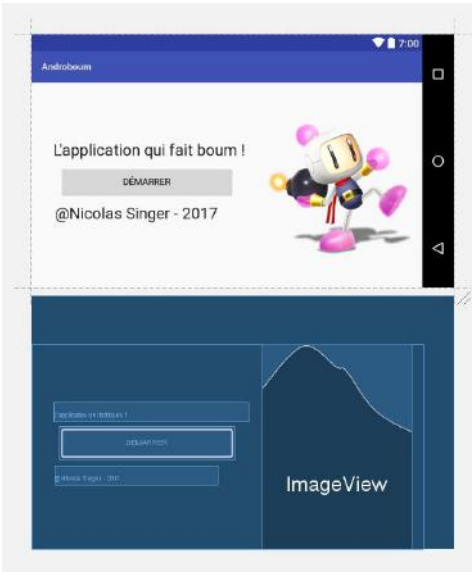
Plus d'informations sur l'utilisation des *Constraint Layout* sont disponibles par exemple à l'adresse :

https://developer.android.com/training/constraint-layout/index.html

Au final vous pouvez obtenir quelque chose ressemblant à l'écran ci-dessous, mais vous pouvez certainement faire mieux :



Et en mode paysage :



IV. Authentification de l'utilisateur

Objectif : Concevoir une interface permettant à l'utilisateur de s'authentifier.

Pour cela nous utiliserons les services proposés par la plateforme *firebase* (https://firebase.google.com). L'authentification d'un utilisateur est toujours quelque chose de délicat à coder dans une application. D'un côté il faut pouvoir gérer la liste des utilisateurs autorisés à se connecter, de l'autre s'assurer que l'authentification a lieu dans un cadre sécurisé sans possibilité par exemple de captation du mot de passe. C'est tout l'intérêt d'utiliser pour cela une plateforme externe qui met à disposition des services éprouvées qui plus est dans notre cas dotés déjà des interfaces homme machine adéquates.

1. Firebase

Firebase hébergera donc le « backend » de notre application, c'est-à-dire les services auxquels elle fera appel pour son fonctionnement, et les données qu'elle aura besoin de stocker. Techniquement, ces services sont proposés sous la forme d'une API qui sera plus ou moins masquée par les outils automatiques d'intégration que nous propose Android Studio.

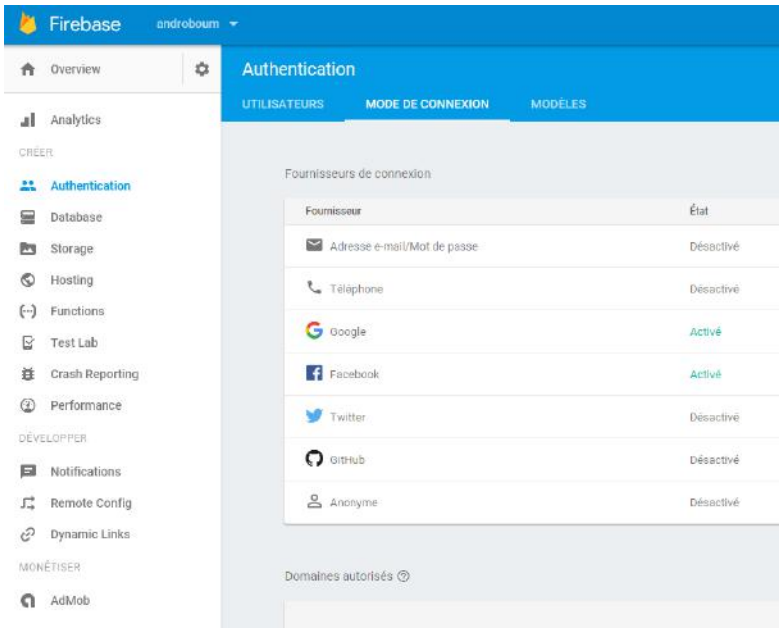
L'accès à *firebase* nécessite un compte google. Un compte a donc été créé spécialement pour notre application (sachant qu'un même compte peut bien entendu gérer plusieurs applications). Voici ces coordonnées :

Login : comptedream@gmail.com

Password : (donné par l'enseignant sur demande)

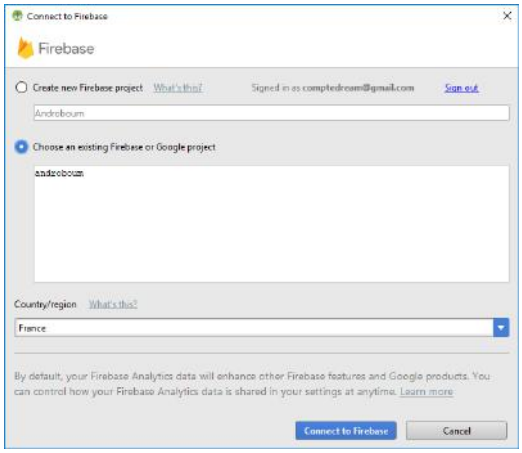
Connectez-vous sur la plateforme à l'adresse https://firebase.google.com/ avec le compte ci-dessus et rendez-vous dans la console de gestion des projets en cliquant sur le bouton « Go To Console ». Vous constaterez qu'un projet « androboum » a déjà été créé. Si ce n'était pas le cas vous auriez pu créer un nouveau projet en cliquant sur « Ajouter un projet ». Ce projet *firebase* sera lié à notre application mobile dès que vous aurez faire le nécessaire sous Android Studio.

Le projet a été préconfiguré pour accepter deux fournisseurs d'identités permettant de se connecter : facebook et google. Vous pouvez le constater par vous-même en cliquant sur le projet puis sur la rubrique « Authentification » et enfin sur l'onglet « Mode de Connexion ».



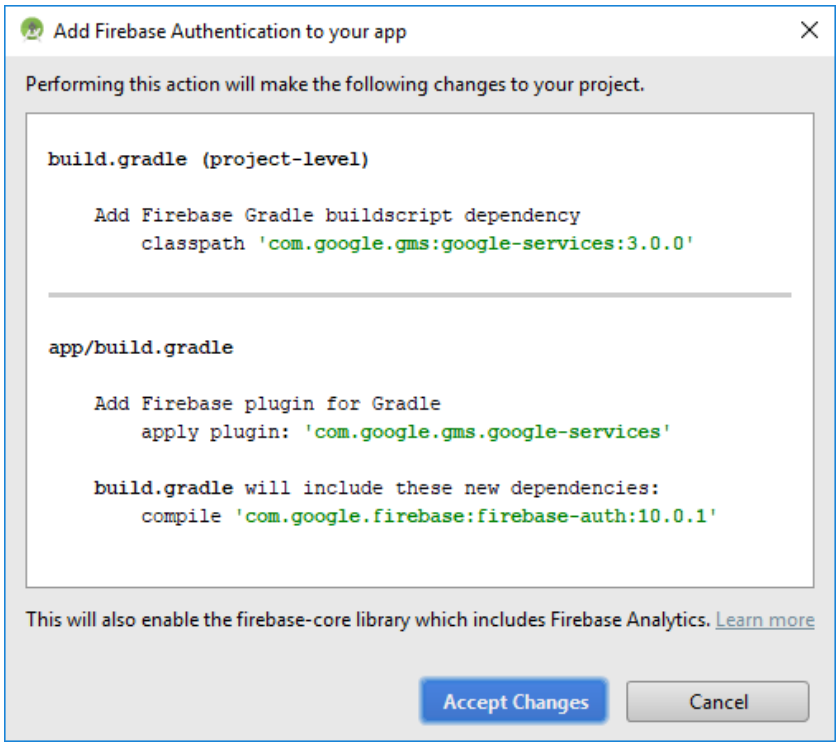
2. Liaison entre Android Studio et firebase

Retournez sur Android Studio et sélectionnez le menu « Tool/Firebase ». Cela ouvre un panel présentant brièvement certains services de firebase. Choisissez dans la partie « Authentification », l'item « Email and password authentification ». Puis cliquer sur le bouton « connect to Firebase » pour effectuer la liaison entre votre projet mobile et la plateforme firebase. Menez la procédure à son terme en utilisant le compte comptedream@gmail.com. Vous devez arriver à l'écran ci-dessous :



Ne créez pas un nouveau projet mais sélectionnez le projet androboum existant comme sur la copie écran. Puis cliquez sur « Connect to Firebase ». Il est important que vous utilisiez tous le même projet car vos applications doivent partager la même base commune d'utilisateur (entre autres choses).

En principe Android Studio est désormais connecté à firebase. Cliquez à présente sur le bouton « Add Firebase Authentication to your app ». Les dépendances nécessaires vont être ajoutés à votre projet comme en témoigne l'écran ci-dessous :



Cliquez sur « Accept Changes ».

A ce point cessez de suivre l'aide proposé par le panel car nous allons utiliser l'interface proposée par *firebase* pour gérer directement les écrans nécessaires à l'authentification.

Pour information cette interface et sa procédure d'installation est disponible à partir d'un dépôt GitHub d'adresse : https://github.com/firebase/FirebaseUI-Android

Sous Android Studio, modifiez le fichier `build.gradle (Project : Androboum)` pour intégrer le dépôt Maven à la liste des dépôts disponibles. Pour cela la dernière partie du fichier doit ressembler à :

```
allprojects {
    repositories {
        jcenter()
        maven {
            url "https://maven.google.com"
        }
    }
}
```

Modifiez ensuite le fichier `build.gradle (Module : app)` pour y ajouter les lignes de dépendances suivantes :

```
dependencies {
    // ...
 compile 'com.google.firebase:firebase-auth:11.0.1'
 compile 'com.firebaseui:firebase-ui-auth:2.0.1'
 compile 'com.facebook.android:facebook-android-sdk:4.22.1'
}
```

Si tout va bien, le projet doit se resynchroniser correctement.

Pour que l'authentification via *facebook* fonctionne également il faut ajouter les lignes suivantes dans le fichier `strings.xml` (ne remettez pas la balise `<resources>`, elle n'est là que pour vous indiquer où insérer les lignes) :

```
<resources>
    <!-- ... -->
    <string name="facebook_application_id" translatable="false">APP_ID</string>
    <!-- Facebook Application ID, prefixed by 'fb'.  Enables Chrome Custom tabs. -->
    <string name="facebook_login_protocol_scheme" translatable="false">fbAPP_ID</string>

</resources>
```

En remplaçant APP_ID par 1809914232370472 qui est un identifiant déclaré auprès de facebook pour les besoins de ce cours. Donc au final insérez les deux lignes :

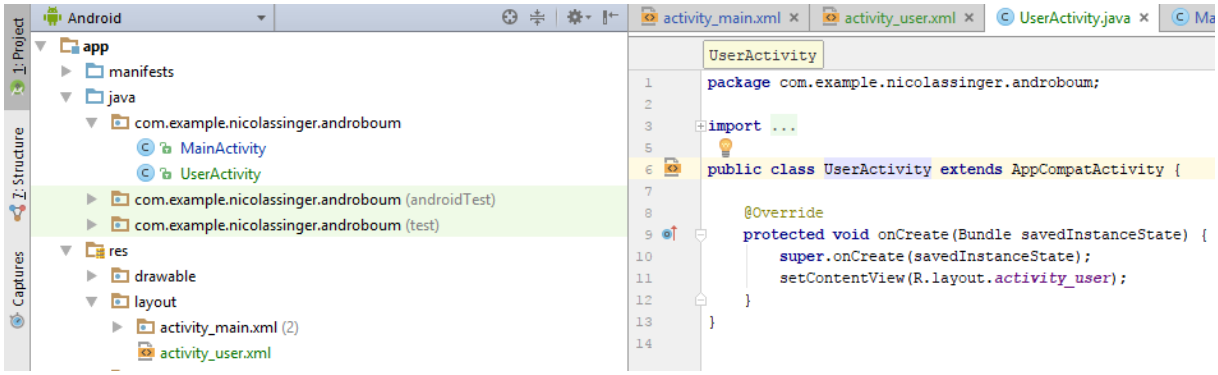
```
<resources>
    <string name="facebook_application_id" translatable="false">1809914232370472</string>
    <string name="facebook_login_protocol_scheme"
translatable="false">fb1809914232370472</string>

</resources>
```

La mise en place est terminée, passons à présent au développement de l'activité d'authentification.

3. Création d'une nouvelle activité

Nous voulons qu'un clic sur le bouton « Démarrer » de notre application mène à un écran qui vérifie que l'utilisateur est connecté et qui sinon lui propose de le faire. Cet écran doit ensuite (pour l'instant) simplement afficher le nom de l'utilisateur.

Pour créer cet écran, créez une nouvelle activité du nom de *UserActivity* en passant par le menu d'Android Studio « File/New/Activity/Empty Activity ». Les fichiers du projet doivent désormais ressembler à l'écran ci-dessous :

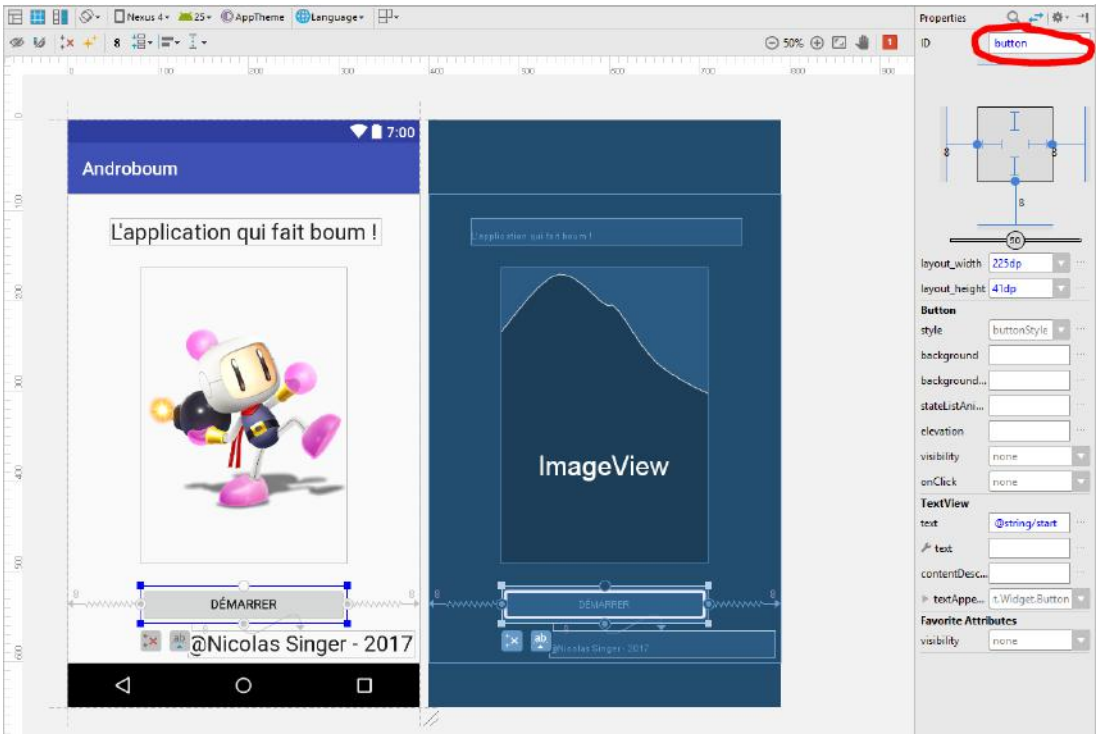


Cette activité ne fait pour l'instant rien, mais nous allons néanmoins ajouter le code nécessaire pour qu'un clic sur le bouton « Démarrer » de *MainActivity* la lance. Voici le code à ajouter qui permet de lancer cette activité :

```
/** appelé quand l'utilisateur clique sur le bouton */
public void lancerUser() {
    Intent intent = new Intent(this, UserActivity.class);
    startActivity(intent);
}
```

Ce code crée un nouvel objet de type Intent qui est lié à l'activité à démarrer. Le lancement de l'activité se fait ensuite en passant cet Intent à la méthode `startActivity(Intent)`.

Il ne reste plus qu'à faire en sorte que ce code soit appelé en cas de clic sur le bouton. Pour cela cliquez sur le fichier de layout `activity_mail.xml` pour obtenir l'identifiant associé au bouton (ici il s'agit tout simplement de `button`) :



Utilisez cet identifiant en tant que paramètre de la fonction `findViewById(…)` en le préfixant par `R.id` comme dans le code ci-dessous. Ce code est à ajouter dans la méthode `OnCreate` de *MainActivity* (juste après le `setContentView`) :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button bouton = (Button) findViewById(R.id.button);
    bouton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            lancerUser();
        }
    });
}
```

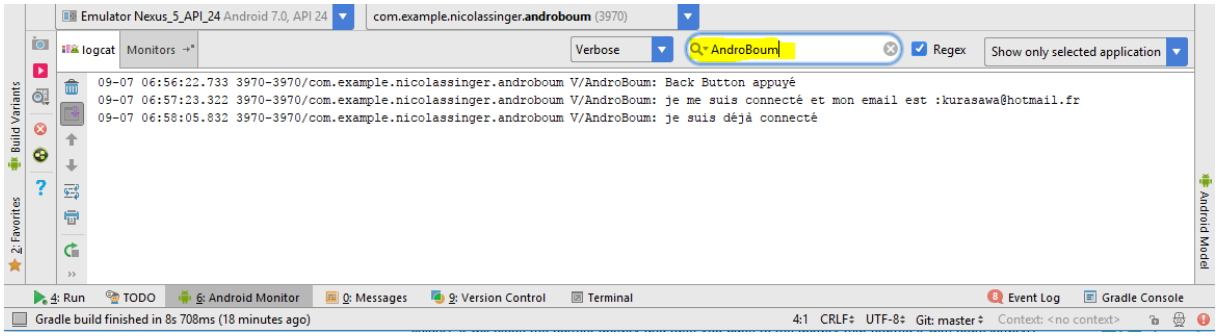
Lancez l'application pour vérifier que tout fonctionne. En principe un clic sur le bouton doit vous mener à un écran vide et vous pouvez revenir à l'écran principal en cliquant sur le bouton « back » du téléphone ou de l'émulateur.

Maintenant que la liaison entre les deux activités fonctionne, modifions notre écran vide pour qu'il authentifie l'utilisateur.

Deux cas possibles : Soit l'utilisateur est déjà connecté et on affiche directement son identifiant. Soit-il ne l'est pas, auquel cas on lance l'écran d'authentification. L'écran d'authentification est entièrement géré par *firebase*, nous n'avons donc pas à le créer. Voici donc le code complet à insérer dans `UserActivity`, les commentaires expliquant sa structure :

```
public class UserActivity extends AppCompatActivity {

    //  on choisit une valeur arbitraire pour représenter la connexion
    private static final int RC_SIGN_IN = 123;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_user);

        // on demande une instance du mécanisme d'authentification
        FirebaseAuth auth = FirebaseAuth.getInstance();
        // la méthode ci-dessous renvoi l'utilisateur connecté ou null si personne
        if (auth.getCurrentUser() != null) {
            // déjà connecté
            Log.v("AndroBoum","je suis déjà connecté sous l'email :"
                 +auth.getCurrentUser().getEmail());
        } else {
           // on lance l'activité qui gère l'écran de connexion en
           // la paramétrant avec les providers googlet et facebook.
            startActivityForResult(AuthUI.getInstance().createSignInIntentBuilder()
                            .setAvailableProviders(Arrays.asList(
                new AuthUI.IdpConfig.Builder(AuthUI.GOOGLE_PROVIDER).build(),
                new AuthUI.IdpConfig.Builder(AuthUI.FACEBOOK_PROVIDER).build()))
            .build(), 123);
        }
    }

    // cette méthode est appelée quand l'appel StartActivityForResult est terminé
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        // on vérifie que la réponse est bien liée au code de connexion choisi
        if (requestCode == RC_SIGN_IN) {
            IdpResponse response = IdpResponse.fromResultIntent(data);

            // Authentification réussie
            if (resultCode == RESULT_OK) {
                Log.v("AndroBoum","je me suis connecté et mon email est :"+
                          response.getEmail());
                return;
            } else {
                // echec de l'authentification
                if (response == null) {
                    // L'utilisateur a pressé "back", on revient à l'écran
                    // principal en fermant l'activité
                    Log.v("AndroBoum","Back Button appuyé");
                    finish();
                    return;
                }
                // pas de réseau
                if (response.getErrorCode() == ErrorCodes.NO_NETWORK) {
                    Log.v("AndroBoum","Erreur réseau");
                    finish();
                    return;
                }

                // une erreur quelconque
                if (response.getErrorCode() == ErrorCodes.UNKNOWN_ERROR) {
                    Log.v("AndroBoum","Erreur inconnue");
                    finish();
                    return;
                }
            }

            Log.v("AndroBoum","Réponse inconnue");
        }
    }
}
```

Vous pouvez tester ce code et utiliser l'onglet « Android Monitor » d'Android Studio pour vérifier les logs générés par lies lignes `Log.v("AndroBoum",…");` Ces lignes sont des commandes qui permettent de tracer le déroulement du programme en générant des sorties dans la console de monitoring. Le premier mot (*AndroBoum*) permet de donner une étiquette aux logs de façon à pouvoir filtrer ces lignes-là. La copie écran ci-dessous montre la console de log filtrée suivant *AndroBoum* :



Si tout se passe bien, en cas de connexion réussie, l'email de la personne connecté doit apparaître dans les logs. Si vous poussez l'expérience vous devez constater qu'après la première connexion réussie, l'application ne demande plus de s'authentifier et se connecte automatiquement avec les derniers identifiants utilisés. Pour refaire une authentification il va falloir que nous implémentions un moyen de déconnecter l'utilisateur. La barre des taches est un bon endroit ou loger cette action.

4. Ajout d'une barre d'action à l'activité

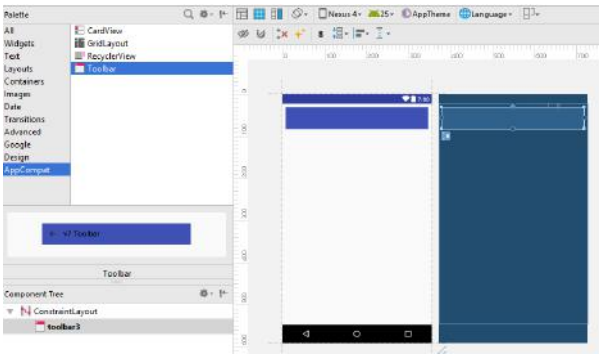
Commencez par enlever l'*action bar* par défaut en changeant le style de l'application dans le fichier styles.xml qui se trouve dans le dossier des ressources. Pour cela remplacez la ligne :

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
```

par la ligne :

```
<style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
```

Cliquez ensuite sur le fichier *activity_user.xml* et ajoutez une *ToolBar* (on la trouve dans la catégorie *AppCompat*) que vous placerez tout en haut du *layout*.



Ceci fait, retournez dans le code de la classe *UserActivity* et insérez les deux lignes suivantes dans la méthode `OnCreate`, juste après le `setContentView` :

```
Toolbar myToolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(myToolbar);
```

Si vous testez votre application, la barre d'action a dû disparaître de l'écran principal (puisque nous avons défini un style par défaut sans barre d'action) mais doit être présente sur l'écran de la *UserActivity*.

Ajoutons maintenant des actions dans la barre. Cela passe par la définition d'items définis dans un fichier contenu dans un sous-dossier menu du dossier des ressources. Si ce sous-dossier n'existe pas déjà, créez-le en passant par le menu `File/New/Android Resource Directory`. Un écran s'ouvre dans lequel il faut spécifier *menu* dans le champ *Resource type* et cliquer sur « ok ».

Dans ce dossier créez un nouveau fichier (vous pouvez l'appeler `actions.xml`) et placez-y les actions quoi doivent apparaître dans la *ToolBar* sous la forme suivante :

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <!-- "Mark Favorite", should appear as action button if possible -->
    <item
        android:id="@+id/action_logout"
        android:title="@string/action_logout"
        app:showAsAction="ifRoom"/>

    <!-- Settings, should always be in the overflow -->
    <item android:id="@+id/action_settings"
        android:title="@string/action_settings"
        app:showAsAction="never"/>

</menu>
```

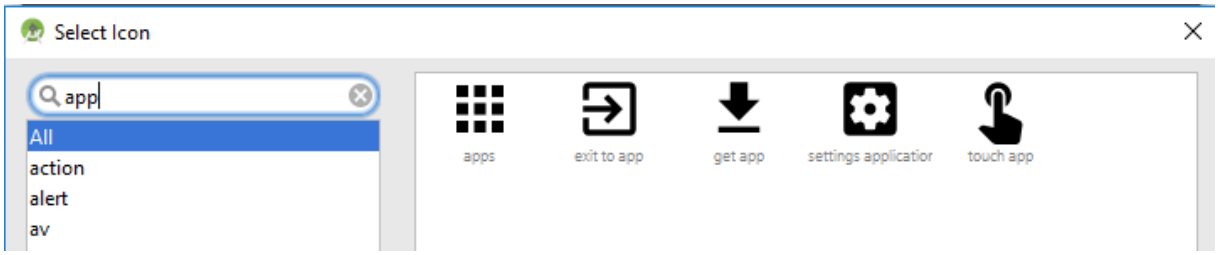
Vous n'oublierez pas d'ajouter dans les fichiers `strings.xml` les versions françaises et anglaises des chaines de caractères `action_logout` et `action_settings`.

Pour utiliser ces items dans la barre, on ajoute dans *UserActivity* la méthode suivante :

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is
present.
    getMenuInflater().inflate(R.menu.actions, menu);
    return true;
}
```

Testez l'effet de ce code, vous devez obtenir une barre d'action ressemblant à l'écran ci-dessous :

Le *LOGOUT* n'est pas très sexy, aussi nous allons le remplacer par une icône issue de la bibliothèque d'icône *material design*. Pour ajouter une icône à votre projet vous pouvez utiliser le menu *File/New/Vector Asset* puis cliquez sur le champ image à côté du label « Icon » . S'ouvre alors un écran vous permettant de sélectionner l'icône que vous désirez. Prenez par exemple celle qui s'appelle « exit to app » qui se rapproche assez de quelque chose pouvant illustrer une déconnexion :



L'icône étant ajoutée dans le dossier *drawable* du projet, vous pouvez la lier au bouton « LOGOUT » en insérant la ligne suivante dans son item du fichier `actions.xml` :

```
<item
     android:id="@+id/action_logout"
     android:title="@string/action_logout"
     android:icon="@drawable/ic_exit_to_app_black_24dp"
     app:showAsAction="ifRoom"/>
```

Relancez l'application, et constatez que le texte LOGOUT est bien remplacé par une jolie icône :

Il reste maintenant à traiter la sélection des actions. Le code à ajouter pour intercepter un clic de l'utilisateur sur une des actions est le suivant (à mettre dans *UserActivity*) :

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_settings:
            // choix de l'action "Paramètres", on ne fait rien
              // pour l'instant
            return true;

        case R.id.action_logout:
            // choix de l'action logout
            // on termine l'activité ce qui déconnectera l'utilisateur
            finish();
            return true;

        default:
            /// aucune action reconnue
            return super.onOptionsItemSelected(item);

    }
}
```

Il faut également modifier l'action onDestroy() de l'activité, qui est appelée quand l'activité se termine (si la méthode n'existe pas déjà, ajoutez-là) :

```
@Override
protected void onDestroy() {
    // on déconnecte l'utilisateur
    AuthUI.getInstance().signOut(this);
```

```
    super.onDestroy();
}
```

L'effet de la ligne `AuthUI.getInstance().signOut(this);` est de déconnecter l'utilisateur. Si vous testez l'application, vous pouvez constater qu'après une déconnexion, l'application demande la réauthentification de l'utilisateur lors de la prochaine connexion.

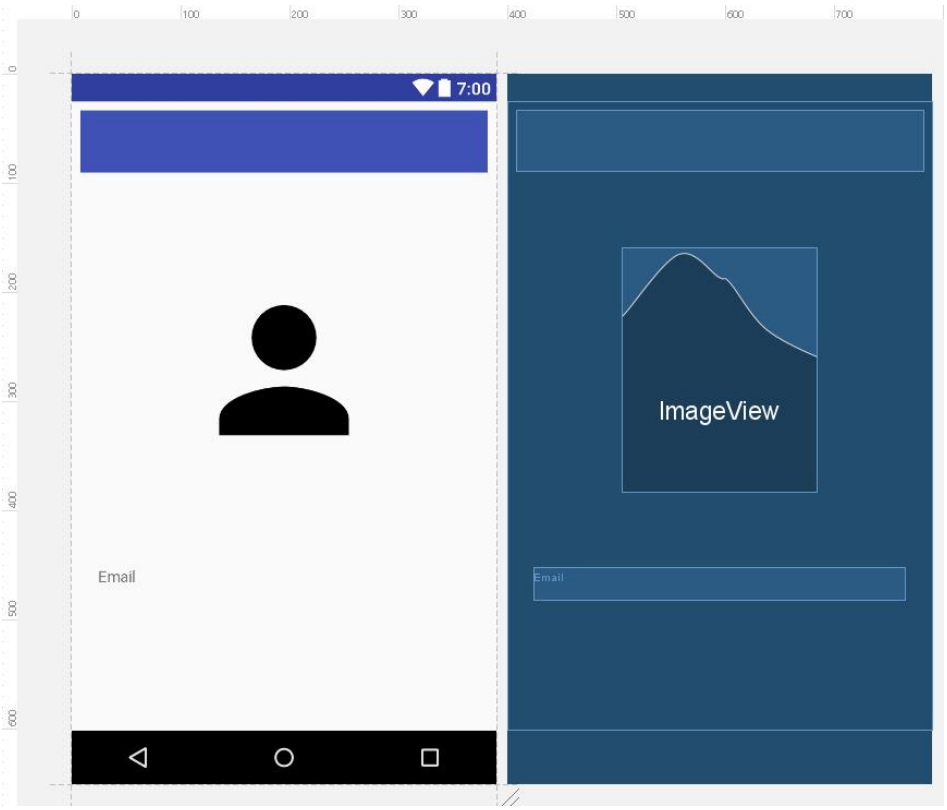
V. Afficher son profil

Objectif : Créer un écran d'affichage des informations de votre profil.

L'activité *UserActivity* est pour l'instant bien vide. Nous allons en faire un écran d'affichage du profil de l'utilisateur connecté. Pour les besoins de ce cours, le profil se réduira à un email et une image vous représentant.

En utilisant le designer d'Android Studio, ajoutez donc au *layout* user_activity.xml un champ texte qui contiendra l'email et un emplacement pour une image. Pour l'image par défaut associé au profil, vous pouvez utiliser l'icône *person* que l'on trouve dans la catégorie sociale des icônes *material design*. Pour l'ajouter au projet, procédez comme dans la section précédente.

Au final votre écran doit ressembler à la copie écran ci-après :



Voyons maintenant comment remplir le champ email avec votre adresse de connexion.

Repérez tout d'abord l'identifiant associé à ce champ. Pour cela cliquez sur le champ texte et repérez (ou changez-le) son identifiant. Comme vu précédemment c'est cet identifiant (précédé de R.id) qui doit être passé à la méthode findViewById(…) pour accéder à l'objet représentant le champ texte.

Un fois l'objet obtenu, on peut faire appel à sa méthode setText(…) pour spécifier le contenu textuel. Mettons que votre champ ait pour identifiant « email », cela donne :

```
TextView textView = (TextView) findViewById(R.id.email);
textView.setText(…);
```

Modifiez donc le code de *UserActivity* pour insérer les lignes de code nécessaire pour que l'email de l'utilisateur connecté figure dans le champ texte. Attention, l'email doit s'afficher aussi bien si l'utilisateur est déjà connecté, qu'après être passé par l'écran d'authentification.

Occupons-nous à présent de l'image. Pour la définir nous demanderons à l'utilisateur d'effectuer un *long press* dessus. Ce clic long aura pour effet de demander la sélection d'une image qui pourra provenir du téléphone ou de la caméra. Cette image une fois sélectionnée remplacera l'image du profil.

La bonne nouvelle c'est que pour réaliser cela, Android va prendre en charge une bonne partie de la procédure. Le système va en effet nous permettre de sélectionner n'importe quelle application de l'appareil qui permet d'obtenir une image. Une fois que cette application aura terminé la sélection, elle passera en résultat l'image sélectionnée à notre application. Cet communication inter-application, qui permet à une application d'en appeler une autre pour obtenir quelque chose est un des points forts de la plateforme.

Pour le mettre en œuvre, nous utilisons ce qu'Android appelle un `Intent`. Un *Intent* est un objet qui permet de lancer une nouvelle activité en décrivant uniquement ce que l'on souhaite obtenir. C'est le système qui choisira ensuite (ou plutôt qui fera choisir par l'utilisateur) la bonne application à lancer pour cela.

Nous allons donc définir un Intent de type « image » et lancer une activité générique pour en obtenir une. Voici les lignes qui permettent cela (ne les ajoutez pas encore dans votre application) :

```
// défini un numéro unique pour repérer plus tard ce code
// dans la méthode onActivityResult(…)
private static final int SELECT_PICTURE = 124;


Intent intent = new Intent();
intent.setType("image/*");
intent.setAction(Intent.ACTION_GET_CONTENT);
```

Comme on peut le voir, on se contente de dire qu'on veut obtenir une image de n'importe quel type `(image/*)`

Pour que l'appareil photo fasse partie des applications proposées nous devons en plus créer un Intent de type IMAGE_CAPTURE et l'ajouter à la liste lors de la création du sélecteur. Voici ce que l'on doit ajouter aux lignes ci-dessus :

```
Intent captureIntent = new Intent(
        android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
intent.setAction(Intent.ACTION_PICK);
Intent chooserIntent = Intent.createChooser(intent, "Image Chooser");
chooserIntent.putExtra(Intent.EXTRA_INITIAL_INTENTS
        , new Parcelable[] { captureIntent });
startActivityForResult(chooserIntent, SELECT_PICTURE);
```

Pour récupérer l'image sélectionnée on procédera comme dans l'écran de connexion. Autrement dit c'est la méthode onActivityResult(…) qui sera appelé avec en paramètre un moyen de récupérer l'image.

Remettons à présent tout cela dans le bon ordre : Commencez par intercepter l'évènement Long Press sur l'image. Pour cela, repérez l'identifiant associé à *l'ImageView* et insérez les lignes suivantes à la fin de la méthode `OnCreate` de *UserActivity* :

```
ImageView imageView = (ImageView) findViewById(R.id.imageProfil);
imageView.setOnLongClickListener(new View.OnLongClickListener() {
    @Override
    public boolean onLongClick(View v) {
        ...
    }
});
```

A la place des …, placez les lignes de code définissant l'Intent de type image et lançant le sélecteur d'application que nous avons vu plus haut.

Vous pouvez à présent tester le code pour vérifier que le sélecteur d'application fonctionne et que vous pouvez sélectionner une image.

Reste maintenant à récupérer cette image dans le code. Nous l'avons dit, cette récupération va se faire dans la méthode `onActivityResult`. Attention de bien vérifier la valeur du `requestCode` pour distinguer le retour de l'activité d'authentification, de celui de l'activité de sélection d'images. Voici le code en question qui permet au final de récupérer l'image fourni par l'application (camera ou galerie par exemple) et de remplacer l'image du profil par cette image (ce code est à ajouter à la fin de la méthode `OnActivityResult`) :

```
if (requestCode == SELECT_PICTURE) {
    if (resultCode == RESULT_OK) {
        try {
            ImageView imageView = (ImageView) findViewById(R.id.imageProfil);
            boolean isCamera = (data.getData() == null);
            final Bitmap selectedImage;
            if (!isCamera) {
                final Uri imageUri = data.getData();
                final InputStream imageStream =
getContentResolver().openInputStream(imageUri);
                selectedImage = BitmapFactory.decodeStream(imageStream);
            }
            else {
                selectedImage = (Bitmap) data.getExtras().get("data");
            }
            // on redimensionne le bitmap pour ne pas qu'il soit trop grand
            Bitmap finalbitmap = Bitmap.createScaledBitmap(selectedImage, 500,
(selectedImage.getHeight() * 500) / selectedImage.getWidth(), false);
            imageView.setImageBitmap(finalbitmap);
        }
        catch (Exception e) {
            Log.v("AndroBoum",e.getMessage());
        };
    }
}
```

Notez la petite subtilité dans le code liée au fait que selon que l'image provient de la caméra ou de l'appareil, on ne récupère pas son *Bitmap* de la même façon.

Testez le code et vérifiez que l'image du profil est bien mise à jour avec l'image sélectionnée, quel que soit sa provenance.

Evidemment vous noterez que dès que l'application est redémarrée, l'image redevient celle par défaut. C'est normal puisque pour l'instant cette image de profil n'est sauvegardée nulle part. Nous pourrions ajouter le code nécessaire pour la stocker sur le téléphone mais l'objectif de cette application étant de faire interagir différents utilisateurs, nous allons stocker cette photo dans le cloud de façon à ce que tous les utilisateurs de l'application puissent y accéder.

Là encore *firebase* va nous aider car il propose des méthodes toute prêtes, aussi bien pour stocker des images (*firebase storage*) que pour gérer la liste des utilisateurs connectés (*firebase database*).

VI. Firebase Cloud Storage et Real Time Database

Objectif : Stocker de l'information dans le cloud.

Firebase Cloud Storage est un service de stockage en ligne dédié aux informations binaires comme des images ou des vidéos. *Firebase Real Time Database* est quant à lui un service de base de données accessible à travers une API NoSql. Nous en démontrerons l'usage au travers le stockage de la liste des utilisateurs connectés et des photos de leur profil.

1. Firebase Cloud Storage

Pour pouvoir utiliser le service, il faut inclure à notre projet la bibliothèque qu'il est lui a associé. Ajoutez donc dans le fichier `build.gradle (Module : app)` les lignes de dépendances suivantes :

```
compile 'com.google.firebase:firebase-storage:11.0.1'
compile 'com.firebaseui:firebase-ui-storage:2.0.1'
```

Dans le code, on obtient l'objet représentant le service en utilisant la ligne :

```
FirebaseStorage storage = FirebaseStorage.getInstance();
```

Cet objet dispose des méthodes nécessaires pour stocker et récupérer les objets stockés dans le cloud. Les objets sont organisés de la même façon que sur un disque dur. Ils disposent d'une racine dont on peut obtenir la référence en utilisant la ligne :

```
StorageReference storageRef = storage.getReference();
```

A partir de cette racine (`storageRef`), on peut créer un sous-dossier (appelé *child* dans la terminologie *firebase*) dont il faut donner le nom, par exemple :

```
imagesRef = storageRef.child("images");
```

Et pour placer un objet dans ce sous-dossier, on crée simplement un nouveau child du nom de l'objet par exemple :

```
photoRef = imagesRef.child("maphoto.jpg");
```

Pour uploader des données dans cet objet, on utilisera une des méthodes `putBytes()`, `putFile()`, ou `putStream()` selon d'où viennent les données (mémoire, fichier, ou flux). Par exemple, imaginons que les données de la photo soient en mémoire, on pourra écrire :

```
UploadTask uploadTask = photoRef.putBytes(data);
```

L'objet `uploadTask` contient des méthodes qui nous permettront ensuite d'être prévenu du déroulement de l'*upload*.

De façon symétrique, pour télécharger les données de l'objet, on utilisera une des méthodes `getBytes()`, `getStream()` ou `getDownloadUrl()`, cette dernière permettant simplement d'obtenir un lien « http » vers les données. Par exemple :

```
DownloadTask  task = photoRef.getFile(localFile);
```

Mettons cela en pratique pour uploader l'image de notre profil quand elle est mise à jour et pour la télécharger quand on se connecte pour la première fois. Nous placerons l'image dans un sous-dossier portant le nom de notre email. De cette façon, chaque utilisateur stockera sa photo de profil dans un sous-dossier différent.

Vous trouverez ci-dessous le code qui permet de télécharger et d'uploader une photo vers firebase. Ce code est adapté de la documentation officielle que l'on trouve à l'adresse :

https://firebase.google.com/docs/storage/android/start

```
private StorageReference getCloudStorageReference() {
    // on va chercher l'email de l'utilisateur connecté
    FirebaseAuth auth = FirebaseAuth.getInstance();
    if (auth == null) return null;
    String email = auth.getCurrentUser().getEmail();

    FirebaseStorage storage = FirebaseStorage.getInstance();
    StorageReference storageRef = storage.getReference();
    // on crée l'objet dans le sous-dossier de nom l'email
    StorageReference photoRef = storageRef.child(email + "/photo.jpg");
    return photoRef;
}

private void downloadImage() {
    StorageReference photoRef = getCloudStorageReference();
    if (photoRef == null) return;
    ImageView imageView = (ImageView) findViewById(R.id.imageProfil);
    // Load the image using Glide
    Glide.with(this /* context */).using(new FirebaseImageLoader())
            .load(photoRef)
            .skipMemoryCache(true)
            .diskCacheStrategy(DiskCacheStrategy.NONE)
            .placeholder(R.drawable.ic_person_black_24dp)
            .into(imageView);

}

private void uploadImage() {
   StorageReference photoRef = getCloudStorageReference();
    if (photoRef == null) return;
    // on va chercher les données binaires de l'image de profil
    ImageView imageView = (ImageView) findViewById(R.id.imageProfil);
    imageView.setDrawingCacheEnabled(true);
    imageView.buildDrawingCache();
    Bitmap bitmap = imageView.getDrawingCache();
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```
    bitmap.compress(Bitmap.CompressFormat.JPEG, 100, baos);
    byte[] data = baos.toByteArray();

    // on lance l'upload
    UploadTask uploadTask = photoRef.putBytes(data);
    uploadTask.addOnFailureListener(new OnFailureListener() {
        @Override
        public void onFailure(@NonNull Exception exception) {
            // si on est là, échec de l'upload
        }
    }).addOnSuccessListener(new
OnSuccessListener<UploadTask.TaskSnapshot>() {
        @Override
        public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
            // ok, l'image est uploadée
            // on fait pop un toast d'information
            Toast toast = Toast.makeText(getApplicationContext(),
getString(R.string.imageUploaded), Toast.LENGTH_SHORT);
            toast.show();
        }
    });
}
```

Ce code est composé de trois méthodes. La première `getCloudStorageReference()` est utilisé par les deux autres pour obtenir une référence à l'image stockée. La deuxième `downloadImage()` télécharge l'image stockée dans le cloud et la positionne dans l'*ImageView*. Enfin la troisième `uploadImage()` sauvegarde dans le cloud l'image contenue dans l'*ImageView*.

Vous remarquerez dans ce code l'utilisation d'un objet de type *Toast* qui permet d'afficher un message pendant un court instant à l'écran. On l'utilise aussi pour signaler à l'utilisateur que l'image de son profil a bien été sauvegardée.

Pour mieux comprendre ce code, vous pouvez également consulter la documentation de la bibliothèque *Glide* que nous utilisons ici pour positionner facilement les données téléchargées dans l'I*mageView* : https://github.com/bumptech/glide.

Recopiez ce code dans *UserActivity* et appelez les méthode `downloadImage()` et `uploadImage()` aux endroits adéquats pour que la gestion de votre photo de profil fonctionne correctement.

Testez votre code. Vous pouvez également utiliser la console de *firebase* pour vérifier que les images sont bien stockées sur cette plateforme.

2. Firebase Realtime Database

La *Firebase realtime Database* est une base de données stockée dans le cloud. Elle reprend les concepts d'organisation vu dans la section précédente mais pour stocker des données structurées par opposition à des données binaires. Elle ajoute également la possibilité pour les applications l'utilisant d'être informées en temps réel de la modification des données. Elle permet également un mode offline, dans la mesure où elle répliquée en local sur le téléphone.

Vous pouvez consulter sa documentation à l'adresse : https://firebase.google.com/docs/database/

Pour pouvoir utiliser ce service, ajoutez la dépendance suivante à votre projet :

```
compile 'com.google.firebase:firebase-database:11.0.1'
```

Nous l'utiliserons pour stocker la liste des utilisateurs enregistrés et connectés ce qui permettra de l'afficher dans l'application.

Pour cela nous allons commencer par créer dans notre application une classe métier qui nous permettra de spécifier toutes les propriétés du profil d'un utilisateur (à l'exception de sa photo de profil que nous avons gérée précédemment).

Pour l'instant, et pour simplifier, nous dirons qu'un utilisateur se résume à son email, son identifiant unique donné par *firebase* (son uid), et son statut de connexion (connecté ou déconnecté).

La classe Java représentant un objet de ce type peut s'écrire :

```
public class Profil {

    private String email;
    boolean isConnected;
    private String uid;

    public Profil() {
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public boolean isConnected() {
        return isConnected;
    }

    public void setConnected(boolean connected) {
        isConnected = connected;
    }

    public String getUid() {
        return uid;
    }

    public void setUid(String uid) {
        this.uid = uid;
    }
}
```

Ajoutez donc une classe *Profil* à votre projet sur le modèle ci-dessous.

La documention de *la firebase realtime database* nous apprend qu'on peut stocker directement un objet de ce type en précisant simplement le nom du *child* auquel il doit appartenir. Nous allons structurer la base de la façon suivante :

- Un objet du nom de `Users` qui sera créé à la racine ;
- Cet objet contiendra des *childs* qui porteront le nom de chaque uid (identifiant unique) des utilisateurs enregistrés ;

- Ces *childs* contiendront un objet de type *Profil* correspondant aux informations de l'utilisateur concerné.

Avec un telle structure, pour stocker un nouvel utilisateur représenté par la variable user de type *Profil* on pourra écrire :

```
DatabaseReference mDatabase = FirebaseDatabase.getInstance().getReference();
mDatabase.child("Users").child(user.getUid()).setValue(user);
```

Il ne reste plus qu'à modifier *UserActivity* pour qu'à chaque connexion d'un utilisateur son profil soit mise à jour dans la base. Inversement quand on quittera l'activité il faudra déclarer que l'utilisateur n'est plus connecté.

Voici une méthode utilitaire que vous pouvez ajouter à UserActivity pour construire un objet de type Profil à partir des informations d'authentification données par firebase (notez que la variable « user » est déclarée au niveau global pour être visible dans toute l'activité) :

```
private Profil user = new Profil();


private void setUser() {
    FirebaseAuth auth = FirebaseAuth.getInstance();
    FirebaseUser fuser = auth.getCurrentUser();
    if (fuser != null) {
        user.setUid(fuser.getUid());
        user.setEmail(fuser.getEmail());
        user.setConnected(true);
    }
}
```

Et voici la méthode permettant de mettre à jour le profil dans la base de données :

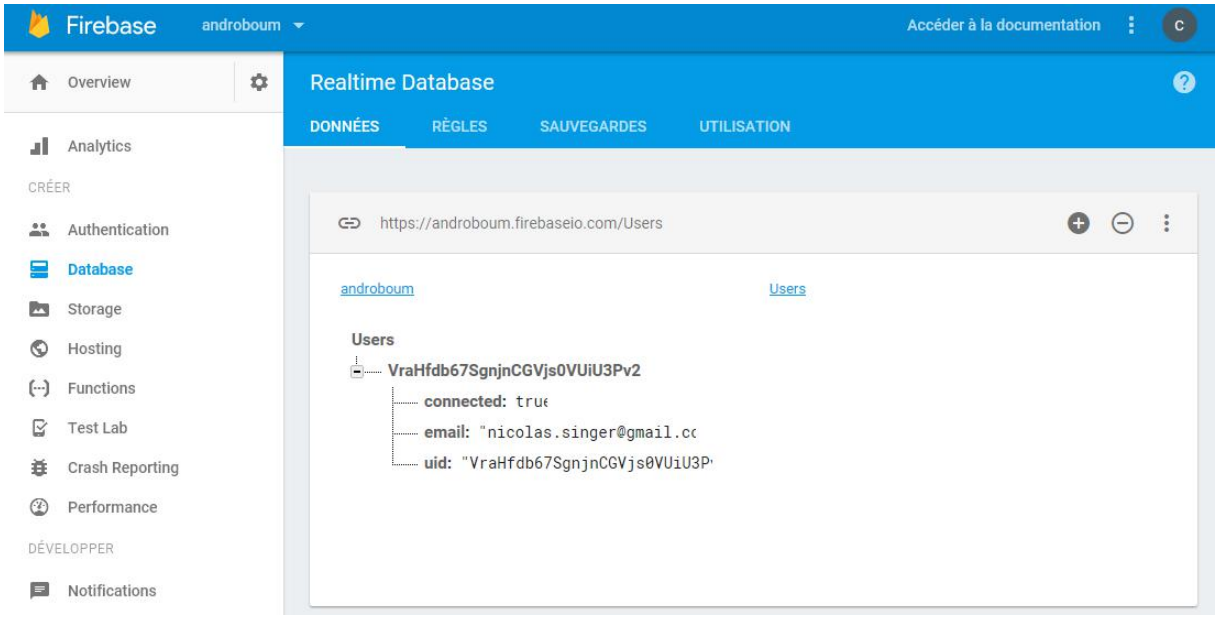
```
private void updateProfil(Profil user) {
    DatabaseReference mDatabase = FirebaseDatabase.getInstance().getReference();
    DatabaseReference ref = mDatabase.child("Users").child(user.getUid());
    ref.child("connected").setValue(true);
    ref.child("email").setValue(user.getEmail());
    ref.child("uid").setValue(user.getUid());
}
```

Il ne vous reste plus qu'à faire appel à ces méthodes pour que l'application fonctionne correctement, c'est-à-dire pour que le profil soit mis à jour à chaque nouvelle connexion. Pour intercepter la déconnexion, on pourra utiliser la méthode *onDestroy()* appelé par le cycle de vie de l'activité. Le code ci-dessous réalise cela en passant à faux la propriété « connected » de l'utilisateur (sans toucher aux autres propriétés).

```
@Override
protected void onDestroy() {
    user.setConnected(false);
    FirebaseAuth auth = FirebaseAuth.getInstance();
    if (auth != null) {
        FirebaseUser fuser = auth.getCurrentUser();
        if (fuser != null) {
            final FirebaseDatabase mDatabase = FirebaseDatabase.getInstance();
            DatabaseReference mreference =
                mDatabase.getReference().child("Users").child(fuser.getUid());
            mreference.child("connected").setValue(connectStatus);
        }
    }
```

```
    // on déconnecte l'utilisateur
    AuthUI.getInstance().signOut(this);
    super.onDestroy();
}
```

Pour vérifier que tout fonctionne, vous pouvez consulter le contenu de la base de données en vous connectant sur la console *firebase*, puis en cliquant sur *Database* dans le menu gauche.



Si tout va bien, vous pouvez même voir les données évoluer en temps réel au gré de vos connexions et déconnexions (et de celles de vos camarades).

VII. Liste des utilisateurs

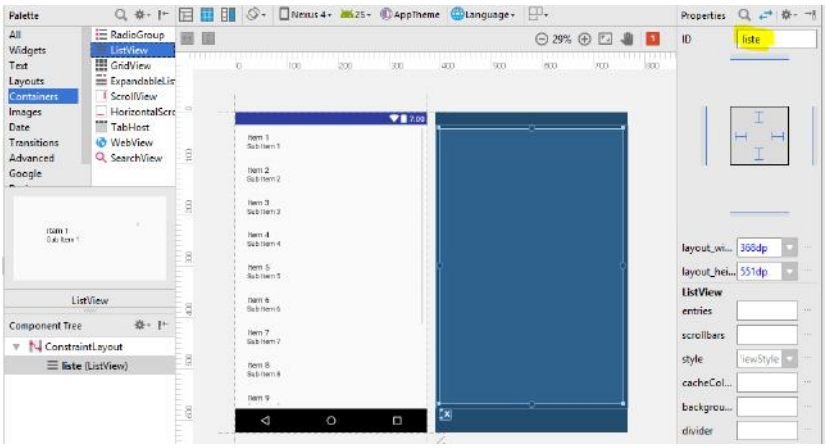
Objectif : Obtenir et afficher la liste des utilisateurs de l'application.

1. Le composant ListView

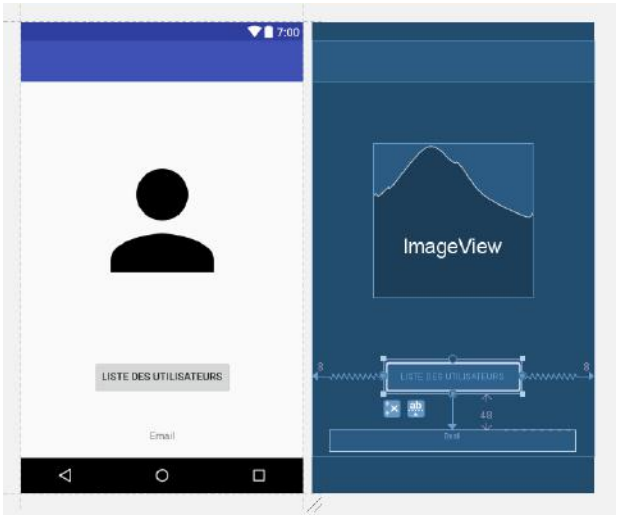
Dans cette section, nous allons créer une activité supplémentaire qui affichera la liste des utilisateurs enregistrés de l'application. Cette liste sera filtrable selon le statut (connecté ou non) de l'utilisateur.

Ajoutez à votre projet une nouvelle activité vide que vous appellerez User*ListActivity*.

C'est le composant `ListView` qui permet d'afficher une liste d'items. Modifiez donc le *layout* de *UserListActivity* pour y ajouter un composant de type `ListView` (vous le trouverez dans la catégorie « Containers » du mode Design) que vous doterez d'un identifiant (appelez le `liste`) :



Ajoutez au *layout* de *UserActivity* un bouton qui lancera l'activité *UserListActivity* quand on cliquera dessus. Placez-le entre l'image et l'email.



Sur le modèle de ce que nous avons déjà vu, ajoutez le code nécessaire pour qu'un clic sur le bouton lance *UserListActivity*.

Repassez à présent dans le code de *UserListActivity*. L'objectif va être de peupler le composant *ListView* avec la liste des utilisateurs enregistrés. Nous allons voir que cela passe par la définition d'un adaptateur mais d'abord il nous faut récupérer cette liste.

Dans la section précédente, nous avons appris comment stocker une valeur dans la base de données de *firebase*, voyons maintenant comment lire ces valeurs. La méthode utilisé par les bibliothèques de *firebase* est basée sur la mise en place d'un écouteur d'évènements qui sera appelée chaque fois que la valeur écoutée changera. Dans notre cas, nous allons donc mettre en place un écouter sur le *child* « `Users` » créé précédemment et nous seront avertis de toute modification de la liste de ses enfants, c'est-à-dire de la liste des utilisateurs de l'application.

Le code permettant de récupérer la liste des utilisateurs est le suivant :

```
final List<Profil> userList = new ArrayList<>();


DatabaseReference mDatabase = FirebaseDatabase.getInstance().getReference().child("Users");
ValueEventListener postListener = new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        userList.clear();
        for (DataSnapshot child : dataSnapshot.getChildren()) {
```

```
            userList.add(child.getValue(Profil.class));
        }
        adapter.notifyDataSetChanged();
    }

    @Override
    public void onCancelled(DatabaseError databaseError) {
        // Getting Post failed, log a message
        Log.v("AndroBoum", "loadPost:onCancelled", databaseError.toException());

    }
};

mDatabase.addValueEventListener(postListener);
```

Vous pouvez placer ce code à la fin de la méhode *OnCreate*() même si pour l'instant il ne fait rien d'autre que construire l'objet `userList`.

L'objectif à présent est de peupler le composant `ListView` du *layout* avec cette liste.

Pour ce faire, nous allons utiliser un objet intermédiaire appelé un *ArrayAdapter*. Cet objet va faire la liaison entre notre liste d'objets java (les profils) et la *ListView* en charge d'afficher tous les éléments de cette liste. Pour être plus précis, le rôle de cet intermédiaire est d'être capable de produire une vue (*View* au sens android du terme) pour chacun des éléments de la liste. Cette vue sera ensuite utilisée par la ListView pour afficher chaque élément. En ce qui nous concerne, et pour commencer, cette vue sera un simple TextView affichant l'email d'un utilisateur.

Maintenant que nous avons vu à quoi servait un ArrayAdapter, voyons comment on s'en sert. Nous devons définir une classe héritant de ce type d'objet et redéfinissant au moins ses méthodes getView() et getCount(). Voici le code qui réalise cette opération :

```
private class MyArrayAdapter extends ArrayAdapter<Profil> {
    List<Profil> liste;

    private MyArrayAdapter(Context context, int resource, List<Profil> liste) {
        super(context, resource, liste);
        this.liste = liste;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        TextView tv = new TextView(getContext());
        tv.setText(liste.get(position).getEmail());
        return tv;
    }

    @Override
    public int getCount() {
        return liste.size();
    }
}
```

Analysons ce code : Nous avons appelé notre classe `MyArrayAdapter`. Comme prévue, elle hérite de la classe `ArrayAdapter` et défini trois méthodes :

- un constructeur qui prend trois paramètres : Le contexte de l'application, une ressource android, et une liste d'objets de type `Profil`. Nous nous contentons dans ce constructeur de ranger la liste d'objets dans une variable appelée `liste`.
- la méthode *getView* qui prend elle aussi trois paramètres : Le numéro de l'item à afficher et deux autres paramètres qui pour l'instant ne nous intéressent pas. Cette méthode crée un nouveau `TextView` et initialise sont contenu (méthode `setText`()) avec l'email de

l'utilisateur correspondant au numéro passé en paramètre. Elle retourne ensuite ce *TextView*.

- La méthode *getCount* qui doit retourner le nombre d'items à afficher, c'est-à-dire `liste.size()`.

C'est donc un objet de ce type qui sera utilisé pour peupler notre ListView et voici les lignes de code qui réalisent cette opération :

```
ListView listeView = (ListView) findViewById(R.id.liste);
final MyArrayAdapter adapter = new MyArrayAdapter(this,android.R.layout.simple_list_item_1,
userList);
listeView.setAdapter(adapter);
```

La première ligne va chercher le `ListView`, la deuxième crée un nouvel objet de type `MyArrayAdapter`. La dernière positionne cet objet dans le *ListView*.

Voici donc le code complet de l'activité `ListActivity` :

```
public class UserListActivity extends AppCompatActivity {

    private class MyArrayAdapter extends ArrayAdapter<Profil> {
        List<Profil> liste;

        private MyArrayAdapter(Context context, int resource, List<Profil> liste) {
            super(context, resource, liste);
            this.liste = liste;
        }

        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
            TextView tv = new TextView(getContext());
            tv.setText(liste.get(position).getEmail());
            return tv;
        }

        @Override
        public int getCount() {
            return liste.size();
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        final List<Profil> userList = new ArrayList<>();

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_user_list);

        ListView listeView = (ListView) findViewById(R.id.liste);
        final MyArrayAdapter adapter = new
MyArrayAdapter(this,android.R.layout.simple_list_item_1, userList);
        listeView.setAdapter(adapter);


        DatabaseReference mDatabase =
FirebaseDatabase.getInstance().getReference().child("Users");
        ValueEventListener postListener = new ValueEventListener() {
            @Override
            public void onDataChange(DataSnapshot dataSnapshot) {
                userList.clear();
                for (DataSnapshot child : dataSnapshot.getChildren()) {

                    userList.add(child.getValue(Profil.class));
                }
                adapter.notifyDataSetChanged();
            }

            @Override
```

```
            public void onCancelled(DatabaseError databaseError) {
                // Getting Post failed, log a message
                Log.v("AndroBoum", "loadPost:onCancelled", databaseError.toException());

            }
        };


        mDatabase.addValueEventListener(postListener);
    }
}
```

La seule ligne que nous n'avions pas encore vu dans ce code est l'instruction `adapter.notifyDataSetChanged();` qui notifie l'adaptateur que la liste a changée. C'est le cas lors du tout premier chargement, mais aussi à chaque fois qu'un nouvel utilisateur s'enregistre.

Testez ce code et vérifiez que la liste s'affiche correctement.

2. Construire les items d'un ListView

Rendons un peu plus sympa l'affichage de la liste des utilisateurs. Pour cela nous souhaitons afficher en plus de son email, une version miniature de l'image de son profil, ainsi que son statut de connexion sous la forme d'une icône.

Créez donc un nouveau *layout* de type `ConstraintLayout` que vous appellerez `profil_list_item.xml`. Ajoutez à ce *layout*, une *ImageView* qui contiendra l'image du profil en petit format (vous pouvez utiliser l'icône *person* pour cela). Ajoutez ensuite un `TextView` pour afficher l'email et un deuxième *ImageView* qui affichera une icône si l'utilisateur est connecté. Pour cette icône nous utiliserons l'asset material design « *verified user* » que vous devez ajouter à votre projet comme nous l'avons fait précédemment. Placez ces différents éléments les uns à côté des autres, pour obtenir quelque chose ressemblant à la copie écran ci-après :

Affectez aux trois éléments les identifiants suivants : `imageView`, `textView`, et `imageView2`.

Revenez dans le méthode `getView()` de la classe `MyArrayAdapter`. Cette fois au lieu de renvoyer un simple TextView, nous allons renvoyer le layout que nous venons de concevoir, en l'ayant préalablement rempli avec les informations du profil, image comprise. Si l'utilisateur n'est pas connecté, nous règlerons la visibilité de l'icône de connexion sur invisible, pour qu'elle n'apparaisse pas.

Voici le code qui permet de réaliser cela. Vous noterez au passage comment on instancie un *layout* dynamiquement pour récupérer l'objet qui lui correspond. C'est la méthode `View.inflate()` qui réalise cela.

Voici le nouveau code la méthode `getView()` de la classe `MyArrayAdapter` :

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

    // on va chercher le bon profil dans la liste
```

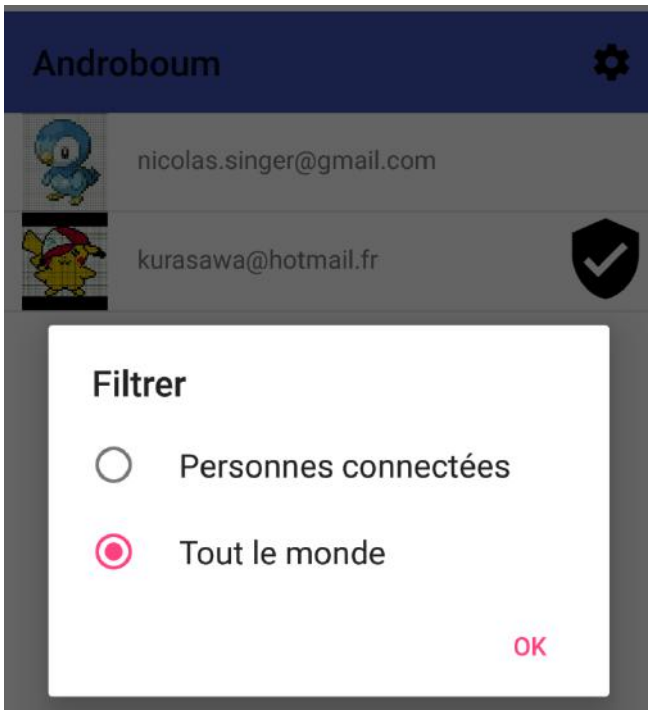
```
    Profil p = liste.get(position);

    // on instancie le layout sous la forme d'un objet de type View
    View layout = View.inflate(getContext(), R.layout.profil_list_item, null);


    // on va chercher les trois composants du layout
    ImageView imageProfilView = (ImageView) layout.findViewById(R.id.imageView);
    TextView textView = (TextView) layout.findViewById(R.id.textView);
    ImageView imageConnectedView = (ImageView) layout.findViewById(R.id.imageView2);

    // on télécharge dans le premier composant l'image du profil
    StorageReference photoRef = storage.getReference().child(p.getEmail() + "/photo.jpg");
    if (photoRef != null) {
        Glide.with(getContext()).using(new FirebaseImageLoader())
                .load(photoRef)
                .skipMemoryCache(true).diskCacheStrategy(DiskCacheStrategy.NONE)
                 .placeholder(R.drawable.ic_person_black_24dp)
                 .into(imageProfilView);
    }

    // on positionne le email dans le TextView
    textView.setText(p.getEmail());

    // si l'utilisateur n'est pas connecté, on rend invisible le troisième
    // composant
    if (!p.isConnected) {
        imageConnectedView.setVisibility(View.INVISIBLE);
    }

    // on retourne le layout
    return layout;
}
```

Testez ce code, vous devez obtenir une jolie liste avec tous les utilisateurs du système, photos de profil et statut de connexion compris.

3. Filtrer les items d'une liste

Nous allons proposer un moyen de filtrer les utilisateurs de la liste selon leur statut de connexion. Pour cela commençons par introduire une nouvelle brique de la construction d'interfaces sous android : la boite de dialogue. Ce composant permet de faire *popper* au-dessus de l'interface une fenêtre permettant de réaliser un choix parmi une liste d'options.

La documentation disponible à l'adresse https://developer.android.com/guide/topics/ui/dialogs.html explique le principe de leur construction.



Pour concevoir une telle boite, vous devez commencer par créer un nouveau fichier de valeurs dans le dossier *value* des ressources, que vous appellerez `arrays.xml`. Tant que vous y êtes, créez aussi sa version en anglais comme nous l'avons fait pour le fichiers `strings.xml`.

Placez dans ces deux fichiers une liste de valeurs qui devra apparaître comme choix possible dans la boite de dialogue. Voici un exemple de ce que vous pouvez mettre dans le fichier en français :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="users_filter">
        <item>Personnes connectées</item>
        <item>Tout le monde</item>
    </string-array>
</resources>
```

Et dans celui en anglais :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="users_filter">
        <item>Only Connected</item>
        <item>All Users</item>
    </string-array>
</resources>
```

Ceci fait ajoutez dans la classe UserListActivity, une nouvelle méthode qui s'occupera de créer, d'afficher, et de gérer les choix faits dans cette boite de dialogue. Voici le code commenté de cette méthode :

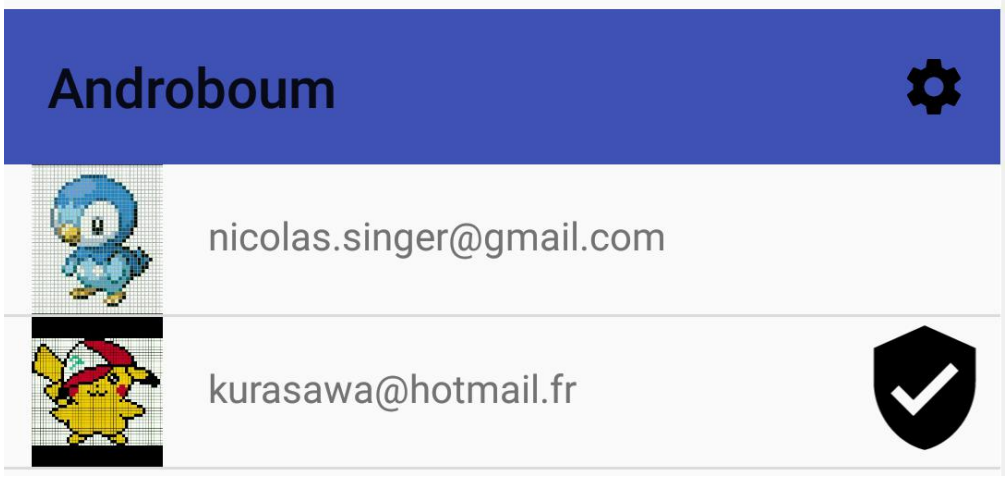
```
// variable globale pour spécifier si on filtre ou pas
boolean filterConnected = false;
```

```
private void showFilterDialog() {
    // on crée un nouvel objet de type boite de dialogue
    AlertDialog.Builder builder = new AlertDialog.Builder(this);

    // on lui affecte un titre, et une liste de choix possibles
    builder.setTitle(R.string.filter_dialog_title)
            .setSingleChoiceItems(R.array.users_filter, filterConnected?0:1, new
DialogInterface.OnClickListener() {
                @Override
                // méthode appelée quand l'utilisateur fait un choix
                // i contient le numéro du choix
                public void onClick(DialogInterface dialogInterface, int i) {
                    // si le premier item a été choisie, on filtre sur
                    // uniquement les utilisateurs connectés.
                    filterConnected = (i == 0);
                    // et on signale a l'adaptateur qu'il faut remettre
                    // la liste à jour.
                    adapter.notifyDataSetChanged();
                }
            })
            .setPositiveButton("Ok", new DialogInterface.OnClickListener() {
                // on a cliqué sur "ok", on ne fait rien.
                public void onClick(DialogInterface dialog, int id) {

                }
            });

    // on crée la boite
    AlertDialog dialog = builder.create();
    // et on l'affiche
    dialog.show();

}
```

Pensez également à rendre globale la variable `adapter`.

Pour appeler cette méthode, ajoutez une *ToolBar* au *layout* de *UserListActivity*, dotée d'un bouton d'action qui l'appellera. Pour réaliser cette opération, reportez-vous à ce que nous avons fait précédemment pour la *ToolBar* de *UserActivity*. Pour l'icône du bouton d'action, vous pouvez par exemple utiliser l'icône material design « *settings* » que vous devrez donc ajouter à votre projet.

Voici un exemple de ce que à quoi pourra ressembler votre interface finale :



Faites donc en sorte qu'un clic sur l'action de la *toolbar* appelle la méthode *showFilterDialog()* et la boite de dialogue devrait s'ouvrir.

Il faut maintenant modifier la classe `MyArrayAdapter` pour qu'elle filtre les utilisateurs en fonction de la valeur de la variable `filterConnected`. Un des moyens de faire cela est

d'intercepter l'appel `notifyDataSetChanged()` en surchargeant (*override*) la méthode `notifyDataSetChanged()` et en modifiant la liste des utilisateurs en fonction du filtrage. Pour cela on gardera une copie de la liste complète dans une variable (`origListe`) et on modifiera la liste sur laquelle les méthodes de l'adaptateur agissent en fonction du filtrage.

Voici le code modifié qui réalise cela :

```
private class MyArrayAdapter extends ArrayAdapter<Profil> {
    List<Profil> liste, origListe;
    FirebaseStorage storage = FirebaseStorage.getInstance();

    private MyArrayAdapter(Context context, int resource, List<Profil> liste) {
        super(context, resource, liste);
        this.liste = liste;
        // on fait une copie de la liste dans une autre variable
        this.origListe = this.liste;

    }

    @Override
    public void notifyDataSetChanged() {
        if (filterConnected) {
            // on alloue une nouvelle liste et on la remplit uniquement
            // avec les utilisateurs connectés.
            liste = new ArrayList<>();
            for (Profil p : origListe) {
                if (p.isConnected()) liste.add(p);
            }
            // sinon on reprend la liste complète que l'on avait
            // sauvegardé dans la variable origListe.
        } else liste = origListe;
        super.notifyDataSetChanged();
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
   // suite du code identique
```

Testez ce code, le filtrage doit fonctionner.

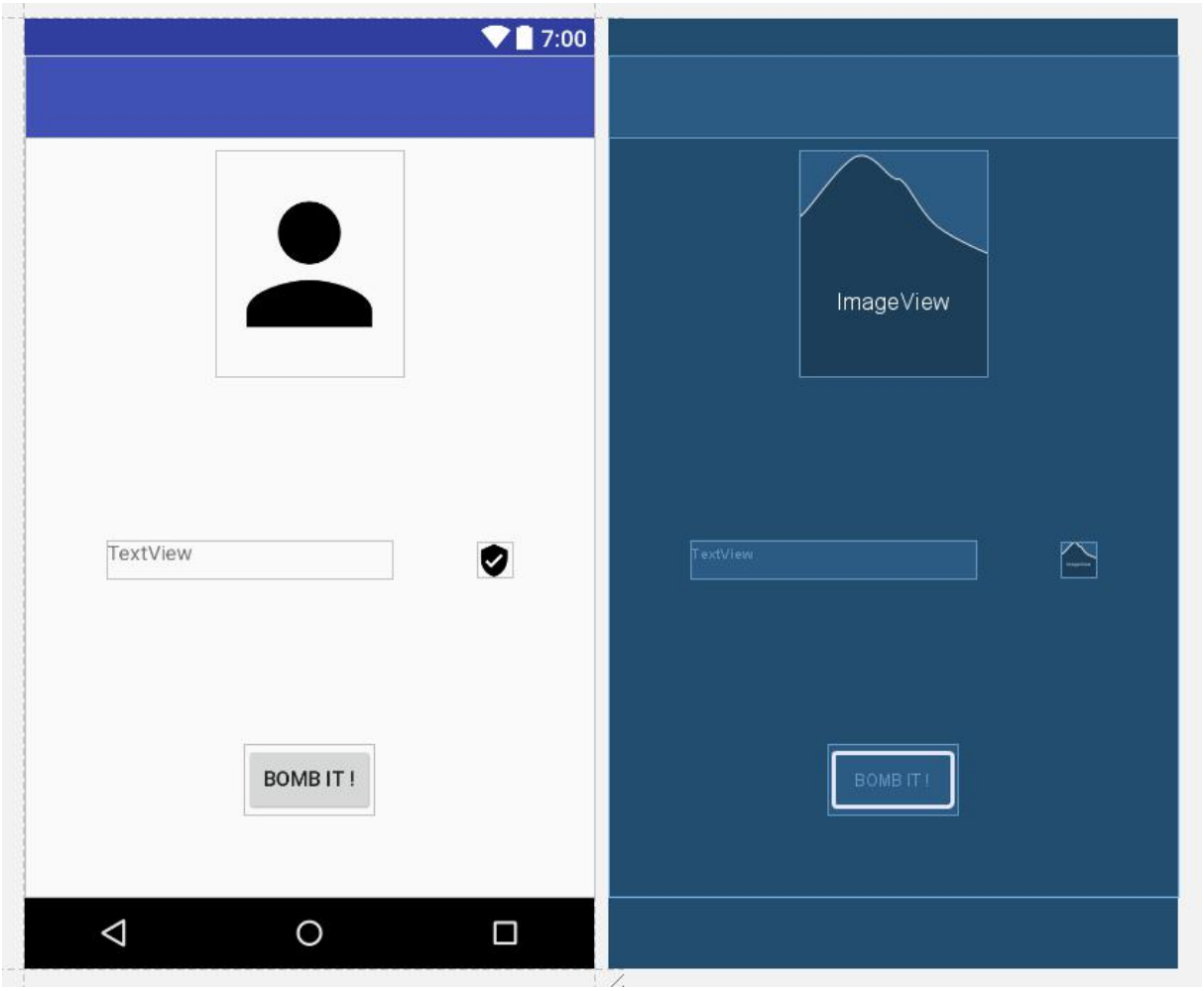
VIII. Interaction avec un utilisateur

Objectif : Réaliser de l'interface de ciblage d'un utilisateur.

Dans cette partie nous allons construire une nouvelle activité qui permettra d'interagir avec le profil choisi. Cette activité sera lancée quand l'utilisateur cliquera sur un des utilisateurs de la liste. Elle affichera en plein écran les informations du profil cliqué, un peu comme dans UserActivity mais avec des éléments supplémentaires.

1. Affichage d'un seul utilisateur de la liste

Commencez donc par créer une nouvelle activité vide que vous nommerez *OtherUserActivity*. Concevez son *layout* pour qu'il soit composé d'une *ToolBar* (pour l'instant vide), d'une image (qui contiendra l'image du profil), d'un texte (qui contiendra l'email) éventuellement suivi de l'icône de connexion, et d'un bouton « Bomb it » qui permettra plus tard de poser une bombe chez l'utilisateur visé. L'interface doit ressembler à la proposition ci-dessous :



Maintenant que nous avons une activité affichant un profil complet, nous voulons ajouter le code nécessaire dans *UserListActivity* à ce qu'un *clic* sur une des intox de la liste ouvre *OtherUserActivity*. Pour cela il nous faut faire deux choses :

- Ajouter un gestionnaire d'évènement `onItemClickListener` sur la *ListView* pour pouvoir spécifier quoi faire en cas de sélection d'une intox ;
- Dans ce gestionnaire d'évènements, écrire le code nécessaire à lancer l'activité d'affichage en plein écran avec en paramètre le numéro de l'intox à afficher.

Pour le premier point, on positionne le gestionnaire d'évènement par les lignes suivantes (avec la variable liste référençant le *ListView*) :

```
listeView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> adapterView, View view, int position, long l) {
        // code exécuté quand on clique sur un des items de la liste.
        // le paramètre position contient le numéro de l'item cliqué.
    }
});
```

Pour lancer une nouvelle activité, vous savez qu'on doit d'abord construire un `Intent`. Ce que nous n'avons pas encore vu c'est que l'Intent permet de préciser des paramètres (sous la forme d'objets de type simple ou de type `Bundle`) que l'activité lancée pourra réceptionner. L'usage que nous en ferons est de passer le numéro du profil cliqué à l'activité lancée. Pour préciser un paramètre, on utilise la méthode `putExtra()` de l'Intent qui permet de préciser le nom d'une clé et sa valeur. En ce qui nous concerne nous voulons simplement passer en paramètre le numéro de l'intox à afficher.

Voici comment on crée l'Intent avec son paramètre et comment on lance l'activité :

```
Intent intent = new Intent(context,OtherUserActivity.class);
intent.putExtra("position",position);
startActivity(intent);
```

Au final voici le code complet exécuté en cas de clic. Vous noterez que pour pouvoir préciser le contexte en premier paramètre du constructeur (à l'intérieur du callback *onItemClickListener*), on le stocke dans une variable avant l'appel du callback.

```
// on sauve le contexte dans une constante pour pourvoir s'en servir
// dans le callback.
final Context context = this;

listeView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> adapterView, View view, int
position, long l) {
        // code exécuté quand on clique sur un des items de la liste.
        // le paramètre position contient le numéro de l'item cliqué.
        Intent intent = new Intent(context,OtherUserActivity.class);
        intent.putExtra("position",position);
        startActivity(intent);
    }
});
```

Passons à présent à l'activité *OtherUserActivity*. Celle-ci doit récupérer le paramètre passé par l'activité appelante (donc la position du profil à afficher), aller chercher la liste des utilisateurs, et remplir le *layout* avec les informations du bon profil.

Voici comment cette activité peut récupérer le paramètre dans sa méthode onCreate :

```
// on obtient l'intent utilisé pour l'appel
Intent intent = getIntent();
// on va chercher la valeur du paramètre position, et on
// renvoie zéro si ce paramètre n'est pas positionné (ce qui ne devrait
// pas arriver dans notre cas).
int position = intent.getIntExtra("position",0);
```

A vous de jouer maintenant pour compléter le code pour :

- Aller chercher à nouveau la liste des utilisateurs, exactement comme nous l'avons fait dans le chapitre précédent.
- Une fois cette liste obtenue, remplir le *layout* avec les informations de l'utilisateur de numéro *position* comme nous l'avons déjà fait plusieurs fois.
- Afficher ou pas l'icône de connexion en fonction du statut connecté de l'utilisateur (Remarque : pour un rendu plus joli vous pouvez choisir de régler la visibilité sur View.GONE au lieu de View.INVISIBLE. Le premier enlève l'élément quand le deuxième le conserve à sa place sans l'afficher).

2. Swipe pour parcourir les utilisateurs

Nous souhaitons à présent pouvoir passer d'un profil à un autre tout en restant en affichage plein écran, en utilisant le geste de *swipe*, qui consiste à balayer l'écran de droite à gauche ou de gauche à droite avec le doigt, de façon à pouvoir passer au profil précédent ou suivant.

Si Android permet de gérer finement le repérage des gestes de l'utilisateur, il met aussi à la disposition du programmeur quelques vue prédéfinies pour gérer certains de ces gestes. C'est le cas de la vue *ViewPager*. Cette vue permet de gérer le passage d'une vue à une autre selon le même mécanisme que la *ListView* vue précédemment mais sous la forme d'une succession d'écrans accessibles par glissement du doigt.

La documentation officielle accessible à l'adresse :

http://developer.android.com/training/animation/screen-slide.html

donne plus de détails à ce sujet.

Néanmoins la méthode donnée passe par les fragments qui sont une sorte de sous-activités. Cela complique beaucoup le code, c'est pourquoi nous allons procéder un peu différemment.

Pour pouvoir utiliser le *ViewPager*, nous allons restructurer le *layout* de *OtherUserActivity*.

Créez un nouveau *layout* (en passant par le menu contextuel du dossier `layout` puis *new Layout Resource File*), et appelez-le `other_user_fragment.xml`. Placez-y le contenu actuel de `activity_other_user.xml`.

Dans le layout `activity_other_user.xml`, remplacez l'ensemble des lignes par les lignes ci-dessous :

```
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
android:layout_height="match_parent"
    android:id="@+id/pager">

</android.support.v4.view.ViewPager>
```

L'idée de ce que nous venons de faire, est de définir le *layout* de l'activité plein écran comme étant un *ViewPager* initialement vide. Le fragment de *layout* que nous avons copié dans `other_user_fragment.xml` sera instancié par chaque page dans le *ViewPager* au fur et à mesure du parcours des intox.

Tout comme nous avions utilisé un *ArrayAdapter* pour afficher les éléments de la *ListView*, nous allons ici utiliser un *PagerAdapter* pour afficher les éléments du *PageView*. Là encore, le *PagerAdapter* va jouer le rôle d'intermédiaire entre le *ViewPager* et les intox. Son rôle est de générer dynamiquement le *layout* de l'intox à afficher.

Pour implémenter un objet de ce type, il faut au minimum définir quatre méthodes. La plus importante s'appelle `instantiateItem`. Elle est appelée à chaque fois qu'une nouvelle Intox doit devenir visible. Sa tâche principale est de mettre en page l'intox. Voici son code commenté :

```
@Override
public Object instantiateItem(ViewGroup container, int position) {

    // on va chercher la layout
    ViewGroup layout = (ViewGroup) View.inflate(context,
```

```
R.layout.other_user_fragment, null);
     // on l'ajoute à la vue
     container.addView(layout);
     // on le remplit en fonction du profil
     remplirLayout(layout, liste.get(position));
     // et on retourne ce layout
     return layout;
}
```

La deuxième méthode à implémenter s'appelle `getCount()`, elle doit simplement renvoyer le nombre d'éléments de la liste d'intox.

La troisième est `destroyItem()`. Elle est appelée quand la vue représentant un profil peut être détruite. Son code va donc supprimer son *layout*.

La quatrième et dernière est `isViewFromObject()`, qui doit déterminer si les deux vues passées en paramètre sont les mêmes.

Voici donc le code complet de notre *PagerAdapter*. Notez que nous l'avons doté d'un constructeur qui prend en paramètre notre liste d'utilisateurs et un objet de type `Context`, pour qu'il puisse par la suite accéder à ces deux objets.

```
class MyPagerAdapter extends PagerAdapter {

    List<Profil> liste;
    Context context;

    public MyPagerAdapter(Context context, List<Profil> liste) {
        this.liste = liste;
        this.context = context;
    }

    @Override
    public Object instantiateItem(ViewGroup container, int position) {

        // on va chercher la layout
        ViewGroup layout = (ViewGroup) View.inflate(context, R.layout.other_user_fragment,
null);
        // on l'ajoute à la vue
        container.addView(layout);
        // on le remplit en fonction du profil
        remplirLayout(layout, liste.get(position));
        // et on retourne ce layout
        return layout;
    }

    @Override
    public int getCount() {
        return liste.size();
    }

    @Override
    public void destroyItem(ViewGroup container, int position, Object object) {
        container.removeView((View) object);
    }

    @Override
    public boolean isViewFromObject(View view, Object object) {
        return view == object;
    }

    private void remplirLayout(ViewGroup layout,Profil p) {
        ImageView imageView = (ImageView) layout.findViewById(R.id.imageView);
        ImageView imageView2 = (ImageView) layout.findViewById(R.id.imageView2);
        TextView textView = (TextView) layout.findViewById(R.id.textView);

        // on télécharge dans le premier composant l'image du profil
        FirebaseStorage storage = FirebaseStorage.getInstance();
```

```
        StorageReference photoRef = storage.getReference().child(p.getEmail() + "/photo.jpg");
        if (photoRef != null) {
            Glide.with(context).using(new FirebaseImageLoader())
                    .load(photoRef)
                     .skipMemoryCache(true).diskCacheStrategy(DiskCacheStrategy.NONE)
                     .placeholder(R.drawable.ic_person_black_24dp)
                     .into(imageProfilView);
        }

        if (!p.isConnected()) {
            imageView2.setVisibility(View.GONE);
        }

        // on positionne le email dans le TextView
        textView.setText(p.getEmail());
        Log.v("Androboum","bingo"+p.getEmail()) ;
    }
}
```

Il reste à modifier l'activité *OtherUserActivity* pour qu'elle utilise correctement l'adaptateur. Voici le nouveau code de sa méthode *onCreate* :

```
final List<Profil> userList = new ArrayList<>();
final MyPagerAdapter adapter = new MyPagerAdapter(this, userList);
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_other_user);

final ViewPager pager = (ViewPager) findViewById(R.id.pager);

// on obtient l'intent utilisé pour l'appel
Intent intent = getIntent();
// on va chercher la valeur du paramètre position, et on
// renvoie zéro si ce paramètre n'est pas positionné (ce qui ne devrait
// pas arriver dans notre cas).
final int position = intent.getIntExtra("position",0);


DatabaseReference mDatabase = FirebaseDatabase.getInstance().getReference().child("Users");
ValueEventListener postListener = new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        userList.clear();
        for (DataSnapshot child : dataSnapshot.getChildren()) {
            userList.add(child.getValue(Profil.class));
        }
        adapter.notifyDataSetChanged();
        pager.setCurrentItem(position);
    }

    @Override
    public void onCancelled(DatabaseError databaseError) {
        // Getting Post failed, log a message
        Log.v("AndroBoum", "loadPost:onCancelled", databaseError.toException());

    }
};

mDatabase.addValueEventListener(postListener);
pager.setAdapter(adapter);
```

Le code va chercher le *ViewPager*, positionne son adaptateur à une nouvelle instance de `MyPageAdapter`, et quand la liste des utilisateurs a été téléchargée, positionne l'adaptateur sur le profil dont on a récupéré le numéro comme précédemment.

Testez l'application ainsi modifiée, et vérifiez que vous pouvez parcourir les intox par glissement latéral du doigt sur l'écran.

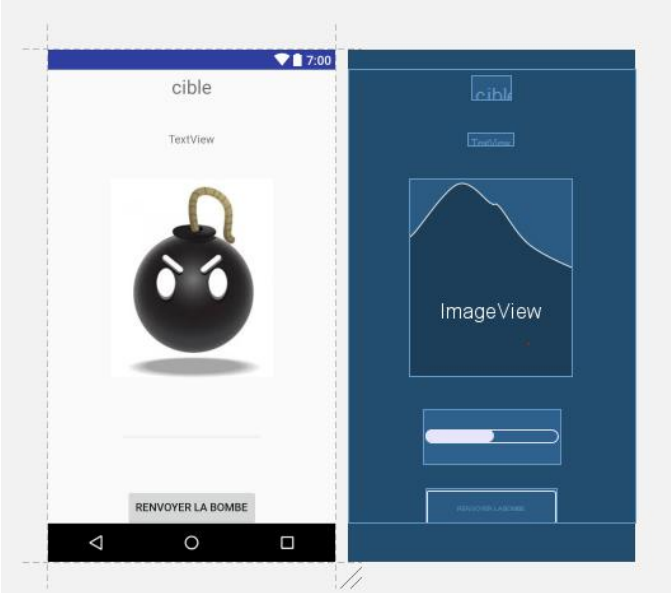
IX. Envoi et reception des bombes

Objectif : permettre l'envoi et la réception des bombes entre utilisateurs.

Dans cette section, nous allons ajouter au projet le code nécessaire à la gestion de l'envoi et de la réception des bombes entre appareils.

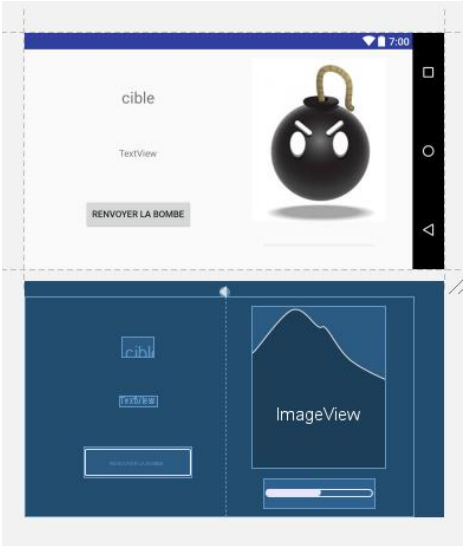
1. L'activité BombActivity

Créez une nouvelle activité vide que vous appellerez *BombActivity*. Concevez son layout pour qu'il fasse apparaître deux champs texte, une image, un bouton, et une barre de progression (ProgressBar). Vous pouvez vous inspirer du modèle donné ci-dessous :



Prêtez attention aux identifiants que vous donnerez aux différents éléments en appelant les champs respectivement du haut vers le bas : `textViewOther`, `cible`, `imageView`, `progressBar` et `button`. Vous prévoirez également une version française et anglaise pour les textes du champ « cible » et du bouton.

Une fois ce premier layout réalisé, faites-en une adaptation en mode *landscape* pour que les éléments soient correctement présentés en mode paysage. Cela pourra ressembler à la version ci-dessous :



Laissons pour l'instant de côté le code de *BombActivity* et intéressons-nous aux modifications à apporter au profil de l'utilisateur pour permettre la gestion des bombes.

2. Modification de Profil.java

Nous allons ajouter aux propriétés d'un utilisateur, les informations nécessaires à l'envoi et la réception des bombes.

Pour commencer, on posera comme contrainte qu'un utilisateur ne pourra cibler qu'un seul utilisateur à la fois. Inversement il ne pourra pas être la cible de plusieurs utilisateurs en même temps.

Dès lors, concernant une bombe, un utilisateur pourra être dans quatre états différents :

- L'état normal (IDLE), c'est-à-dire ni ciblé par une bombe, si n'ayant déposé de bombes.
- L'état bombeur (BOMBER) s'il a ciblé quelqu'un d'autre avec une bombe.
- L'état bombé (BOMBED) quand il aura été ciblé par quelqu'un.
- L'état « en attente d'acquittement » (WAITING) s'il vient d'être ciblé par une bombe mais qu'il n'a pas encore acquitté la « bonne » réception de celle-ci.

Le quatrième état est nécessaire pour confirmer qu'un utilisateur ciblé par une bombe est bien en ligne. L'attaquant attendra donc deux secondes que la cible confirme la réception de la bombe. Si ce délai s'écoule sans acquittement, la cible sera considérée comme « hors ligne » et la bombe ne pourra pas être déposée.

Un utilisateur dans l'état « bombé » a la possibilité de renvoyer la bombe à l'expéditeur. S'il le fait avant que la bombe n'explose, les deux utilisateurs s'échangent les rôles (le bombeur devient bombé et le bombé devient bombeur) et on repart pour un tour avec le temps avant explosion remis à sa valeur initiale.

Si elle n'est pas renvoyée à temps (ce temps est réglé sur dix secondes dans cette version du jeu), la bombe explose, l'attaquant marque un point, la cible en perd un, et les deux utilisateurs repassent dans l'état normal.

En plus de ce statut, chaque utilisateur devra donc mémoriser son score et l'identité de l'autre (cible ou attaquant) sous la forme de son email et de son identifiant unique (uid).

Voilà pourquoi il vous faut modifier la classe Profil. Java pour qu'au final elle contienne les propriétés suivantes :

```
public class Profil  {

    // les différents status d'un utilisateur
    enum BombStatut { IDLE, AWAITING, BOMBER, BOMBED };

    // mon email
    private String email;
    // mon statut de connexion (vrai ou faux)
    boolean isConnected;
    // mon identifiant unique
    private String uid;
    // mon statut actuel
    private BombStatut statut = BombStatut.IDLE;
    // l'identifiant de mon adversaire
    private String otherUserUID;
    // l'email de mon adversaire
    private String otheruseremail;
    // mon score
    private int score = 0;
```

Vous prendrez soin de doter chacune de ses propriétés des *getters* et *setters* associés.

3. Création d'une classe d'application.

Jusqu'à présent nous avons vu qu'un programme Android était composé d'activités qui représentent les différents écrans de l'application. Ces activités ont leur propre cycle de vie ce qui rend difficile le partage d'informations entre elles.

Or la réception d'une bombe doit pouvoir se faire quelque-soit l'activité en cours. Nous pourrions certes faire appel à ce code dans chacune des activités, mais il est plus efficace de le faire s'exécuter dans un contexte commun à toutes les activités : c'est le contexte de l'application. Pour pouvoir faire appel à ce contexte, ajoutez une nouvelle classe à votre projet que vous appellerez *AndroBoumApp.java*. Cette classe doit hériter de la classe Application comme illustré ci-dessous :

```
public class AndroBoumApp extends android.app.Application {
}
```

Vous devez ensuite modifier le contenu du fichier *manifest* pour dire à Android de charger cette classe au démarrage. Il faut pour cela ajouter un attribut « name » à la balise « application ». Voici la modification à opérer :

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:name=".AndroBoumApp"
    android:theme="@style/AppTheme">
```

Ceci fait, lors du démarrage de votre application, le système va instancier un objet du type *AndroBoum* et le conservera tant que l'application ne sera pas détruite. Cela va donc nous permettre de maintenir un état qui sera transversal aux activités.

Pour être plus précis, c'est cet objet qui s'occupera de détecter les changements de statut de votre utilisateur ce qui permettra en particulier d'avertir l'utilisateur qui est ciblé par une bombe. Cet avertissement prendra la forme d'une notification qui apparaîtra en cas de ciblage. L'utilisateur ciblé a alors 10 secondes pour cliquer sur cette notification, ce qui lancera l'activité BombActivity » et ce qui lui permettra peut-être d'avoir le temps de renvoyer la bombe à l'expéditeur.

Tant que vous êtes dans le fichier *manifest*, nous devons également procéder à une autre modification, celle-ci concernant cette fois l'activité *BombActivity*. La modification consiste à ajouter un attribut « *android:launchMode* » dont la valeur va expliquer à Android de ne pas créer plusieurs instances de cette activité, même si elle est lancée plusieurs fois. Voici la modification à opérer :

```
<activity android:name=".BombActivity" android:launchMode="singleTask">
</activity>
```

La raison de cette modification est que l'activité *BombActivity* va maintenir un compte à rebours du temps qu'il reste à la bombe avant d'exploser. Si jamais on quitte cette activité pour y revenir ensuite, il faut que le compte à rebours continue son œuvre, et pas qu'une nouvelle activité soit créée.

4. Finalisation et explication du système de bombes

Le code qui gère l'échange de bombes est un peu délicat à mettre au point car les transitions entre les différents états doivent être correctement gérés, et un certain nombre de cas particuliers viennent compliquer les choses. Il est aussi indispensable que chacune de vos applications gèrent de la même façon l'envoi et la réception des bombes, sinon elles ne pourront pas communiquer correctement.

C'est pourquoi ce code vous est directement donné sous la forme du contenu des classes *BombActivity* et *AndroBoumApp*, et également sous la forme d'un fichier supplémentaire contenant les méthodes clés de ce code et qui s'appelle `Bomber.java`. Vous trouverez ces trois fichiers sur *moodle* et vous devez ajouter leur contenu à votre projet. Vous penserez également à ajouter les phrases qui manquent aux fichiers `strings.xml` français et anglais. Voici ces chaines :

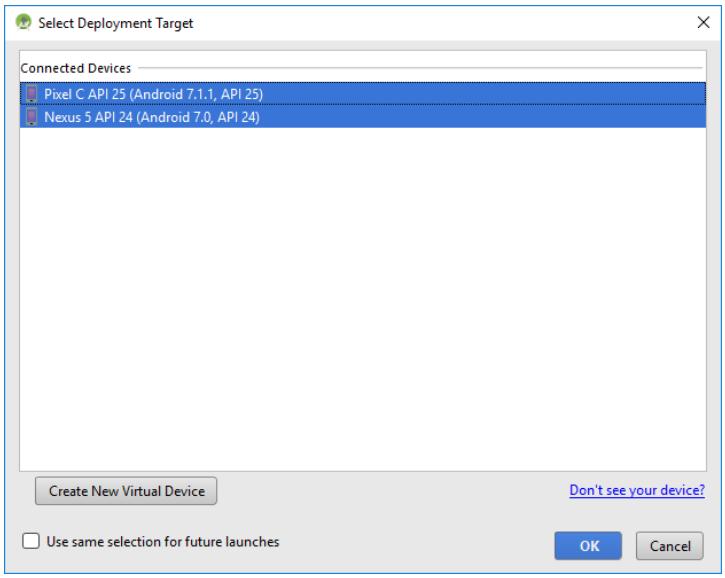
```
<string name="bombedText">Vous avez été bombé par </string>
<string name="target">cible</string>
<string name="source">attaquant</string>
<string name="resendBomb">Renvoyer la bombe</string>

<string name="nouserresponse">Pas de réponse de l\'utilisateur</string>
<string name="userbusy">L\'utilisateur est occupé</string>
<string name="cantbombme">Vous ne pouvez pas vous bomber vous même !</string>

<string name="bombexplosed">La bombe a explosé !</string>

<string name="win">Gagné !</string>
<string name="lost">Perdu !</string>
```

Vous devez également intercepter le clic sur le bouton « bomb it » de O*therUserActivity* en ajoutant le code suivant à la fin de sa méthode remplirLayout() :

```
bouton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        AndroBoumApp.getBomber().setBomb(p, new Bomber.BomberInterface() {
            @Override
            public void userBombed() {

            }

            @Override
            public void userBomber() {
                // on lance l'activité de contrôle de la bombe
                Intent intent = new Intent(context, BombActivity.class);
                context.startActivity(intent);
            }
        });
    }
});
```

Et enfin vous devez construire un nouvel objet de type Bomber en appelant AndroBoumApp.*buildBomber*(**this**); Le bon endroit pour le faire est dans UserActivity quand on est sûr que l'utilisateur est connecté (par exemple dans SetUser()).

Vous pouvez maintenant tester le code et si tout va bien, vous devriez pouvoir déposer une bombe chez un autre utilisateur (on ne peut pas s'envoyer de bombe à soi-même). Pour réaliser ce test, vous pouvez par exemple lancer deux émulateurs différents, déployer l'application sur les deux, et vous connecter sur chaque émulateur avec un compte différent. Une astuce : au moment du *Run* du projet, on peut sélectionner plusieurs appareils de déploiement en même temps comme sur la photo écran ci-après où on voit qu'on sélectionne à la fois le Pixel C et le Nexus 5. L'application est alors déployée et lancée sur les deux :



Si un autre utilisateur a correctement implémenté cette partie, vous devriez également pouvoir lui envoyer une bombe.

Si vous voulez plus de détails sur le fonctionnement de l'envoi et de la réception des bombes, vous pouvez consulter le contenu des fichiers *Bomber.java* et *BombActivity.java* pour accéder au code commenté.

5. Affichage du score

Complétez les activités *UserListActivity* et *OtherUserActivity* pour qu'elles affichent le score des utilisateurs. Vous devrez donc pour cela redéfinir leurs *layouts* pour faire une place au score et ajouter aux activités le code nécessaire à l'obtention de ce code.

Attention : la méthode setText qui permet de fixer le texte d'un TextView doit prendre pour paramètre une chaine de caractère et pas un nombre. Pour convertir un nombre en String, on pourra écrire :

```
String.valueOf(p.getScore());
```

X. Localiser l'utilisateur et afficher sa position

Objectif : Obtenir les coordonnées GPS de l'utilisateur et partager sa position avec les autres.

1. Obtenir la dernière position connue

L'API *Location* d'Android fournit un moyen simple d'obtenir la dernière position connue de l'utilisateur telle qu'enregistré par l'appareil. Cette API fait partie des *google play services*, il faut donc commencer par ajouter au fichier `build.gradle (Module : app)` la ligne suivante en remplaçant le numéro de version (ici 11.0.1) par celui que vous avez déjà utilisé pour les dépendances liées à *firebase*.

```
compile 'com.google.android.gms:play-services-location:11.0.1'
```

Il faut également insérer dans le fichier manifeste les permissions nécessaires à l'obtention de la localisation.

Insérez donc dans le fichier AndroidManifest.xml (juste après l'ouverture de la balise <manifest>) la ligne suivante :

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

Dans la partie `OnCreate` de l'activité *UserActivity*, obtenez une instance d'un *Fused Location Provider* en écrivant :

```
private FusedLocationProviderClient mFusedLocationClient;


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this);

    // suite du onCreate
}
```

Cette fois dans la méthode `setUser()`, obtenez la position de l'utilisateur en ajoutant (cela va générer une erreur, nous en parlons dans la section suivante) :

```
mFusedLocationClient.getLastLocation()
        .addOnSuccessListener(this, new OnSuccessListener<Location>() {
            @Override
            public void onSuccess(Location location) {
                // L'objet Location contient la position asuf s'il est null
                if (location != null) {
                    // faire quelquechose avec la position
                   System.out.println("Coordonnées GPS: Latitude="+
                             location.getLatitude()+" Longitude="+location.getLongitude());

                }
            }
        });
```

2. La gestion des permissions depuis Android 6

En principe, Android Studio doit générer une erreur lorsque vous insérez le code de la section précédente. En effet, avant Android 6, les permissions nécessaires aux applications pour fonctionner étaient demandées lors de l'installation de l'application. Depuis Android 6, elles le sont lorsque l'application en a besoin pour la première fois. Par conséquent, quand on écrit du code nécessitant une certaine permission, il faut vérifier que cette permission existe, et sinon la demander à l'utilisateur. Vous obtiendrez plus d'informations à l'adresse :

https://developer.android.com/training/permissions/requesting.html

Appliquez la méthode décrite en déplaçant le code ajouté à `setUser()` dans une méthode que vous appellerez `getLocation()` et dont voici le contenu :

```
private void getLocation() {
    if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_COARSE_LOCATION)
!= PackageManager.PERMISSION_GRANTED) {
        // on demande les permissions
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.ACCESS_COARSE_LOCATION},
                MY_PERMISSIONS_REQUEST_READ_CONTACTS);
        return;
```

```
    }
    mFusedLocationClient.getLastLocation()
            .addOnSuccessListener(this, new OnSuccessListener<Location>() {
                @Override
                public void onSuccess(Location location) {
                    // location contient la position, sauf si il est null.
                    if (location != null) {
                        Log.v("Androboum","Coordonnées GPS: Latitude="+
location.getLatitude()+" Longitude="+location.getLongitude());
                    }
                }
            });
}
```

et placez un appel à cette méthode à la fin de `setUser()`.

Ajoutez à *UserActivity* le *callback* qui réceptionne le fait que l'utilisateur a (ou n'a pas) accordé la permission nécessaire) :

```
@Override
public void onRequestPermissionsResult(int requestCode, String permissions[],
                                       int[] grantResults) {
    switch (requestCode) {
        case MY_PERMISSIONS_REQUEST_READ_CONTACTS: {
            // If request is cancelled, the result arrays are empty.
            if (grantResults.length > 0
                    && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                // on est autorisé, donc on rappelle getLocation()
                getLocation();

            } else {

                // on n'a pas l'autorisation donc on ne fait rien
            }
            return;
        }

    }
}
```

Vous pouvez à présent tester le code et vérifier que les coordonnées GPS apparaissent bien dans les logs.

3. Partager sa position

Pour faire cela, nous allons simplement stocker nos coordonnées dans la *firebase realtime database*. Commencez par ajouter deux propriétés à la casse Profil.java, en les dotant des *setters* et *getters* nécessaires :

```
// coordonnées GPS
private double latitude;
private double longitude;
```

Et ajoutez à la méthode `getLocation()` de *UserActivity* le code nécessaire pour remplir les deux propriétés de ce profil avec les coordonnées obtenues :

```
user.setLatitude(location.getLatitude());
user.setLongitude(location.getLongitude());
updateProfil(user);
```

Enfin ajoutez à la méthode `updateProfil()`, les deux lignes nécessaires pour stocker les deux coordonnées dans la base :

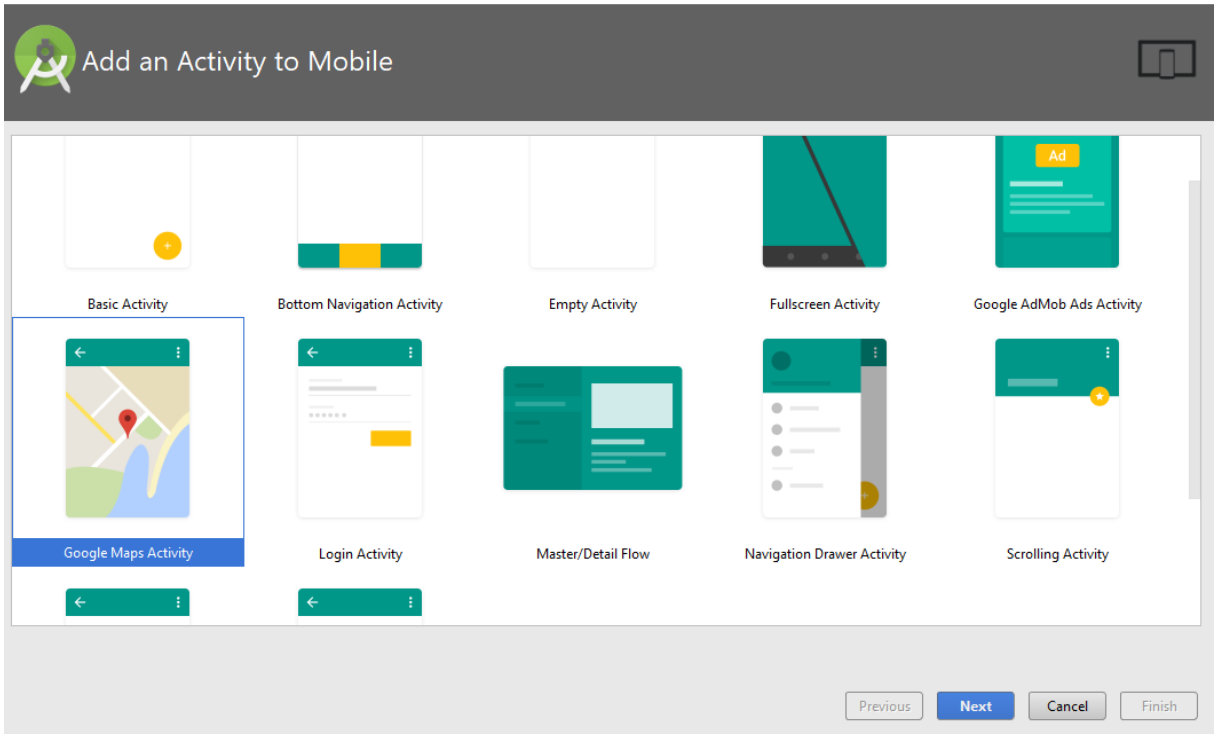
```
ref.child("latitude").setValue(user.getLatitude());
ref.child("longitude").setValue(user.getLongitude());
```

Testez que tout fonctionne en vérifiant via la console de *firebase*, que la base de données stocke désormais votre position.

4. Affichage de la position sur une google map

Nous allons faire en sorte d'afficher la position de chaque utilisateur sur une carte *google map*. Cette carte sera lancée par l'activité *UserListActivity* par un clic sur un des boutons de sa barre d'action.

Créez une nouvelle activité de type *Google Map Activity* que vous appellerez *MapsActivity*.

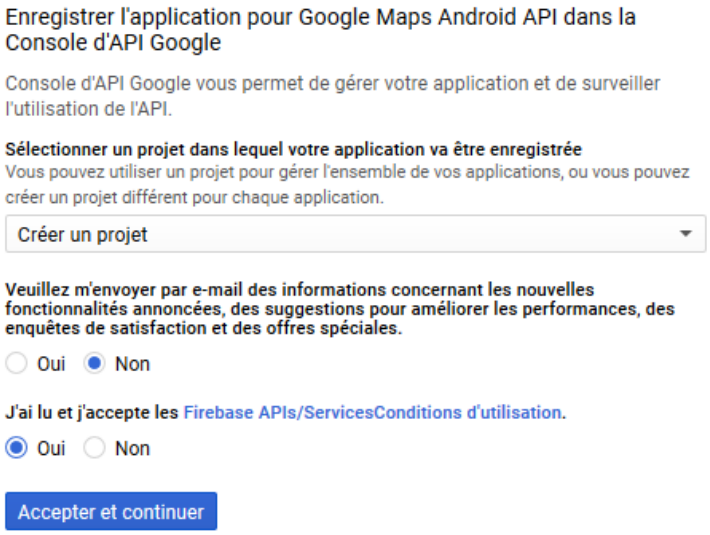


Cette action, en plus de créer l'activité, ajoute à vos dépendances la ligne suivante :

```
compile 'com.google.android.gms:play-services-maps:11.0.2'
```

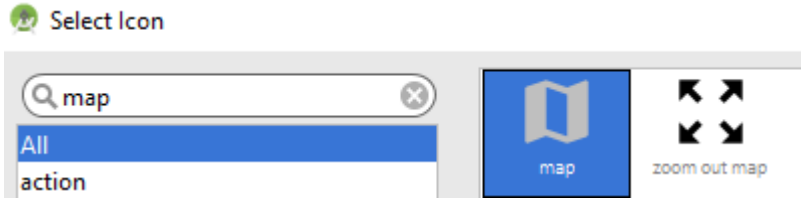
Vous ferez le nécessaire pour « aligner » le numéro de version choisi automatiquement par Android Studio avec la version de vos dépendances *firebase*.

Vous avez peut-être également remarqué qu'un fichier `google_maps_api.xml` a été créé dans le dossier *values* des ressources et qu'il est destiné à contenir votre clé d'accès à l'API *google maps*. Suivez les indications figurant dans les commentaires pour obtenir la clé.



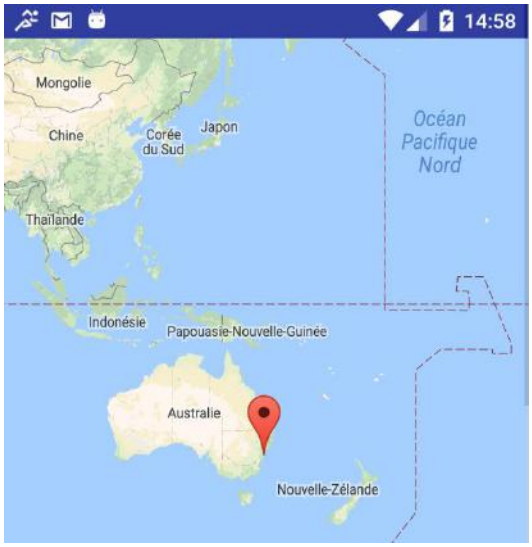
Suiviez la procédure jusqu'à son terme pour obtenir la clé et collez-là à l'endroit indiqué du fichier.

Ajoutez à présent une action supplémentaire dans la *ToolBar* de *UserListActivity*. Pour cela vous suivrez la procédure déjà écrite par exemple section IV.4 (page 21), le fichier à modifier étant `actions_user_list.xml`. Pour l'icône vous pouvez utiliser par exemple l'icône **map** du jeu d'icônes *material design*.



Ajoutez à *UserListActivity* le code nécessaire pour qu'un clic sur l'action *map* de sa barre d'action lance l'activité *MapsActivity*.

Vérifiez que cela fonctionne.



Le code par défaut de *MapsActivity* centre la carte sur Sidney. Nous allons changer ce comportement en faisant en sorte que la carte soit centrée sur notre position.

Pour cela ajoutez à *MapsActivity*, le code nécessaire pour :

- Repérez votre email. Pour cela vous procéderez comme dans *UserActivity*.
- Récupérer la liste de tous les utilisateurs. Vous procéderez comme dans *UserListActivity*.
- Repérez dans cette liste l'utilisateur portant votre email (vous le mettrez dans une variable globale que vous pouvez appeler par exemple `me`) et si la variable `mMap` n'est pas *null*, appelez la méthode `onMapReady(nMap)`.
- Modifiez la méthode `onMapReady()` pour qu'elle devienne (la variable `me` est de type Profil et désigne votre profil de connexion) :

```
@Override
public void onMapReady(GoogleMap googleMap) {
    mMap = googleMap;

    // ajoute un marker sur ma position et centre la carte dessus
    if (me != null) {
        LatLng maPosition = new LatLng(me.getLatitude(), me.getLongitude());
        mMap.addMarker(new MarkerOptions().position(maPosition).title(me.getEmail()));
        mMap.moveCamera(CameraUpdateFactory.newLatLng(maPosition));
    }

}
```

Remarque : Le défi que nous devons relever ici est de gérer correctement la bonne synchronisation des deux appels asynchrones, le premier étant celui qui télécharge la liste des utilisateurs, l'autre étant celui qui met en place la carte. Ce n'est que quand ces deux appels sont terminés que le marqueur peut être positionné. C'est ce qui explique les deux tests de non nullité. En effet la variable mMap contient une valeur nulle tant que la carte n'est pas prête. De façon similaire, la variable `me` contient une valeur nulle tant que votre profil n'est pas prêt. Ce n'est que quand ces deux variables sont toutes les deux non nulles que l'on peut spécifier le marqueur et le centrage de la carte.

Il nous reste à afficher les marqueurs correspondant à la position des autres utilisateurs. Nous les pointerons en vert. Voici le code à ajouter à la méthode onMapReady() pour réaliser cela :

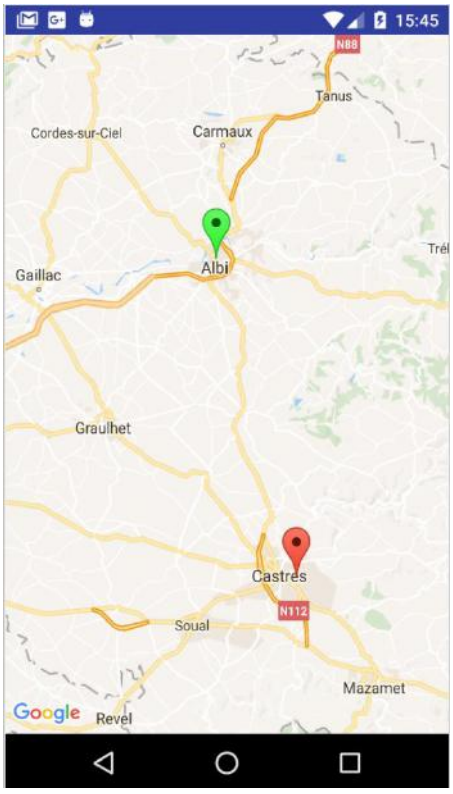
```
for (Profil user : userList) {
    if (user.getEmail() != me.getEmail() && user.getLatitude() != 0 &&
                  user.getLongitude() != 0) {
        LatLng position = new LatLng(user.getLatitude(), user.getLongitude());
        mMap.addMarker((new
MarkerOptions().position(position).title(user.getEmail()).icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_GREEN))));
    }
}
```

Autrement dit on itère sur chaque utilisateur et on ajoute pour chacun un marqueur vert, si sa position est connue.

Testez l'effet obtenue. Les marqueurs doivent apparaître.

Finalisons la carte, en la zoomant automatiquement autour des marqueurs (avec un effet d'animation en prime). On peut réaliser cela en calculant la zone délimitée par chaque marqueur et en zoomant par rapport au périmètre identifié. Voici le code complet de la méthode `onMapReady()` qui réalise cela :

```
@Override
public void onMapReady(GoogleMap googleMap) {
    mMap = googleMap;

    // le constructeur de périmètre
    LatLngBounds.Builder builder = new LatLngBounds.Builder();

    // ajoute un marker sur ma position et centre la carte dessus
    if (me != null) {
        LatLng maPosition = new LatLng(me.getLatitude(), me.getLongitude());
        // on ajoute ma position à la zone
        builder.include(maPosition);
        mMap.addMarker(new MarkerOptions().position(maPosition).title(me.getEmail()));
        mMap.moveCamera(CameraUpdateFactory.newLatLng(maPosition));


        // ajoute un marker pour chaque utilisateurs dont la position est repérée
        for (Profil user : userList) {
            if (user.getEmail() != me.getEmail() && user.getLatitude() != 0 &&
user.getLongitude() != 0) {
                LatLng position = new LatLng(user.getLatitude(), user.getLongitude());
                // on ajoute la position à la zone
                builder.include(position);
                mMap.addMarker((new
MarkerOptions().position(position).title(user.getEmail()).icon(BitmapDescriptorFactory.default
Marker(BitmapDescriptorFactory.HUE_GREEN))));
            }
        }


        // on construit le périmètre
        LatLngBounds bounds = builder.build();
        // on en calcule la hauteur et la largeur en pixels
        int width = getResources().getDisplayMetrics().widthPixels;
        int height = getResources().getDisplayMetrics().heightPixels;
        int padding = (int) (width * 0.10); // offset from edges of the map 10% of screen
        // on déplace la caméra sur la zone
        CameraUpdate cu = CameraUpdateFactory.newLatLngBounds(bounds, width, height, padding);
        mMap.animateCamera(cu);
    }
```



Enfin nous souhaitons qu'un clic sur le titre d'un marqueur lance l'activité qui permet de bomber l'utilisateur (c'est-à-dire *OtherUserActivity*).

Pour cela, ajoutez un *Tag* sur le marqueur des autres utilisateurs qui nous servira à mémoriser leur position dans la liste des utilisateurs. Voici comment modifier légèrement le marquage pour faire cela :

```
Marker marker = mMap.addMarker((new
MarkerOptions().position(position).title(user.getEmail()).icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_GREEN))));
// on ajoute un tag sur le marqueur
marker.setTag(pos);
```

Et on ajoute un écouteur de clic sur le titre des marqueurs lors de l'initialisation de la carte. Voici le nouveau début de la méthode `onMapReady()` :

```
@Override
public void onMapReady(GoogleMap googleMap) {
    mMap = googleMap;

    final Context c = this;

    // on positionne l'écouteur de clic sur les marqueurs
    mMap.setOnInfoWindowClickListener(new GoogleMap.OnInfoWindowClickListener() {
        @Override
        public void onInfoWindowClick(Marker marker) {
            // récupère le tag du marker qui correspond au numéro de l'utilisateur cliqué
            int pos = (int) marker.getTag();

            Intent intent = new Intent(c, OtherUserActivity.class);
            intent.putExtra("position", pos);
            startActivity(intent);
        }
    });
```

Testez que les clics sur les titres des marqueurs sont désormais pris en compte.

XI. Vérifier la proximité des autres utilisateurs

Attention ! Cette partie n'est testable que si vous disposez d'un vrai appareil Android. L'émulateur ne dispose pas des équipements matériels nécessaires pour détecter la proximité d'autres appareils.

Dans le chapitre précédent nous avons vu les utilisateurs pouvaient partager leur position et qu'il était donc possible de vérifier leur proximité géographique avec la nôtre. Android propose néanmoins une autre API qui permet de détecter les appareils à proximité en utilisant une combinaison du GPS, et des ondes Wifi et Bluetooth. Cette API est la Nearby Connexion Api, étudions son fonctionnement.

1. Mise en place

Ajoutez la dépendance suivante à votre projet en corrigeant si nécessaire le numéro de version :

```
compile 'com.google.android.gms:play-services-nearby:11.0.1'
```

Faites-en sorte que le fichier manifeste contienne les besoins en permissions suivants :

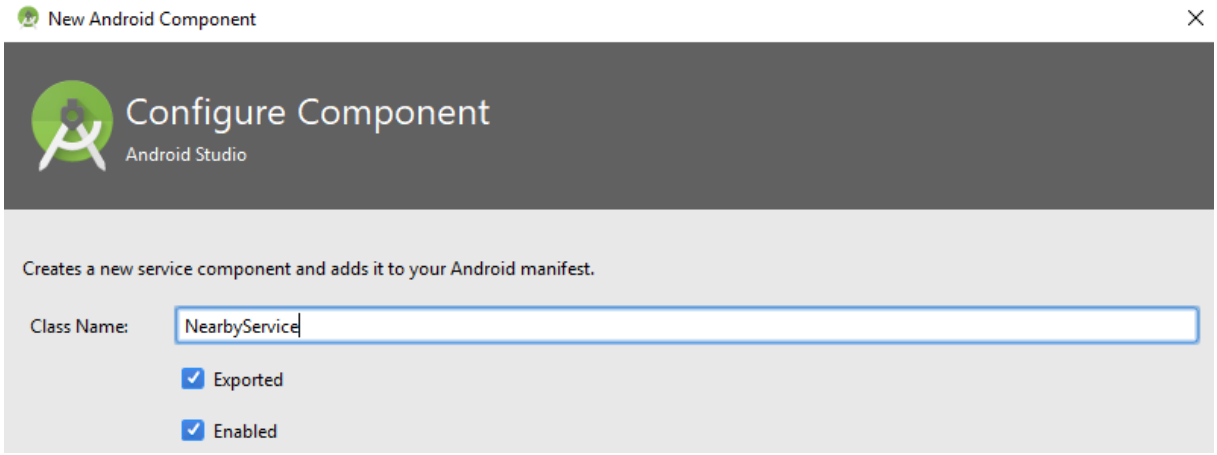
```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"
/>
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
```

2. Création d'un service

Pour que la détection des utilisateurs à proximité fonctionne pendant toute la durée de l'application, nous allons en faire un service. Un service se distingue d'une activité par le fait qu'il ne possède pas d'IHM et qu'il dispose d'un cycle de vie un peu différent. Pour en savoir plus sur les services vous pouvez vous référer à la documentation de référence :

https://developer.android.com/guide/components/services.html

Créez dans votre projet un nouveau Service en passant par les menus d'Android Studio que vous appellerez NearbyService.



Suivez le tutoriel décrit https://developers.google.com/nearby/connections/overview pour que le service implémente la détection des utilisateurs à proximité.

Attention le tutoriel prend pour référence un contexte d'activité, vous devez donc adapter en fonction de votre contexte qui est celui d'un service. Voici quelques indications pour cela :

- Vous connecterez le client à la *GoogleApiClient* dans la méthode OnCreate() du service.
- Vous démarrerez la découverte des autres (startDiscovery) et votre promotion (startAdvertising) dans la méthode onConnect() qui est appelé quand le GoogleApiClient est bien connecté.
- Pour que la détection fonctionne, vous devrez tous utiliser le même identifiant de service. Nous prendrons pour valeur :

  ```
  final String SERVICE_ID = "123";
  ```

- Quand vous serez en contact avec un autre utilisateur, vous lui enverrez votre email en utilisant les *payload*.

- Quand vous recevrez un *payload* en provenance d'un autre utilisateur, vous ferez *popper* une notification qui affichera le message « Nouvel utilisateur à proximité » avec l'email de l'utilisateur en question.
- Comme bonus, vous ferez en sorte que la notification donne à chaque fois la liste complète des utilisateurs à proximité, ce qui suppose que vous la mémorisiez au fur et à mesure que vous êtes mis en contact avec les autres.