

UNIVERSITÉ DE MONTPELLIER  
MASTER 1 - IMAGINE

---

## Meeting Car

Application mobile pour annonces de ventes et locations de voitures

---

RAPPORT DE PROJET : PROGRAMMATION MOBILE  
PROJET INFORMATIQUE — HAI811I

**Étudiants :**

M. Florentin DENIS  
M. Khélian LARVET

**Année :** 2021 - 2022



# Table des matières

<b>Introduction</b>	<b>2</b>
<b>1 Développement de l'application mobile</b>	<b>4</b>
1.1 Choix de développement . . . . .	4
1.2 Fragments liés aux annonces . . . . .	6
1.2.1 "HomeFragment" . . . . .	6
1.2.2 "AnnounceFragment" . . . . .	9
1.2.3 "StatistiqueFragment" . . . . .	10
1.2.4 "CreateAnnonceFragment" . . . . .	12
1.3 Fragments liés aux profils . . . . .	13
1.3.1 "LoginFragment" . . . . .	13
1.3.2 "ProfileFragment" . . . . .	15
1.3.3 "FollowFragment", "HistoryFragment", "MyAnnouncesFragment" . . . . .	17
1.4 Fragments liés aux messageries . . . . .	18
1.4.1 "MailFragment" . . . . .	18
1.4.2 "DiscussionFragment" . . . . .	20
<b>2 Développement côté serveur</b>	<b>22</b>
2.1 Structure de la base de donnée . . . . .	22
2.2 Serveur web . . . . .	23
2.2.1 Configuration . . . . .	23
2.2.2 Structure serveur . . . . .	24
2.2.3 Connexion utilisateur . . . . .	24
<b>3 Déploiement du projet</b>	<b>27</b>
3.1 Mise en place de l'application . . . . .	27
3.2 Mise en place de la base de données . . . . .	27
3.3 Mise en place du serveur . . . . .	27
<b>Annexes</b>	<b>29</b>
<b>Bibliographie</b>	<b>32</b>

# Introduction

Dans le cadre du module "Programmation mobile" du second semestre de M1 IMAGINE, nous avons développé l'application **Meeting Car** dont le dépôt est disponible à l'adresse suivante :

<https://github.com/Flare00/MeetingCar>

Cette application vise à offrir des outils selon deux types d'utilisateurs : les "*annonceurs*" qui pourront publier des annonces sur les voitures qu'ils vendent ou louent ; les "*acheteurs*" qui auront accès aux différentes annonces publiées et pourront contacter les "*annonceurs*" pour trouver un accord.

## Motivations du projet

Notre première motivation sur ce projet a été la possibilité de créer une véritable application mobile avec plusieurs fonctionnalités à implémenter. Parmi ces fonctionnalités, nous pouvons distinguer l'implémentation d'un système de navigation, ou encore l'utilisation d'un recycleur de vue pour optimiser l'affichage d'un grand nombre de données.

Nous avons également été intéressés par la possibilité de faire communiquer une application Android avec un serveur web.

Ce projet nous semblait donc parfait pour approfondir nos compétences autant du côté client que du côté serveur avec la gestion des données de plusieurs utilisateurs.

## Objectifs du projet et cahier des charges

L'objectif de ce projet a donc été de créer une plateforme permettant à des utilisateurs de vendre ou de louer leurs voitures. Pour mener à bien ce projet, plusieurs objectifs nous ont été donnés :

- Le premier objectif indiquait la mise en place d'un système d'inscription et de connexion pour des utilisateurs. Plusieurs informations sont ainsi retenues et permettent de former un profil qui sera modifiable si l'utilisateur le souhaite.
- Le second objectif a été de créer l'ensemble des fonctionnalités permettant la création des annonces. Un utilisateur connecté a donc le pouvoir de créer des annonces et de les modifier si ce sont les siennes. Il peut également utiliser le système de recherche pour trouver une annonce particulière.
- Puis, il nous a été demandé d'ajouter des fonctionnalités permettant le suivi de certaines annonces ou encore un historique d'achat pour tous les utilisateurs. Pour les comptes "*Pro+*", des informations supplémentaires doivent être affichées comme par exemple des statistiques sur les annonces créées.
- Enfin un système de messagerie était nécessaire pour que des "*Profils acheteurs*" puissent contacter des "*Profils annonceurs*" sur une annonce particulière. Cette fonctionnalité devait également être disponible pour des comptes non connectés à l'aide d'un email personnel.

## Technologies utilisées

Le choix a été fait en accord avec nos compétences respectives et dans l'objectif de simplifier au maximum les tâches à réaliser :



### Android

Fondé sur le noyau Linux, c'est une pile de logiciels permettant la mise en oeuvre d'application mobile.



### Android Plot :

Bibliothèque permettant la représentation graphique des données pour des applications Android. Elle nous sert à afficher plusieurs graphiques sur diverses données pour les utilisateurs professionnels.



### Node.js :

Plateforme logicielle JavaScript permettant la création d'application événementielle, tel que des serveurs web permettant une forte montée en charge.



### MySQL :

C'est un serveur de bases de données relationnelles distribué sous licence libre GNU. Nous pouvons le voir comme un lieu de stockage des données qui seront ensuite récupérées via des requêtes SQL.

# Chapitre 1

## Développement de l'application mobile

### 1.1 Choix de développement

Pour ce projet, nous avons fait le choix de **créer une seule et unique activité** nommée `MainActivity` qui pourra utiliser un ensemble de fragments. Cette décision nous permet de conserver notre `NavigationDrawer` entre chaque nouvel appel de page.

Nous avons également fait le choix **d'utiliser la liaison de vue** qui est une fonctionnalité permettant d'écrire plus facilement du code pour interagir avec les vues.

Une fois la liaison de vue activée, l'application génère une classe de liaison pour chaque fichier de mise en page XML présent. Une instance d'une classe de liaison contient des références directes à toutes les vues qui ont un ID dans la mise en page correspondante.

```
1 private ActivityMainBinding binding;
2 binding = ActivityMainBinding.inflate(LayoutInflater());
3 setContentView(binding.getRoot());
```

Nous avons donc mis en place la structure suivante pour les "layouts" :

- `content_main` : Ce sera lui qui contiendra le fragment affiché dans le `MainActivity`. Il possède un système de navigation correspondant à un `navGraph` (avec notre `mobile_navigation.xml`) indiquant à l'application comment accéder aux autres fragments.
- `app_bar_main` : Inclu `content_main` et ajoute la barre d'outil du haut permettant d'indiquer le label du fragment affiché.
- `activity_main` : Inclu `app_bar_main` et ajoute la `NavigationView` ce qui permettra l'affichage d'un menu déroulant de gauche à droite. Ceci favorisera la fluidité des déplacements dans notre application.

Ainsi la classe `MainActivity` gère plusieurs aspects de notre application. Dans un premier temps et comme énoncé précédemment, nous effectuons la liaison de vue. Puis nous initialisons toutes les navigations vers nos différents fragments pour le `NavigationDrawer` :

```
1 // ATTRIBUTES
2 private AppBarConfiguration mAppBarConfiguration;
3 private ActivityMainBinding binding;
4 private NavigationView navigationView;
5
6 // NAVIGATION DRAWER
7 setSupportActionBar(binding.appBarMain.toolbar);
8 DrawerLayout drawer = binding.drawerLayout;
9 navigationView = binding.navView;
```

```

1 // LOAD NAVIGATION
2 mAppBarConfiguration = new AppBarConfiguration.Builder(
3     R.id.nav_home,
4     R.id.nav_profile,
5     R.id.nav_mail,
6     R.id.nav_follow,
7     R.id.nav_announces,
8     R.id.nav_history,
9     R.id.nav_login,
10    R.id.nav_annonce,
11    R.id.nav_create_announce)
12    .setOpenableLayout(drawer)
13    .build();
14
15 NavController navController = Navigation.findNavController(this,
16     R.id.nav_host_fragment_content_main);
16 NavigationUI.setupActionBarWithNavController(this, navController, mAppBarConfiguration);
17 NavigationUI.setupWithNavController(navigationView, navController);

```

Enfin, elle gère également des aspects liés à la connexion de l'utilisateur sur notre application, notamment avec la méthode `askIsLogin()` permettant la persistance des données :

```

1 public void askIsLogin(boolean isLoggedIn) {
2     if(!isLoggedIn){
3         Sharedpreferences sp = this.getSharedPreferences("auto_connect",
4             Context.MODE_PRIVATE);
5         String email = sp.getString("email", null);
6         String pass = sp.getString("password", null);
7         String date = sp.getString("date", null);
8
8         //Toast.makeText(getApplicationContext(),"EMAIL : " + email+ "\n PASS : " + pass
9         // + "\n DATE : " + date, Toast.LENGTH_SHORT ).show();
10        if(email != null && pass != null){
11            CommunicationWebservice.getINSTANCE().connect(email, pass, this, true);
12        }
13    }
}

```

Nous allons maintenant voir plus en détail l'ensemble des fragments accessibles par notre `MainActivity`.

## 1.2 Fragments liés aux annonces

### 1.2.1 "HomeFragment"

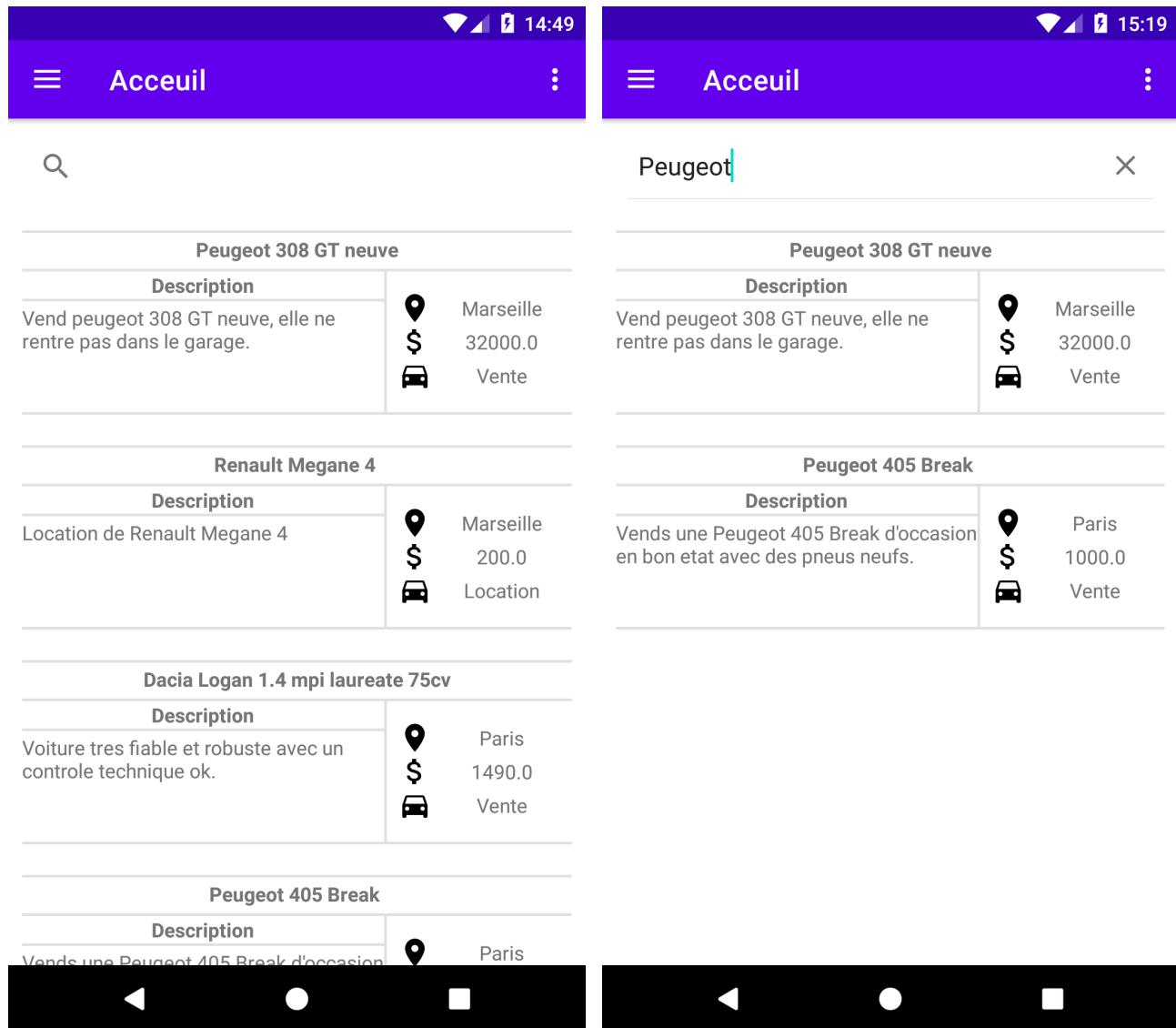
HomeFragment désigne notre fragment central, c'est le premier à être appelé par le `MainActivity`. C'est ici qu'apparaîtront toutes les annonces ainsi qu'une barre de recherche pour trouver une annonce particulière si elle existe.

Dans un premier temps nous devons récupérer la liste des annonces sur le serveur, pour cela nous utilisons la fonction `CommunicationWebService.getINSTANCE().getAnnonceListe()` qui vient demander au serveur les annonces. Cette requête réseau s'effectue à l'aide d'un Thread et nous pouvons voir qu'elle fait une requête GET à l'URL "`annonce/page/`" qui récupère une page de 20 annonces sur la base de donnée. Une fois les données récupérées, La fonction les transforme dans la liste d'objet désiré, ici une `xArrayList<Annonce>` et la renvoie dans le callback.

```

1 public void getAnnonceListe(int page, IListAnnonceLoaderHandler callback) {
2     if (page >= 0) {
3         new Thread(() -> {
4             ArrayList<Annonce> liste = new ArrayList<>();
5             try {
6                 HttpsURLConnection connection = (HttpsURLConnection) new
7                     URL(Config.BASE_URL + "annonce/page/" + page).openConnection();
8                 connection.setConnectTimeout(2500);
9                 connection.setRequestMethod("GET");
10                try (BufferedReader in = new BufferedReader(new
11                    InputStreamReader(connection.getInputStream(), "UTF-8"))) {
12                    StringBuilder sb = new StringBuilder();
13                    String line = "";
14                    while ((line = in.readLine()) != null) {
15                        sb.append(line);
16                    }
17                    JSONObject json = new JSONObject(sb.toString().trim());
18                    JSONArray array = json.optJSONArray("result");
19                    if (array != null) {
20                        for (int i = 0, max = array.length(); i < max; i++) {
21                            liste.add(Annonce.fromJsonObject(array.getJSONObject(i)));
22                        }
23                    }
24                } catch (JSONException e) {
25                    e.printStackTrace();
26                }
27            } catch (MalformedURLException e) {
28                e.printStackTrace();
29            } catch (IOException e) {
30                e.printStackTrace();
31            }
32            callback.onListAnnonceLoad(liste);
33        }).start();
34    }
}

```



Liste des annonces

Recherches dans les annonces

Pour ce faire nous avons développé une classe nommée **SpecialAdapter**, nous permettant d'afficher différents types de données avec le type **IViewModel**. Ainsi nous avons trois types de données :

- **AdvertViewModel** : Une annonce
- **MailViewModel** : Une discussion
- **MessageViewModel** : Un message

Puis nous avons utilisé un **RecyclerView** pour afficher l'ensemble de ces données en ajoutant une méthode **addOnItemTouchListener()** pour que chaque élément soit cliquable.

```

1  protected void touchListener() {
2      // GESTURE
3      GestureDetector gd = new GestureDetector(this.getActivity(), new
4          → GestureDetector.SimpleOnGestureListener() {
5              @Override
6                  public boolean onSingleTapUp(MotionEvent e) {
7                      return true;
8                  }
9              });
10
11     Fragment self = this;
12
13     // LISTENER RECYCLER
14     recycler.addOnItemTouchListener(new RecyclerView.SimpleOnItemTouchListener() {
15         @Override
16             public boolean onInterceptTouchEvent(@NonNull RecyclerView rv, @NonNull
17                 → MotionEvent e) {
18                 View child = rv.findChildViewUnder(e.getX(), e.getY());
19                 boolean touch = gd.onTouchEvent(e);
20                 if (child != null && touch) {
21                     int pos = rv.getChildAdapterPosition(child);
22                     AdvertViewModel avm = (AdvertViewModel) adapter.getData().get(pos);
23
24                     Bundle b = new Bundle();
25                     b.putInt("idAnnonce", avm.getId());
26                     NavController navController = NavHostFragment.findNavController(self);
27                     navController.popBackStack();
28                     navController.navigate(R.id.nav_annonce, b);
29                     return true;
30                 }
31             return false;
32         }
33     });
34 }

```

Voici les différentes opérations du code ci-dessus : Dans un premier temps nous créons un `GestureDetector` permettant de ne prendre en compte que les appuis simples sur une annonce, et pour ne pas surcharger l'élément permettant le défilement de la page.

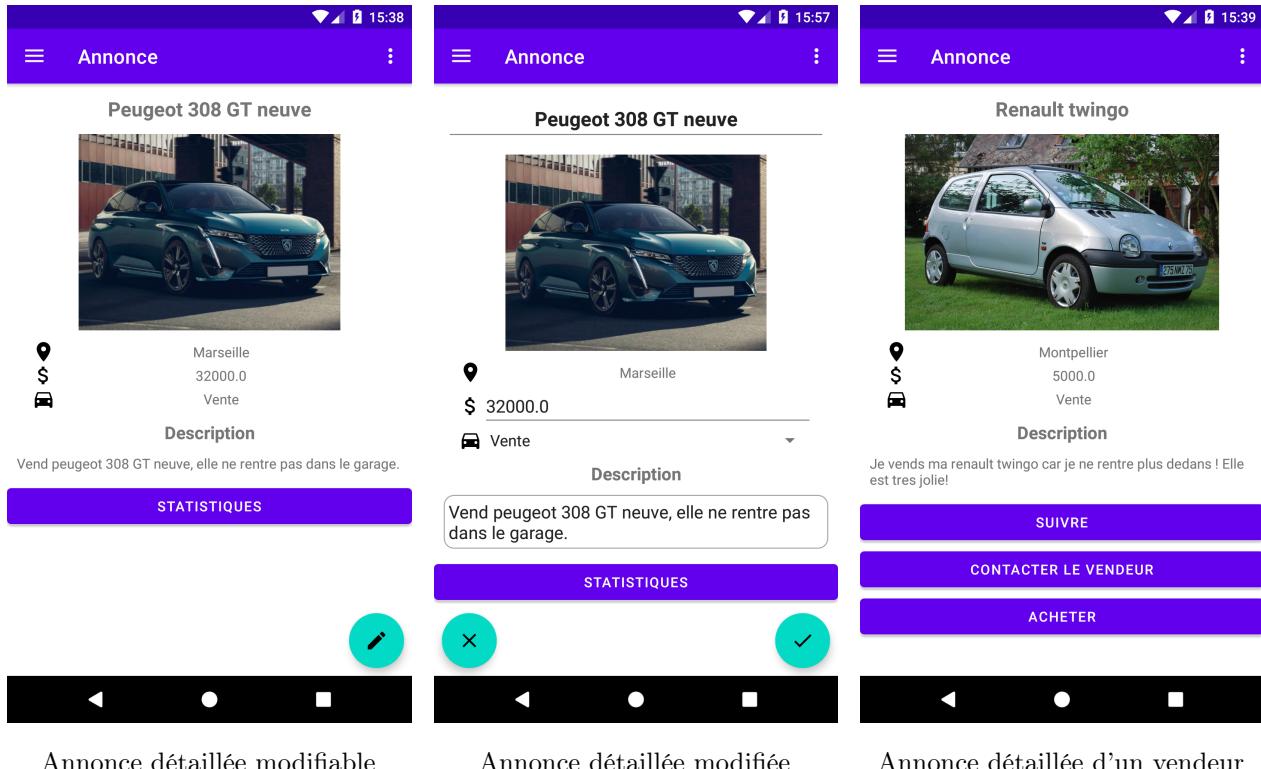
Puis nous avons ajouté la méthode `OnItemTouchListener()`, permettant de tester s'il s'agit bien d'un simple appui à l'aide du `GestureDetector` ci-dessus, avant de venir tester s'il y a bien un élément à l'endroit cliqué.

Ensuite nous récupérons la position dans la liste de l'élément touché, afin de récupérer ses données dans l'adaptateur.

Pour finir nous créons un `Bundle` permettant de passer les données entre les fragments tout en appliquant une navigation vers le fragment affichant les détails d'une annonce : `AnnounceFragment`.

## 1.2.2 "AnnounceFragment"

Comme énoncé précédemment, ce fragment permet l'affichage des détails d'une annonce en utilisant le layout `fragment_announce`.



Annonce détaillée modifiable

Annonce détaillée modifiée

Annonce détaillée d'un vendeur

Ainsi, plusieurs informations sont observables telles que :

- Le titre de l'annonce ;
- Plusieurs images avec un `ViewPager` ;
- La localisation ;
- Le prix ;
- Le type d'annonce (Vente ou Location) ;
- Une description ;

Si l'utilisateur est anonyme, il n'y aura qu'un seul choix : celui de contacter le vendeur. À partir du moment où l'utilisateur est connecté et que la publication ne lui appartient pas, il peut en plus de contacter le vendeur, ajouter l'annonce à sa liste de suivi ou alors acheter directement la voiture.

La publication est modifiable par un utilisateur si et seulement si cet utilisateur l'a créée. Cette modification peut s'opérer en utilisant le bouton d'édition en bas à droite, ce qui permet ainsi de modifier le titre de l'annonce, le prix, le type, etc...

De plus si le compte de l'utilisateur possède l'état "Pro+", un nouveau fragment peut être observé : le `StatistiqueFragment`.

### 1.2.3 "StatistiqueFragment"

Ce fragment est donc réservé aux comptes avec l'état "Pro+", et permet l'affichage de statistiques détaillées sur le nombre de visites des annonces créées. Pour ce faire nous avons dans un premier temps mis en place dans layout l'affichage d'un graphique à deux axes (**XYPlot**).

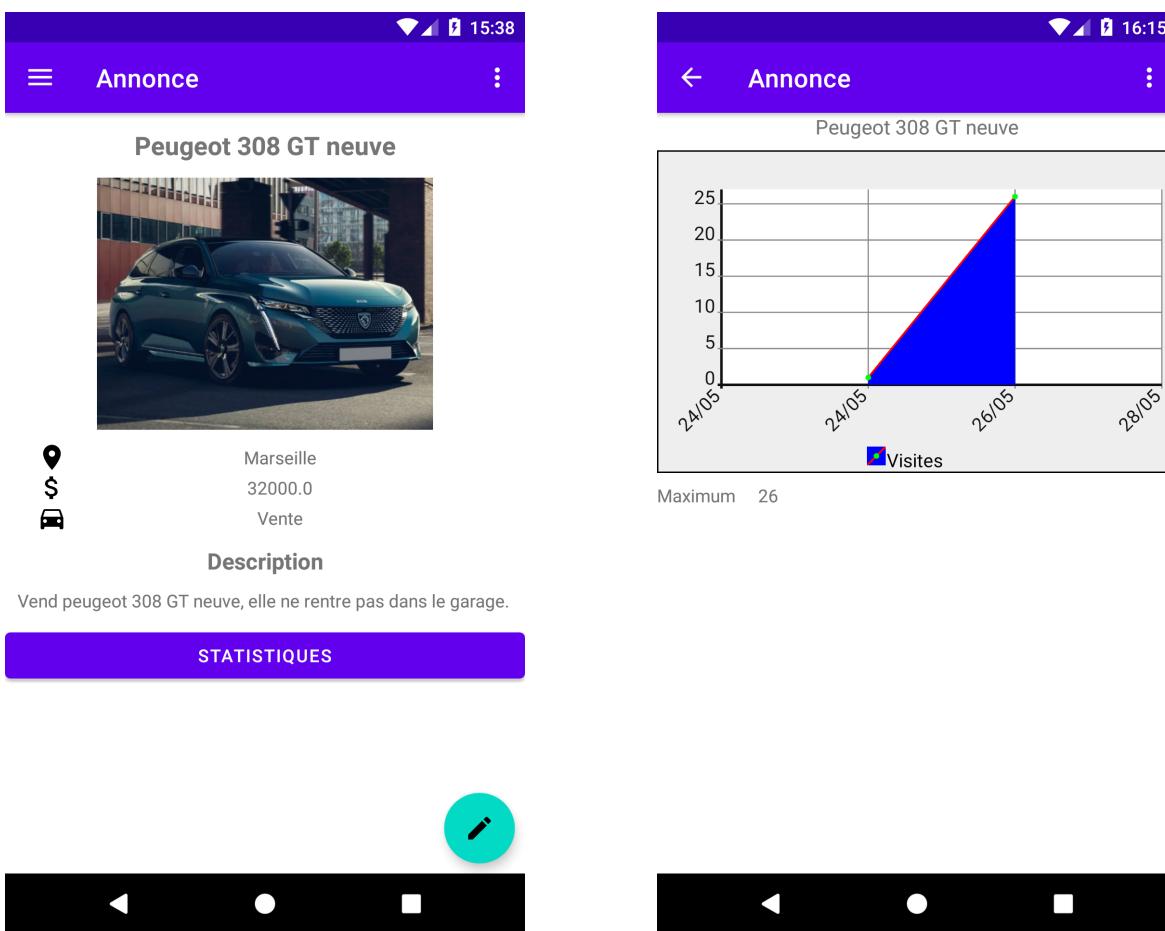
Puis, nous lui fournissons des données à l'aide du serveur. Les données affichées sur l'axe X ont pour unité la date au format JJ/MM, et celles affichées sur l'axe Y, le nombre de visites sur l'annonce.

Le **XYPlot** prend en données deux tableaux d'objet **Number**, ici nous n'avons besoin que d'entier donc nous créons des tableaux d'**Integer**, qui hérite de **Number**.

Pour la récupération des données, nous stockons les différentes valeurs dans un tableau associatif trié, permettant de compter pour chaque date le nombre de visite.

Après cela, nous pouvons simplement transmettre sur l'axe X les clés et sur l'axe Y le nombre d'occurrences de chaque date. Nous configurons le graphique selon les données reçues. La "portée" (Y) affiche entre zéro et le maximum d'occurrence pour une date plus un jour. Le "domaine" (X) affiche un jour avant le premier jour, et un jour après le dernier jour trouvé.

Avec les données nous pouvons créer une série nommée "Visites", que nous donnons à l'élément **XYPlot** pour qu'il le dessine.



```

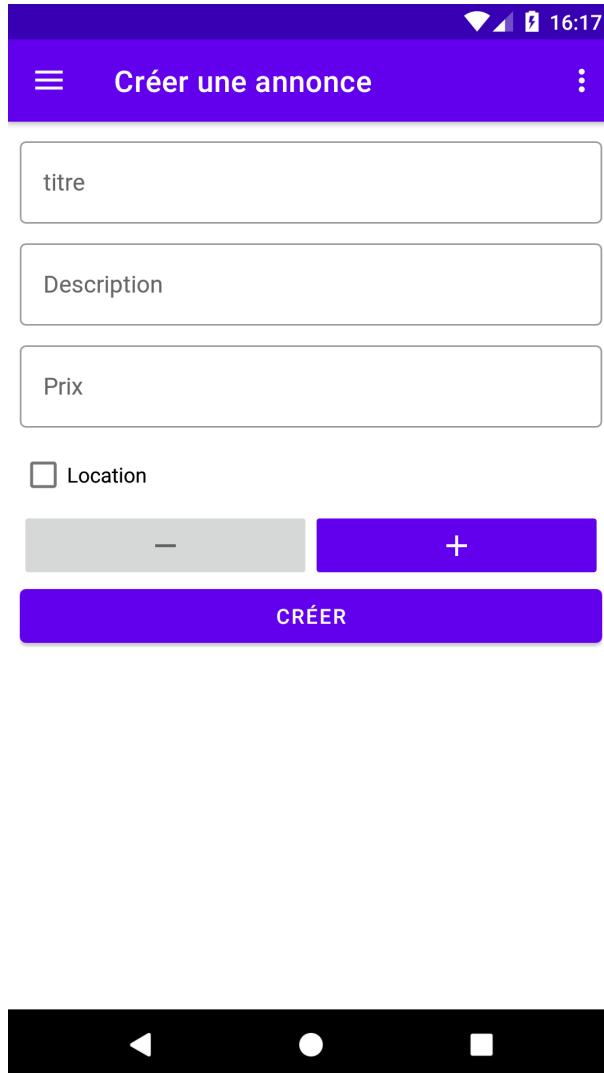
1 CommunicationWebservice.getINSTANCE().getVisites(a, visites -> {
2     SortedMap<String, Integer> values = new TreeMap<>();
3     for(Visite v : visites){
4         String dateString = v.getHorodatage().substring(0,8);
5         if(values.containsKey(dateString)){
6             int val = values.get(dateString) + 1;
7             values.put(dateString, val);
8             [...]
9         } else {
10             values.put(dateString, 1);
11         }
12     }
13     ArrayList<Map.Entry<String, Integer>> entry = new ArrayList<>(values.entrySet());
14     int size = entry.size();
15     Integer[] xVals = new Integer[size];
16     Integer[] yVals = new Integer[size];
17
18     for(int i = 0; i < size;i++ ){
19         xVals[i] = Integer.parseInt(entry.get(i).getKey());
20         yVals[i] = entry.get(i).getValue();
21     }
22     int nbDate = (xVals[size-1] + 1) - (xVals[0] - 1);
23     binding.plotStats.setRangeBoundaries(0, max+1, BoundaryMode.FIXED);
24     binding.plotStats.setRangeStep(StepMode.INCREMENT_BY_FIT, (int)(max/5.0));
25     binding.plotStats.setDomainBoundaries(xVals[0] - 1, xVals[size-1] + 1,
26         BoundaryMode.FIXED);
27     binding.plotStats.setDomainStep(StepMode.INCREMENT_BY_FIT, (int)(nbDate/5.0) +1);
28     [...]
29     XYSeries serie = new SimpleXYSeries(Arrays.asList(xVals), Arrays.asList(yVals),
30         "Visites");
31     getActivity().runOnUiThread(() -> {
32         binding.plotStats.clear();
33         binding.plotStats.addSeries(serie, format);
34         binding.plotStats.redraw();
35         [...]
36     });
37 });
38 });

```

### 1.2.4 "CreateAnnonceFragment"

La création d'une annonce se fait uniquement si l'utilisateur est connecté. Cette création implique diverses informations, notamment un titre, une description, un prix et s'il s'agit d'une location ou d'une vente.

Un ensemble de photos peut être ajouté ou retiré pour illustrer la publication à l'aide des boutons "+" et "-".



Création d'une annonce

Pour ajouter des images, nous utilisons un nouvel Intent avec l'action ACTION\_PICK, ce qui permet à l'utilisateur de chercher des images sur son propre téléphone.

```

1 private void openImageSelector() {
2     Intent pickIntent = new Intent(Intent.ACTION_PICK);
3     pickIntent.setType("image/*");
4     photoChooser.launch(pickIntent);
5 }
```

## 1.3 Fragments liés aux profils

### 1.3.1 "LoginFragment"

Le LoginFragment est nécessaire pour connecter un utilisateur à un compte voire pour créer un nouveau compte.

La connexion s'effectue via la méthode `LoginAction()` qui pour un email et un mot de passe donnés, vérifie si les informations entrées ne sont pas erronées ou vides. Une fois vérifiées, elles sont envoyées vers le serveur pour autoriser la connexion si elles sont correctes.

```

1 public void loginAction(View view) {
2
3     TextInputLayout emailTIL = binding.tfEmailLogin;
4     TextInputLayout passwordTIL = binding.tfPasswordLogin;
5     String email = emailTIL.getEditText().getText().toString().trim();
6     String password = passwordTIL.getEditText().getText().toString().trim();
7     boolean loginPossible = true;
8     if (email.length() <= 0) {
9         emailTIL.setError(getResources().getString(R.string.error_empty));
10        emailTIL.setErrorEnabled(true);
11        loginPossible = false;
12    }
13    if (password.length() <= 0) {
14        passwordTIL.setError(getResources().getString(R.string.error_empty));
15        passwordTIL.setErrorEnabled(true);
16        loginPossible = false;
17    }
18    if (loginPossible) {
19        CommunicationWebservice.getINSTANCE().connect(email, password, this, false);
20    }
21 }
```

L'inscription s'opère via la méthode `registerAction()` qui demande un ensemble d'informations dont certaines optionnelles. Une fois toutes les informations rentrées, nous les récupérons en leur appliquant la méthode `trim()`, puis nous vérifions qu'aucun champ ne soit vide. Si c'est le cas un nouveau Client est créé.

```

1 public void registerAction(View view) {
2     TextInputLayout emailTIL = binding.tfEmailLogin;
3     [...]
4     String email = emailTIL.getEditText().getText().toString().trim();
5     [...]
6     if (passwordSecond.length() <= 0) {
7         passwordSecondTIL.setError(getResources().getString(R.string.error_empty));
8         passwordSecondTIL.setErrorEnabled(true);
9         registerPossible = false;
10    } else if (!passwordSecond.equals(password)) { [...] }
11    [...]
12    Client c = new Client(-1, surname, name, email, phone, birthday, address);
13    CommunicationWebservice.getINSTANCE().inscription(c, password,
14        ↳ binding.cbProfessionalRegister.isChecked(), imageURI,
15        ↳ this.getActivity().getContentResolver(), this);
16 }
```

The screenshot shows the connection screen of a mobile application. At the top, there is a header bar with the title "Connexion". Below the header, there are two input fields: one for "E-mail" containing "utilisateur@utilisateur.fr" and one for "Mot de passe" with an eye icon for password visibility. A large blue button labeled "SE CONNECTER" is centered below the password field. At the bottom left, there is a link "S'INSCRIRE". The bottom of the screen features a black navigation bar with three white icons: a triangle pointing left, a circle, and a square.

Connexion

The screenshot shows the registration screen of a mobile application. At the top, there is a header bar with the title "Connexion". Below the header, there are several input fields: "E-mail\*" (containing "utilisateur@utilisateur.fr"), "Mot de passe\*" (with an eye icon), "Confirmez votre mot de passe\*" (with an eye icon), "Nom\*" (empty), "Prénom" (empty), "Numéro de Téléphone" (empty), "Date de naissance" (with placeholder "jj/mm/aaaa" and a calendar icon), "Adresse" (empty), and a placeholder "IMAGE DE PROFILE". The bottom of the screen features a black navigation bar with three white icons: a triangle pointing left, a circle, and a square.

Inscription

### 1.3.2 "ProfileFragment"

Ce fragment permet l'affichage des informations entrées lors de l'inscription. Dans le respect de la vie privée, nous n'affichons pas la date de naissance ou la position. Cet affichage est effectué par la méthode `OnClientLoad()` :

```

1  @Override
2  public void onClientLoad(Client c, boolean self) {
3      getActivity().runOnUiThread(() -> {
4          if (c.getPrenom() != null) {
5              binding.profileTvName.setText(c.getNom() + " " + c.getPrenom());
6          } else {
7              binding.profileTvName.setText(c.getNom());
8          }
9          binding.profileTvEmail.setText(c.getEmail());
10         if (c.getTelephone().equalsIgnoreCase("null")) {
11             binding.profileTvPhone.setText("");
12         } else {
13             binding.profileTvPhone.setText(c.getTelephone());
14         }
15     }
16     if (c.getClass() == Professionnel.class) {
17         binding.profileTvPro.setVisibility(View.VISIBLE);
18     }
19     if (self) {
20         binding.profileFabEdit.setVisibility(View.VISIBLE);
21         isSelf = true;
22     }
23     if(c.getImage() != null && c.getImage().getId() >= 0){
24         CommunicationWebservice.getINSTANCE().getImage( c.getImage().getId(), img ->
25             -> {
26                 c.setImage(img);
27                 getActivity().runOnUiThread(() -> {
28                     binding.profileImage.setImageDrawable(img.getDrawable());
29                 });
30             });
31     }
32 }

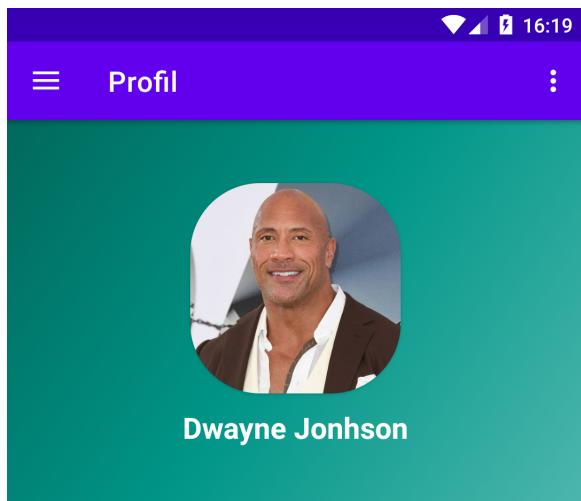
```

Ainsi nous récupérons toutes les informations nécessaires si elles existent sur notre serveur et nous les affichons à leurs emplacements prévus. L'utilisateur peut également modifier ses informations à l'aide du bouton d'édition positionné en bas à droite. À la fin de l'édition, si toutes les informations sont validées, nous utilisons la méthode `updateClient()` avec les nouvelles informations.

```

1  public void checkEditProfile() {
2      // Check Data
3      [...]
4
5      onClientLoad(Metier.getINSTANCE().getUtilisateur(), true);
6      cancelEditProfile();
7      CommunicationWebservice.getINSTANCE().updateClient(
8          -> Metier.getINSTANCE().getUtilisateur(), null, null);
9 }

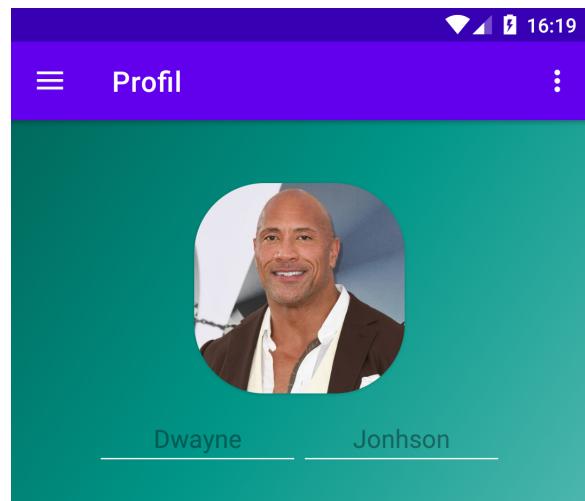
```



utilisateur@utilisateur.fr

123456789

Dwayne Jonhson



utilisateur@utilisateur.fr

123456789

20220526

Montpellier



Profil

Édition du profil

### 1.3.3 "FollowFragment", "HistoryFragment", "MyAnnouncesFragment"

Ces trois fragments héritent de `HomeFragment` et utilisent `SpecialAdapter` pour afficher des annonces. Une différenciation s'effectue sur la méthode utilisée pour récupérer les données sur le serveur :

- `FollowFragment` utilise la fonction `getAnnoncesFollow()`
- `HistoryFragment` utilise la fonction `getAnnoncesAcheteur()`
- `MyAnnouncesFragment` utilise la fonction `getAnnoncesVendeur()`

Nous pouvons voir ci-dessous la fonction surchargée de `queryData` dans `FollowFragment`. Dans un premier temps elle demande les annonces que l'utilisateur suit, puis dans un callback récupère la liste des annonces. Pour chacune des annonces elle l'ajoute au recycleur de vue, puis les affiche.

```

1 if(idClient >= 0){
2     CommunicationWebservice.getINSTANCE().getAnnoncesFollow(idClient, 0, liste -> {
3         for(Annonce a : liste){
4             super.data.add(new AdvertViewModel(a.getId(), a.getTitle(), a.getDesc(),
5                 ↳ a.getVendeur().getAdresse(), ""+a.getPrix(), (a.isLocation() ?
6                     ↳ AdvertViewModel.TYPE.RENT : AdvertViewModel.TYPE.SELL)));
7         }
8         getActivity().runOnUiThread(() -> {
9             super.adapter.setDataAffichage();
10            super.adapter.notifyDataSetChanged();
11        });
12    });
13 }
```

## 1.4 Fragments liés aux messageries

### 1.4.1 "MailFragment"

Pour répertorier les mails, nous avons développé le fragment `MailFragment`. Il hérite également du fragment `HomeFragment` avec une surcharge sur les méthodes `queryData()`, `generateAdapter()` et `touchListener()`. Ces surcharges permettent de charger les bonnes informations avec `getDiscussion()` et en indiquant au `SpecialAdapter` le type "Discussion".

```

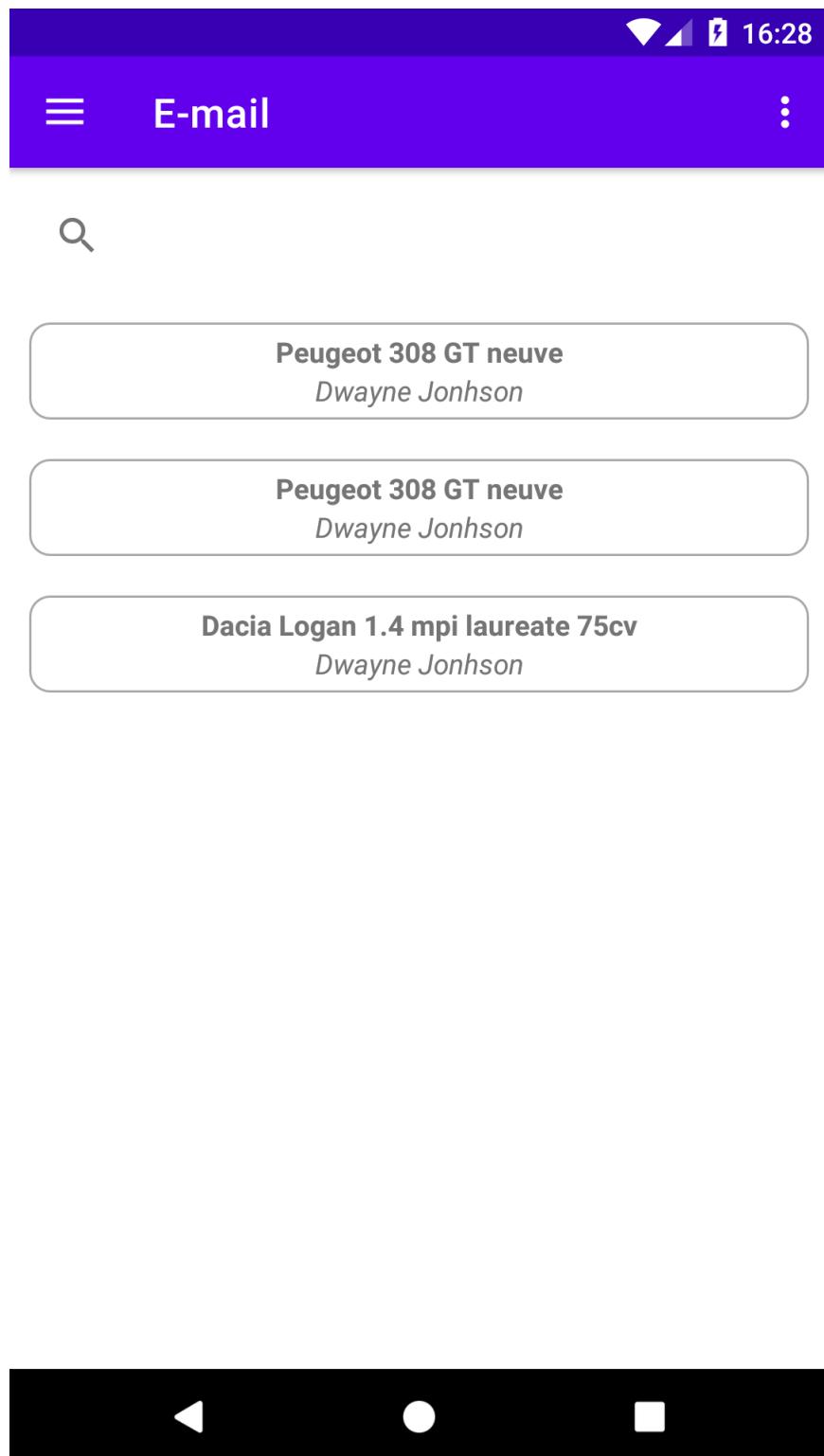
1 @Override
2 protected void queryData(int idClient) {
3     [...]
4     CommunicationWebservice.getINSTANCE().getDiscussions(0, liste -> { [...] });
5     [...]
6 }
```

```

1 @Override
2 protected SpecialAdapter generateAdapter() {
3     return new SpecialAdapter(data, SpecialAdapter.Type.Discussion, getContext());
4 }
```

```

1 @Override
2 protected void touchListener() {
3     [...]
4     Bundle b = new Bundle();
5     b.putSerializable("discussion", mvm.getDiscussion());
6     [...]
7 }
```



Liste des e-mail

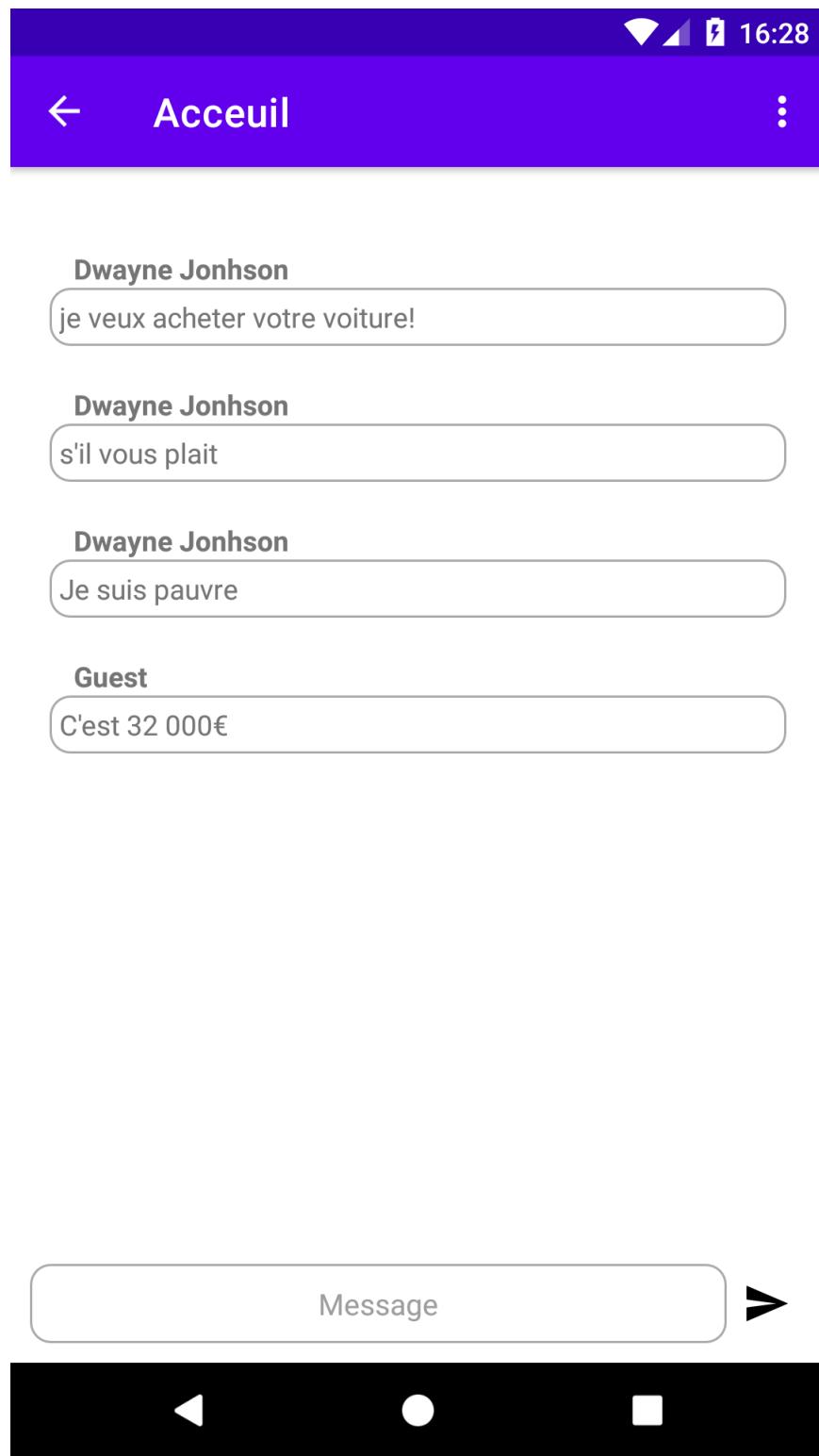
### 1.4.2 "DiscussionFragment"

Dans chaque Email réside une Discussion, qui est représenté par le fragment DiscussionFragment. Une Discussion implique la conversation entre deux utilisateurs qui s'échangent des messages.

Pour faire cela nous utilisons un recycleur de vue avec SpecialAdapter, mais cette fois-ci avec le MessageViewModel.

Voici ci-dessous la fonction permettant d'envoyer un message dans la discussion. D'abord, nous vérifions la présence d'un message, puis nous générerons la date à laquelle le message va être envoyé, et enfin nous l'envoyons grâce à la fonction CommunicationWebservice.getInstance().sendMessage()

```
1 binding.msgButtonSend.setOnClickListener(view -> {
2     String msg = binding.msgEditMessage.getText().toString();
3     if(msg.trim().length() > 0){
4         Date date = new Date(System.currentTimeMillis());
5         SimpleDateFormat format = new SimpleDateFormat("yyyyMMddHHmmss");
6         Message m = new Message(msg, Metier.getInstance().getUtilisateur(),
7             format.format(date).toString(), discussion);
8         CommunicationWebservice.getInstance().sendMessage(discussion, m, null, callback,
9             getApplicationContext().getContentResolver());
10        binding.msgEditMessage.setText("");
11    }
12});
```



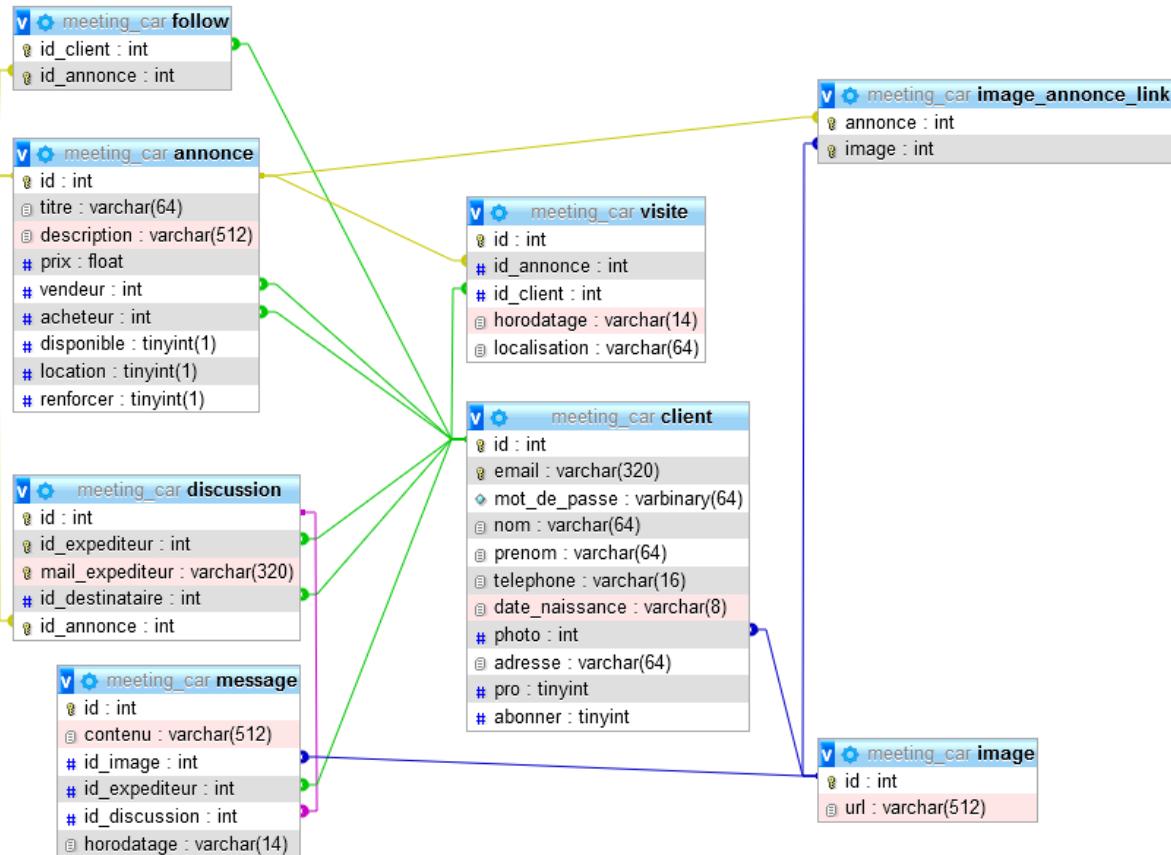
Exemple de discussion

# Chapitre 2

## Développement côté serveur

Pour pouvoir stocker les informations de l'application nous avons décidé de mettre en place un serveur web en **Node.js** utilisant le protocole HTTPS, relié à une base de données **MySQL** communiquant en SSL.

### 2.1 Structure de la base de donnée



Structure de la base de données

## 2.2 Serveur web

### 2.2.1 Configuration

Afin de permettre la configuration du serveur et l'accès à la base de données nous avons créé un fichier `access.json` contenant toutes les informations nécessaires.

```

1  {
2      "host": "",
3      "user": "",
4      "password": "",
5      "database": "meeting_car",
6      "secret": "",
7      "ssl": {
8          "ca": "/certs/ca.pem",
9          "key": "/certs/client-key.pem",
10         "cert": "/certs/client-cert.pem"
11     },
12     "https": {
13         "privateKey" : "/certs/privkey.pem",
14         "certificate" : "/certs/fullchain.pem"
15     },
16     "port" : 9000
17 }
```

Voici la structure de base du fichier, à remplir selon la configuration du serveur MySQL et HTTPS :

- **host** : correspond à l'URL ou l'IP du serveur de la base de données (exemple : localhost)
- **user** et **password** : correspondent aux informations de connexion à la base.
- **database** : correspond à la base de données que nous souhaitons accéder, ici `meeting_car`
- **secret** : correspond à la clé permettant le chiffrement des tokens de connexion au serveur Node.
- **ssl** : correspond aux trois certificats permettant la connexion sécurisée à la base.
- **https** : correspond aux deux certificats permettant la connexion HTTPS.
- **port** : port d'écoute du serveur.

```

1 let access = JSON.parse(fs.readFileSync('access.json'));
2
3 var privateKey = fs.readFileSync(access.https.privateKey, 'utf8');
4 var certificate = fs.readFileSync(access.https.certificate, 'utf8');
5 var credentials = {key: privateKey, cert: certificate};
6
7 var con = mysql.createConnection({
8     host: access.host,
9     user: access.user,
10    password: access.password,
11    database: access.database,
12    ssl: {
13        ca: fs.readFileSync(__dirname + access.ssl.ca),
14        key: fs.readFileSync(__dirname + access.ssl.key),
15        cert: fs.readFileSync(__dirname + access.ssl.cert)
16    }
17});
```

Dans un premier temps nous récupérons le fichier de configuration, puis nous créons les informations HTTPS avant de venir se connecter à la base de données.

## 2.2.2 Structure serveur

```

/
|
+-certs/
+-stockage/
|
+-access.json
+-empty_access.json
+-webservice.js

```

- **certs** : dossier stockant les certificats SSL pour MySQL et HTTPS.
- **stockage** : dossier stockant les images de profil et d'annonce.
- **access.json** : fichier de configuration du serveur.
- **empty\_access.json** : fichier de configuration vide, servant à créer **access.json**.
- **webservice.js** : exécutable du serveur.

## 2.2.3 Connexion utilisateur

Pour tout ce qui est de la connexion d'un utilisateur au serveur nous avons décidé d'utiliser JsonWebToken, permettant d'enregistrer un token pour chaque utilisateur connecté et de les retrouver.

```

1 app.get('/connection',function (req, res) {
2     let username = con.escape(req.headers.username);
3     let password = "0x" + req.headers.password;
4     if(username.length > 0 && password.length > 0){
5         con.query("SELECT * FROM client WHERE email = " + username + " AND
6             → mot_de_passe = " + password + " LIMIT 1", function (err, result) {
7                 if (err) throw err;
8                 if(result.length > 0){
9                     let utilisateur = result[0];
10                    const token = jwt.sign({
11                         id: utilisateur.id,
12                         username: utilisateur.email
13                     }, access.secret, { expiresIn: '1 hours' });
14                     res.json({user : utilisateur, access_token: token});
15                 } else {
16                     res.json({error:"unknown"});
17                 }
18             });
19     } else {
20         res.json({error:"NoNameOrPassword"});
21     }
22 });

```

Nous pouvons voir ici la fonction permettant la connexion d'un utilisateur s'il existe et que les mots de passe correspondent.

La fonction `jwt.sign` permet de créer un token et de stocker l'id de l'utilisateur.

```

1 const authenticateJWT = (req, res, next) => {
2   const token = req.headers.authorization;
3   if (token) {
4
5     jwt.verify(token, access.secret, (err, user) => {
6       if (err) {
7         return res.sendStatus(403);
8       }
9       req.user = user;
10      next();
11    });
12  } else {
13    res.sendStatus(401);
14  }
15};

```

authenticateJWT permet de vérifier si un utilisateur a bel et bien accès à la page, avec son token.

```

1 app.get('/discussion/one/:id', authenticateJWT, function(req,res){
2   let idUser = req.user.id;
3   [...]
4 });

```

Il est appelé directement dans l'appel de la fonction get d'express, pour les pages nécessitant une connexion.

Le serveur permet de réaliser plusieurs tâches :

- Création, récupération et modification d'un client ;
- Création, récupération et modification d'une annonce ;
- Récupération d'une liste d'annonce de façon générale, ou précise (qu'un utilisateur suit, ou qu'il a acheté.) ;
- Création, récupération d'une discussion et de ses messages ;
- Création d'un message dans une discussion ;
- Récupérations d'une liste de plusieurs discussions ;
- Connexion et déconnexion d'un utilisateur ;
- Vérification de connexion d'un utilisateur ;

Exemple d'une fonction de récupération :

```
1 app.get('/annonce/page/:page', function(req, res){
2     let page = parseInt(req.params.page);
3
4     if( page >= 0){
5         let limite = 20;
6         let sql = "SELECT annonce.* , V.nom AS vendeur_nom, V.prenom AS
7             vendeur_prenom, V.photo AS vendeur_photo, A.nom AS acheteur_nom,
8             A.prenom AS acheteur_prenom, A.photo AS acheteur_photo,
9             GROUP_CONCAT(DISTINCT image_annonce_link.image SEPARATOR ',') AS
10            images_id"
11        + " FROM annonce"
12        + " INNER JOIN client as V ON V.id = annonce.vendeur"
13        + " LEFT JOIN client as A ON A.id = annonce.acheteur"
14        + " LEFT JOIN image_annonce_link ON image_annonce_link.annonce =
15            annonce.id";
16        con.query(sql + " GROUP BY annonce.id LIMIT ? OFFSET ?",
17        [limite,(page*limite)], function (err, result) {
18            if (err) throw err;
19            res.json({result : result});
20        });
21    } else {
22        res.json({error:"NotValid"});
23    }
24});
```

Fonction permettant la récupération de 20 articles selon la page voulue.

# Chapitre 3

## Déploiement du projet

### 3.1 Mise en place de l'application

Pour cette partie il est nécessaire d'avoir `Android Studio` installé. Vous pourrez trouver dans la base du code, au sein du dossier `MeetingCarAndroid` le projet `Android`.

Vous allez devoir l'ouvrir avec `Android Studio` pour pouvoir le compiler. Si vous souhaitez utiliser un serveur autre que notre serveur en ligne vous pouvez modifier `BASE_URL` dans la classe `fr.flareden.meetingcar.Config` pour mettre la valeur de votre serveur.

Pour créer votre serveur veuillez suivre les section 3.2 et 3.3.

### 3.2 Mise en place de la base de données

Pour cette partie il est nécessaire d'avoir un serveur MySQL installé et configuré. Vous pourrez trouver dans la base de code, au sein du dossier `MeetingCarDatabase`, le fichier `meeting_car_database.sql`, permettant la génération avec les références de la base de données `meeting_car`.

Vous pouvez l'importer dans `PhpMyAdmin` ou l'exécuter dans le terminal SQL.

### 3.3 Mise en place du serveur

Pour cette partie il est nécessaire d'avoir `Node.JS` installé. Vous pourrez trouver dans la base de code, au sein du dossier `MeetingCarWebservice` le serveur web du projet.

Pour le faire fonctionner vous devez d'abord avoir mis en place la base de données, puis il faudra créer un fichier `access.json`. Ce fichier est une copie de `empty_access.json`, sur lequel il faut remplir les différents détails.

Pour l'exécuter vous pouvez simplement faire la commande "node webservice.js".

# Bilan, conclusion et perspectives

## Avancement du projet

À l'heure actuelle, notre application est totalement fonctionnelle, et suit au mieux le cahier des charges qui nous a été donné. Ainsi notre application donne la capacité à un utilisateur d'analyser un ensemble d'annonces pour acheter ou louer des voitures. Il peut également créer un compte qui sera par la suite modifiable pour pouvoir suivre plusieurs annonces ou alors en créer. Un compte "*Premium*" peut également être créé pour pouvoir utiliser des fonctionnalités supplémentaires telles que des analyses statistiques sur les annonces créées.

## Difficultés rencontrées

Notre première difficulté a été de mettre en place un **Navigation Drawer** fonctionnel et de naviguer de façon fluide et sans erreur entre les différents fragments.

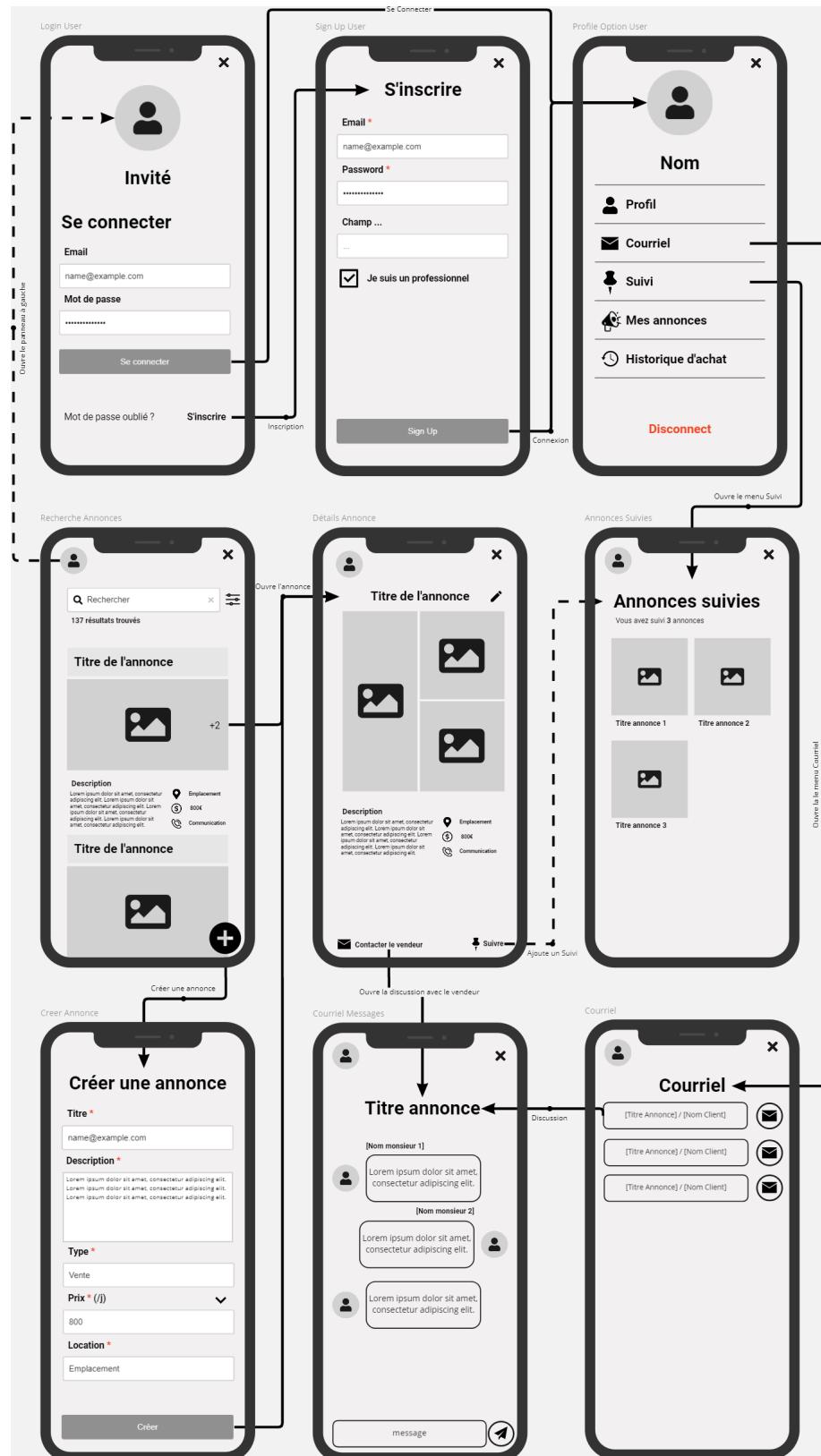
Nous avons également eu des difficultés pour la récupération et le passage des données-images entre le client et le serveur.

## Améliorations possibles

Plusieurs perspectives d'améliorations sont possibles pour ce projet dans le but de le rendre plus complet, stable et efficace telles que :

- L'ajout de la modifications des images d'une annonce.
- Limiter les appels au serveur et faire un stockage local des données.
- Faire un système de notification pour les nouveaux messages.
- Mettre en place un système de paiement sécurisé pour véritablement acheter ou louer des voitures et passer *Pro+*.

## Annexes



Maquette de l'application

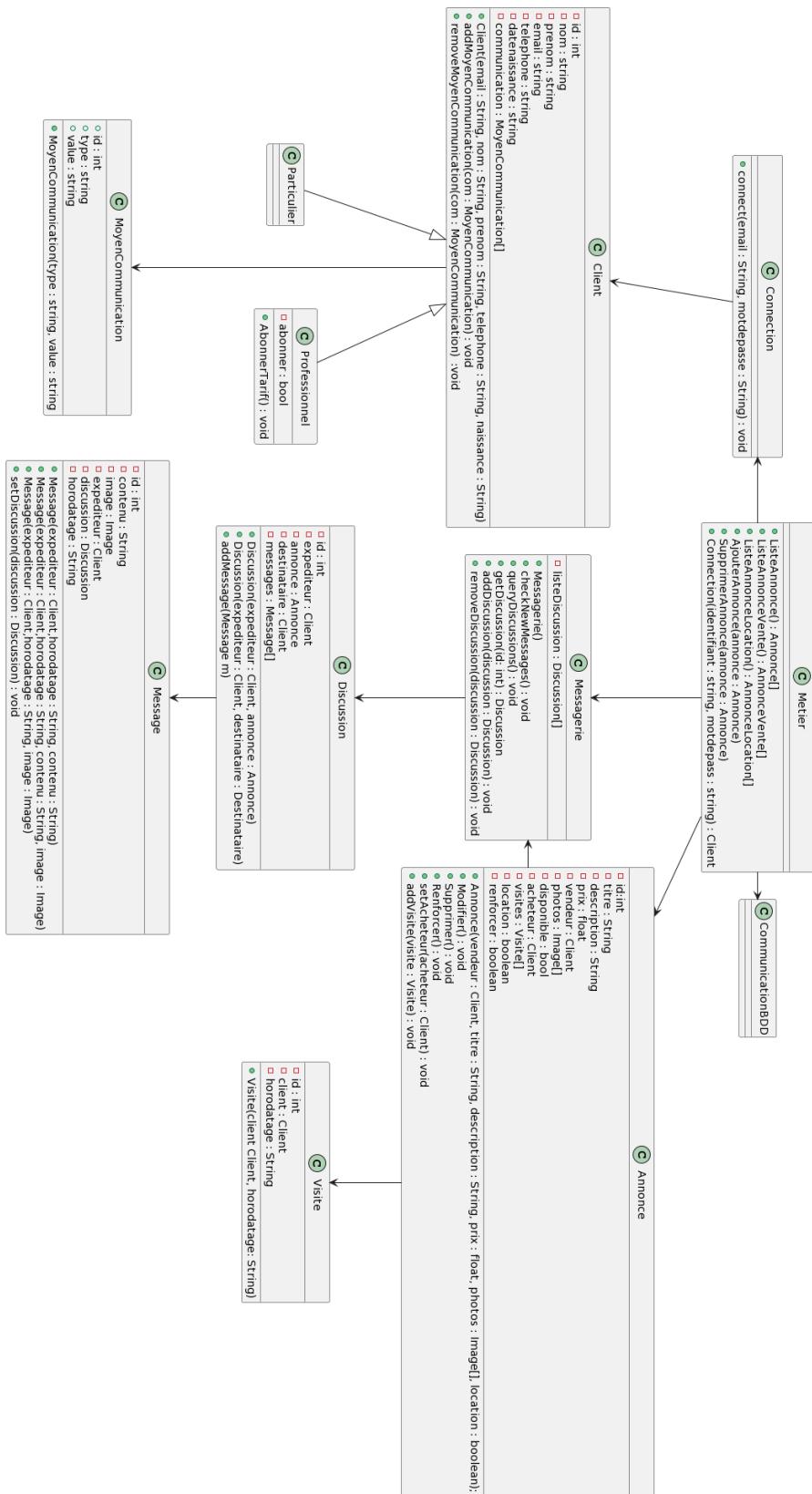


Diagramme UML

# Bibliographie

- [1] Material Design, 2022  
<https://material.io/>
- [2] Android, Site officiel, 2022  
<https://developer.android.com/>
- [3] AndroidPlot, 2022  
<http://androidplot.com/>