# DOCUMENTATION TECHNIQUE

# Gr’immo

GAËL BOTTON

2024 - 2025

# Table des matières

# Introduction

Gr'immo est une agence immobilière située à Grenoble qui prévoit de se développer dans d'autres villes. L'agence nous a donc demandé de développer un client lourd permettant de gérer leur base de données contenant l'ensemble de leurs biens, ainsi qu'un agenda et d'autres fonctionnalités.

# Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation représente les fonctionnalités principales d'un système de gestion immobilière, avec deux types d'utilisateurs identifiés : **l'agent** et **le responsable**. Voici une description des cas d'utilisation associés à chaque acteur :

**1. L'agent**

L'agent, en tant qu'utilisateur du système, dispose des fonctionnalités suivantes :

- **Gérer ses biens** : visualiser la liste des biens qu'il gère personnellement, ajouter de nouveaux biens, modifier ses biens.
- **Gérer son agenda** : organiser et planifier ses rendez-vous, visites ou événements professionnels.
- **Gérer ses clients** : suivre et gérer les informations des clients qui lui sont attribués, telles que leurs coordonnées ou leurs demandes.
- **Se connecter à l'AD (Annuaire LDAP)** : s'authentifier via un serveur LDAP pour accéder au système de manière sécurisée.

**2. Le responsable**

Le responsable dispose d'un rôle élargi avec des privilèges supplémentaires, incluant toutes les fonctionnalités des agents, ainsi que les suivantes :

- **Gérer tous les biens** : accès global à la gestion de l'ensemble des biens du système, indépendamment des agents responsables.
- **Gérer tous les clients** : possibilité de gérer les informations et interactions de tous les clients, sans restriction.
- **Ajouter un agent** : capacité d'ajouter de nouveaux agents au système, leur permettant ainsi d'utiliser la plateforme pour leurs activités.

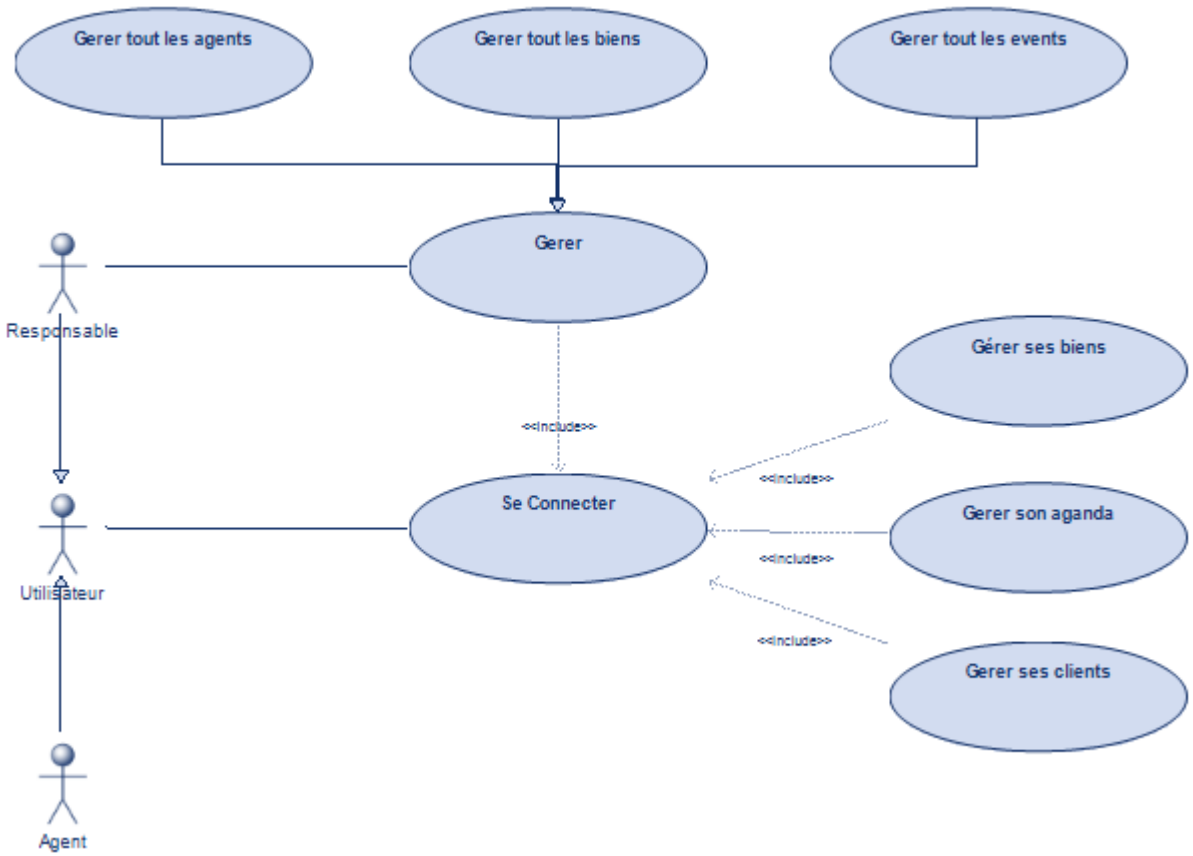


*Figure 1: Diagramme de cas*

# Base de données

## Modèle Conceptuel de Données (MCD)

Pour réaliser ce projet, nous avons dû réaliser un MCD qui nous a permis de créer la structure de la base de données de manière correcte pour répondre aux précisément aux besoins de l'application

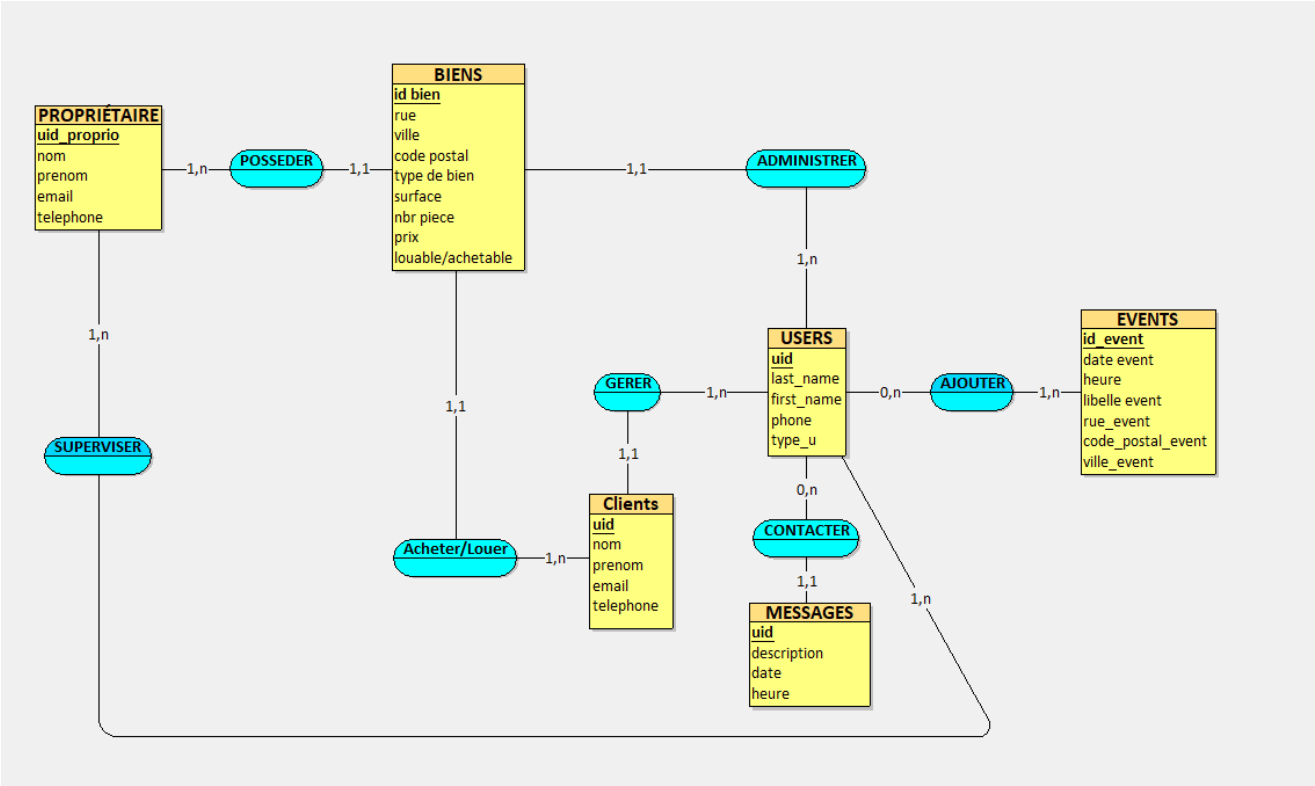


*Figure 2 : Modèle Conceptuel de Données (MCD)*

## Règle de gestion

### *Entité PROPRIÉTAIRE*

Un propriétaire peut posséder un ou plusieurs biens.

### *Entité BIEN*

Un bien doit appartenir à un et un seul propriétaire.

Un bien peut être loué ou acheté par un ou plusieurs clients.

Un bien peut être administré par un et un seul utilisateur

*Entité USERS*

Un utilisateur peut administrer un ou plusieurs biens.

Un utilisateur peut avoir zéro ou plusieurs événements.

Un utilisateur peut avoir un ou plusieurs clients.

Un utilisateur peut avoir zéro ou plusieurs messages.

Un utilisateur peut superviser un ou plusieurs propriétaires.

*Entité EVENTS*

Un événement doit être ajouté par un et un seul utilisateur.

*Entité Clients*

Un client est géré par un et un seul utilisateur.

Un client peut être associé à un ou plusieurs biens

*Entité MESSAGES*

Un message doit être envoyé par un et un seul utilisateur.

Un message est caractérisé par une description, une date et une heure.

# Modèle Logique de Données (MLD)

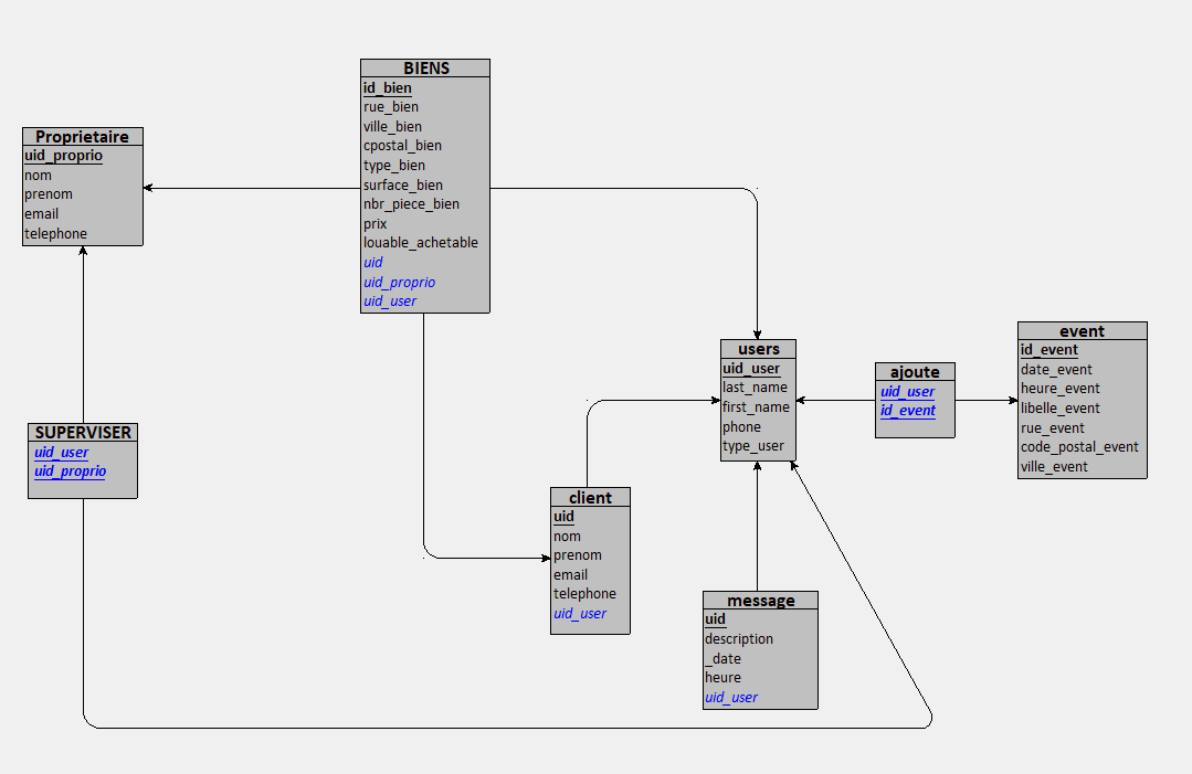


*Figure 3 : Modèle Logique de Données (MLD)*

## Relations

*Entité BIENS* :

uid_user : Clé étrangère faisant référence à la clé primaire uid_user de l'entité USERS.

uid_proprio : Clé étrangère faisant référence à la clé primaire uid_proprio de l'entité PROPRIETAIRE.

*Entité SUPERVISER* :

uid_user : Clé étrangère faisant référence à la clé primaire uid_user de l'entité USERS.

uid_proprio : Clé étrangère faisant référence à la clé primaire uid_proprio de l'entité PROPRIETAIRE.

*Entité CLIENT* :

uid_user : Clé étrangère faisant référence à la clé primaire uid_user de l'entité USERS.

*Entité AJOUTE* :

uid_user : Clé étrangère faisant référence à la clé primaire uid_user de l'entité USERS.

id_event : Clé étrangère faisant référence à la clé primaire id_event de l'entité EVENT.

*Entité MESSAGE* : uid_user : Clé étrangère faisant référence à la clé primaire uid_user de l'entité USERS.

*Entité EVENT* : Pas de clé étrangère directe visible, mais liée à AJOUTE via la clé primaire id_event.

# Diagramme de classe

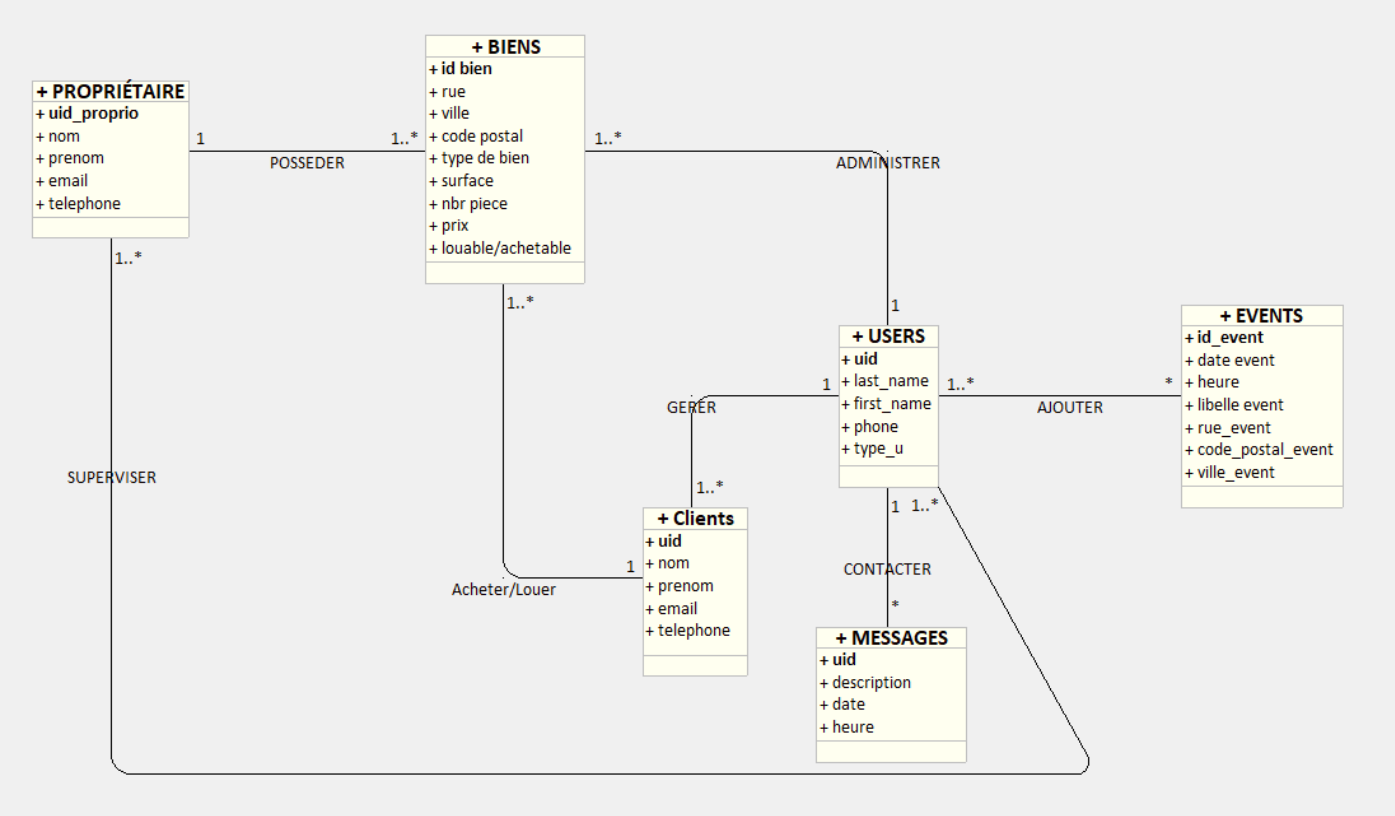


*Figure 4 : Diagramme de classe*

La classe **PROPRIÉTAIRE** représente les propriétaires de biens immobiliers, avec des informations personnelles comme leur nom, prénom, email et téléphone. Chaque propriétaire possède un ou plusieurs biens.

La classe **BIEN** décrit les propriétés immobilières avec des caractéristiques comme l'adresse, la surface, le prix et le statut (louable ou achetable). Chaque bien est associé à un propriétaire et peut être administré par des utilisateurs.

La classe **USERS** représente les utilisateurs du système, comme les agents ou les responsables, avec des informations telles que leur nom, prénom, téléphone et type (agent ou responsable). Ils peuvent administrer des biens et ajouter des événements.

La classe **CLIENTS** regroupe les personnes intéressées par l'achat ou la location de biens. Chaque client est géré par un utilisateur et peut être associé à un ou plusieurs biens.

La classe **EVENTS** contient les événements liés au système, comme des visites de propriétés, avec des informations comme la date, l'heure et le lieu. Les événements sont ajoutés par des utilisateurs.

La classe **MESSAGES** permet de gérer la communication entre utilisateurs et clients, avec des informations comme la description, la date et l'heure d'envoi.

Les relations entre les classes définissent les interactions : les propriétaires possèdent des biens, les utilisateurs administrent des biens et ajoutent des événements, les clients achètent ou louent des biens, et les messages facilitent la communication.

# Répartition des tâches au sein de l'équipe

Dans le cadre de notre projet de développement d'un logiciel pour une agence immobilière, nous avons adopté une méthodologie collaborative qui nous a permis de répartir les tâches efficacement et de suivre l'avancement du projet de manière structurée. L'équipe était composée de trois membres : Issam Moussi, Lucas Troussier et moi-même.

Pour organiser notre travail, nous avons utilisé le logiciel **Gantt Project**, qui nous a permis de planifier les différentes étapes du projet et de répartir les tâches entre les membres de l'équipe. Cet outil a été essentiel pour établir un planning détaillé et pour définir les responsabilités de chacun tout au long du développement du logiciel.

En complément, nous avons également utilisé la plateforme **Trello** pour le suivi quotidien de l'avancement du projet. Trello nous a permis de centraliser nos tâches sous forme de tableaux, avec des colonnes représentant les différentes étapes du workflow : "To Do", "In Progress", et "Finish". Chaque tâche était associée à un membre de l'équipe, ce qui facilitait la collaboration et assurait une visibilité complète sur l'état d'avancement.

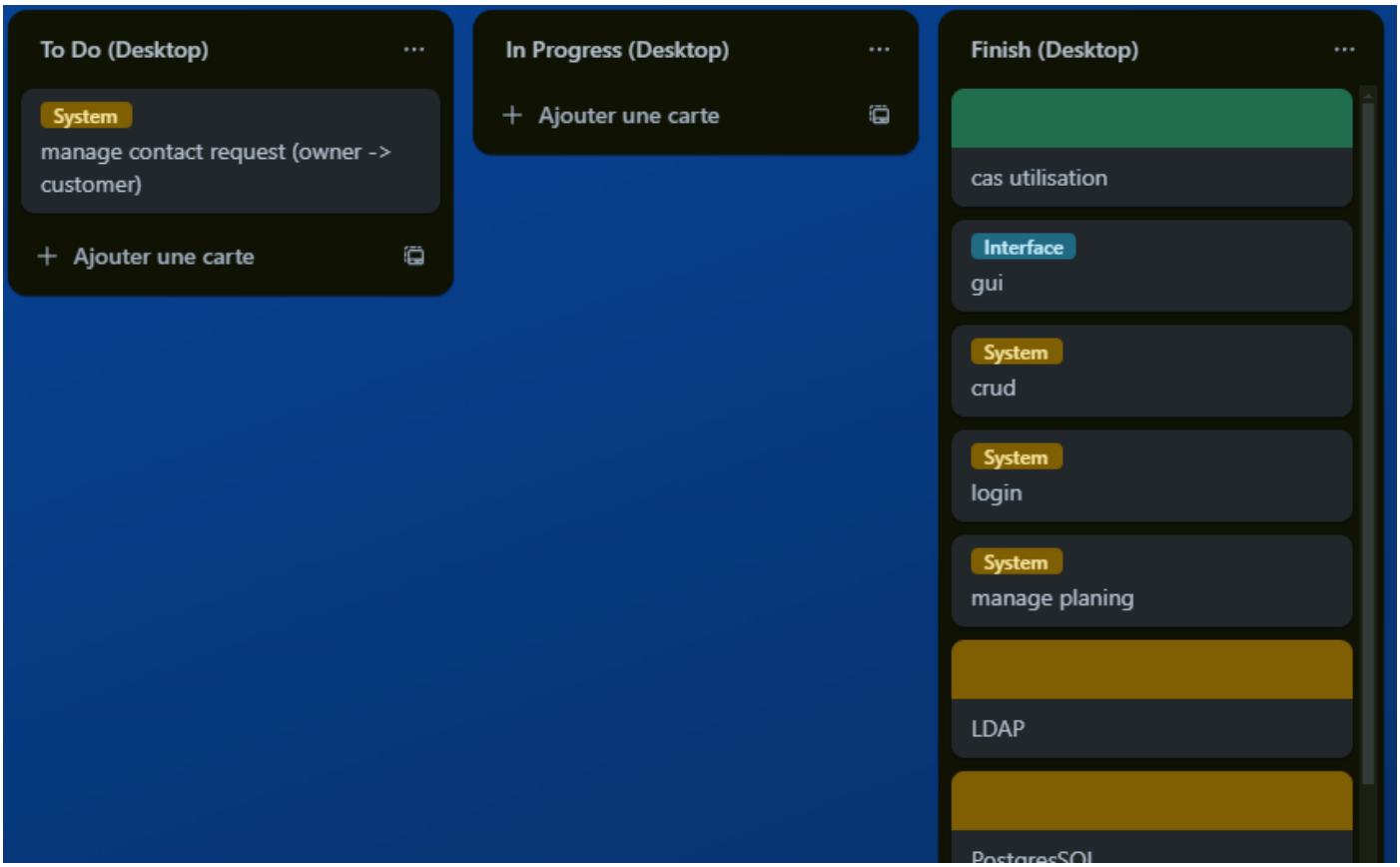


*Figure 5 : Tâche Trello*

Cette approche méthodologique nous a permis de travailler de manière fluide et efficace, en réduisant les risques de confusion ou de chevauchement des tâches. Chacun jouait un rôle précis :

- **Issam Moussi** s'est concentré développement de l'interface utilisateur et la base de données
- **Lucas Troussier** a pris en charge le développement de l'interface utilisateur et la base de données.
- Quant à moi, je me suis occupé du développement back-end et des fonctionnalités.

# Développement de l'application

Pour le développement de l'application nous avons décidé d'utiliser le langage **Python** et de se baser sur une architecture **MVC** (Modèle-Vue-Contrôleur).

Notre choix pour le langage **Python** plutôt qu'un autre langage se démarque du fait, que c'est un langage que nous maitrisons, qui est suffisamment puissant pour l'application que nous devons développer et aussi la faite que ce langage possède une très grosse communauté fournissant un large choix en librairie.

L'architecture **MVC (Modèle-Vue-Contrôleur)** est un **modèle d'architecture logicielle** qui sépare une application en trois composants principaux, chacun jouant un rôle bien défini. Ce découplage permet de mieux organiser le code, de faciliter sa maintenance, et de rendre l'application plus modulaire.

Le modèle : Il joue le rôle d'intermédiaire entre le contrôleur et la base de données. C'est lui qui envoie les requêtes à la base de données pour manipuler les données et, si besoin, les renvoie au contrôleur pour qu'il puisse les utiliser.

La vue : Elle permet d'afficher les pages ou interfaces graphiques à l'utilisateur. C'est la partie **frontend** de l'application.

Le contrôleur : Il gère la logique métier derrière chaque vue. Il réagit aux actions effectuées par l'utilisateur sur la vue, les traite en conséquence, et travail avec le modèle pour récupérer des données dans la base de données, puis les transmettre à la vue afin qu'elle les affiche.

Pour le développement de l'application, nous avons dû utiliser une multitude de librairies. Les voici :

- **PYQt5 :** Cette librairie est un Framework (également disponible en C++) qui permet de créer des interfaces graphiques dynamiques et interactives.
- **LDAP3 :** Elle permet d'établir une communication entre une application et un serveur LDAP distant ou local.
- **Psycopg2 :** Cette librairie permet d'établir une communication entre une application et un serveur PostgreSQL distant ou local.
- **Configparser :** Elle permet de lire un fichier de configuration en « .cfg ».
- **PrivateCode :** en Python, l'encapsulation, au sens strict, n'existe pas réellement. Contrairement à d'autres langages comme C++ ou Java, où l'accès aux méthodes, fonctions ou classes peut être strictement limité via des modificateurs comme private, protected ou public, Python repose sur une convention purement visuelle. Par exemple, lorsqu'une méthode, fonction ou classe est préfixée par un underscore _, elle est considérée comme privée ou protégée, mais rien n'empêche un développeur d'y accéder directement. Cela reste une convention et n'impose aucune restriction réelle.

  Pour pallier cette limitation, j'ai créé une librairie qui permet d'ajouter un niveau de contrôle plus strict à l'encapsulation en Python. Grâce à un décorateur fourni par cette librairie, il est possible de 'rendre privées' certaines méthodes, fonctions ou classes. Concrètement, cette librairie agit comme un filtre en interceptant les appels et en vérifiant leur origine. Si un appel provient d'un module ou d'une classe externe, il est bloqué.

  Cependant, il est important de noter que cette solution n'agit pas au niveau des basses couches, comme la mémoire ou le compilateur. Elle reste au niveau des couches hautes pour uniquement traiter et vérifier les appels, tout en évitant des comportements non souhaités. Cela permet d'émuler une encapsulation stricte en Python, sans modifier son fonctionnement sous-jacent.

# Développement des communications serveur

Les deux premières étapes réalisées durant le développement du logiciel ont été, le développement d'une fonction nous permettant d'établir une connexion avec la base de données pour ainsi exécuter des requêtes nous permettant d'interagir avec les données présentes dans celle-ci. La deuxième fonction a été de créer une fonction nous permettant d'établir une connexion avec le serveur LDAP pour par la suite, pouvoir nous connecter avec des comptes utilisateur présents sur le serveur.

## Connexion et communication avec le serveur LDAP

Pour la connexion et la communication avec le serveur LDAP, nous avons utilisé le langage Python et la librairie LDAP3 qui est une librairie fournissant des méthodes et des classes permettant de se connecter à un serveur LDAP à distance ou bien local et de pouvoir communiquer avec lui.

Pour réaliser cette connexion au serveur LDAP, nous avons créé une classe python qui sera instancié lors du chargement de l'application. Cette classe s'instanciera avec les données de connexion stockées dans le fichier de configuration « config.cfg ».

```
[LDAP]
ldap_url=ldap://192.168.30.47
ldap_port= 389
ldap_base=DC=gli,DC=local
ldap_domain=@gli.local
```

*Figure 6 : Configuration LDAP*

Voici la structure de la classe LDAP :

```python
class LDAPServer:
    """
    This class is used to interact with an LDAP server

    Args:
        - ldap_url : str : The URL of the LDAP server | Example: "ldap://127.0.0.1:389"
        - ldap_port : int : The port of the LDAP server | Example: 389
        - ldap_login : str : CN of the user | Example: "Admin
        - ldap_password : str : The password of the user | Example: "password"
        - search_base : str : The search base of the LDAP server | Example: "DC=example,DC=com"

    Returns:
        - None
    """
    #instance Methodes
    def __init__(self, ldap_url : str, ldap_port : int, ldap_domain : str, search_base: str) -> None:
        self.BASE = search_base;
        self.server = Server(ldap_url, ldap_port, get_info=ALL)
        self.CONN = None
        self.DOMAIN = ldap_domain
```

*Figure 7 : Structure de la classe LDAPServer*

Pour que cette classe se construise correctement lors d'une instanciation, il faut lui passer l'URL du serveur, le port, le domaine, et la base de données.

Cette classe fournie une multitude de méthodes utile à la communication entre le client et le serveur LDAP.

Par exemple la classe fournit une méthode de connexion au serveur LDAP.

```python
def login(self, ldap_login : str, ldap_password : str):
    try:
        #Instanciate the connection
        self.CONN = Connection(self.server, f"{ldap_login}{self.DOMAIN}", ldap_password, auto_bind=True)
        return True

    except Exception as e:
        print(e)
        print("Error while connecting to the LDAP server")
        return False
```

*Figure 8 : LDAP Login méthode*

Cette méthode a besoin d'un mot de passe, et d'un login pour se connecter. Elle peut être utilisée uniquement après l'instanciation de la classe LDAP, car l'instanciation de la classe permet de créer la structure du serveur cible et donc, par la suite, de s'y connecter.

Une fois la connexion établie avec le serveur, nous pouvons désormais communiquer avec celui-ci. Pour ce faire nous avons créé des méthodes permettant de réaliser des tâches très précise et indispensable à l'utilisation de notre application. Par exemple la méthode « get_group » :

```python
def get_groups(self, cn_user : str) -> list:
    #Get Distinguished Name of the user
    if self.CONN.search(self.BASE, f'(sAMAccountName={cn_user})', attributes=['distinguishedName']):
        user_dn = self.CONN.entries[0].distinguishedName

        #Group search filter
        group_search_filter = f'(&(objectClass=group)(member={user_dn}))'

        #Search for the group
        try:
            self.CONN.search(self.BASE, group_search_filter, attributes=['cn'])
            #Return list of user groups
            return [group.cn for group in self.CONN.entries]
        except:
            return print("An Error occured while searching for groups")
```

*Figure 9 : LDAP get_group méthode*

Cette méthode permet de vérifie les groupes AD auquel l'utilisateur qui vient de se connecter appartient. Pour se faire, notre application envoie une requête au serveur qui va aller rechercher avec le filtre créer par la méthode tous les groupes de l'utilisateur. Avec cette méthode, cela nous permet de nous assurer que l'utilisateur connecté possède ou non, le ou les groupes nécessaires pour l'utilisation d'une fonctionnalité plus restreinte.

La classe possède aussi la méthode « add_user »

```python
def add_user(self, last_name, first_name, password, gp_name, uuid4) -> bool:
    try:
        full_name = f"{last_name.upper()} {first_name.upper()}"
        user_id = f"{first_name[0].upper()}{last_name[0].upper()}{last_name[1:].lower()}"

        print(password)
        # Attributes of new user
        user_attributes = {
            'objectClass': ['person', 'top', 'organizationalPerson', 'user'],
            'cn': full_name,
            'sn': last_name,
            'givenName': first_name,
            'userPassword': str(password),
            'sAMAccountName': user_id,
            'userPrincipalName': f"{user_id}{self.DOMAIN}",
            'displayName': full_name,
            'uid' : str(uuid4),
            'pwdLastSet' : -1,
            'userAccountControl' : 544
        }

        # User DN
        user_dn = f"CN={full_name},OU=Users,OU=Grimmo,{self.BASE}"
        # Add user to AD
        self.CONN.add(user_dn, attributes=user_attributes)

        if self.CONN.result['result'] == 0:
            print("User added to AD")

            # Groupe DN
            group_dn = f"CN={gp_name},OU=Groups,OU=Grimmo,{self.BASE}"
            # Add user to group
            self.CONN.modify(group_dn, {'member' : [(MODIFY_ADD, [user_dn])]})

            if self.CONN.result['result'] == 0:
                print("User added to group")
            else:
                print(self.CONN.result['description'])
                return False

            # Active account
            """self.CONN.modify(user_dn, {'userAccountControl': [(MODIFY_REPLACE, [512])]})
            if self.CONN.result['result'] == 0:
                print("ok")
            else:
                print(self.CONN.result['description'])"""

            return True
        else:
            print(self.CONN.result['description'])
            return False

    except Exception as e:
        print(e)
        return False
```

*Figure 10 : Add user LDAP méthode*

Cette méthode permet d'ajouter de nouveaux utilisateurs (agent dans le contexte de l'agence immobilière) au serveur LDAP. Cela nous permet en ne passant que par l'application de directement créer de nouvel utilisateur AD sur le serveur LDAP qui pourront par la suite se connecter et utiliser, eux aussi l'application.

Ensuite, nous avons également créé une méthode permettant de fermer proprement la connexion entre le client et le serveur. Cela vise à minimiser au maximum les risques d'insécurité liés à une ouverture constante et inutile des ports de communication de la machine cliente vers le serveur LDAP lorsqu'ils ne sont plus utilisés.

```python
def disconnect(self) -> None:
    try:
        self.CONN.unbind()
    except:
        print("An error occured while disconnecting from the LDAP server")
    print("Disconnected from the LDAP server")
```

*Figure 11 : Déconnexion du serveur LDAP*

Enfin la classe possède aussi une méthode nous permettant de supprimer des utilisateurs AD du serveur LDAP.

```python
def delete_user(self, cn_user : str) -> bool:
    print(cn_user)
    try:
        if self.CONN.search(self.BASE, f'(sAMAccountName={cn_user})', attributes=['distinguishedName']):
            user_dn = self.CONN.entries[0].distinguishedName.value

            self.CONN.delete(user_dn)
            return True
        else:
            return False
    except Exception as e:
        print(e)
        return False
```

*Figure 12 : Supprimer un utilisateur sur le serveur LDAP*

Cette méthode nous permet d'envoyer une requête de suppression au serveur LDAP via l'application cliente pour supprimer un utilisateur quelconque. C'est très utile, car avec la méthode de création d'utilisateurs, nous pouvons gérer de manière simple et efficace tous les utilisateurs présents sur le serveur via l'application.

# Connexion et communication avec la base de données PostgreSQL

Comme pour la réalisation de la classe LDAP, nous avons utilisé le langage Python et une librairie faite exprès pour les connexion et communication entre un client et un serveur PostgreSQL, psycopg2.

Voici la structure de la classe pour la communication et connexion à un serveur PostgreSQL :

```python
class Database:
    def __init__(self, host : str, db_name : str, user : str, pdw : str):
        self.host = host
        self.db_name = db_name
        self.user = user
        self.pdw = pdw
        self.CONN = None
        self.cursor = None
```

*Figure 13 : Structure de la classe Database*

Cette classe permet la création d'un client ayant la capacité de communiqué avec le serveur local ou distant. Cette classe se fait instancier avec la configuration de la base de données stockée dans le fichier de configuration « config.cfg ».

```
[POSTGRES]
host=192.168.30.47
database=Grimmo
user=postgres
password=Admin2025/
```

*Figure 14 : Configuration serveur PostgreSQL*

Pour que cette classe se construise correctement lors d'une instanciation, il faut lui passer l'host de la base de données, le nom de la base sur le serveur, le login et le mot de passe de connexion.

Cette classe met à disposition plusieurs méthodes permettant la communication et la connexion à un serveur PostgreSQL.

Par exemple la méthode « connect » :

```python
def connect(self):
    try:
        self.CONN = psycopg2.connect(
            host=self.host,
            database=self.db_name,
            user=self.user,
            password=self.pdw
        )
        self.CONN.autocommit = True
        self.cursor = self.CONN.cursor()
        return True
    except Exception as e:
        print(e)
        return False
```

*Figure 15 : Méthode de connexion au serveur PostgreSQL*

Cette méthode affecte à une variable qui sera utilisée tout au long de l'application, la connexion au serveur PostgreSQL. Avec cette variable stockant la connexion au serveur de la base de données, l'application peut continuellement communiquer avec ce dernier.

Ensuite nous avons la méthode « query ».

```python
def query(self, query) -> list[Union[bool,object]]:
    if not self.CONN or self.CONN.closed:
        return [False, "Connection to the Database is closed"]
    try:
        self.cursor.execute(query)
        return [True, self.cursor]
    except Exception as e:
        self.CONN.rollback()
        return [False, e]
```

*Figure 16 : Méthode query*

Cette méthode possède un paramètre « query » et va utiliser l'argument passé au paramètre pour exécuter une query (requête) sur la base de données et renvoyer le résultat à la fonction qui à appeler cette méthode. Nous utilisons cette méthode dans des fonctions de l'application ayant besoin d'envoyer des requêtes à la base de données pour récupérer ou modifier des données.

Enfin la fonction « disconnect ». Cette fonction à exactement la même utilité que la fonction « disconnect » de la classe LDAPServer, c'est-à-dire, fermer la connexion entre le client et le serveur.

```python
def disconnect(self) -> None:
    try:
        self.CONN.close()
    except:
        print("An error occured while disconnecting from the Database")
    print("Disconnected from the Database")
```

*Figure 17 : Méthode de déconnexion de la classe DataBase*

# Développement de l'interface graphique

Comment mentionné plus haut dans les librairies / Framework utilisés, nous avons utilisé le Framework PyQt5 pour réaliser l'interface graphique.

Ce très gros Framework nous met à disposition un logiciel, « Qt Designer », permettant de designer une interface graphique par simple « drag and drop ».

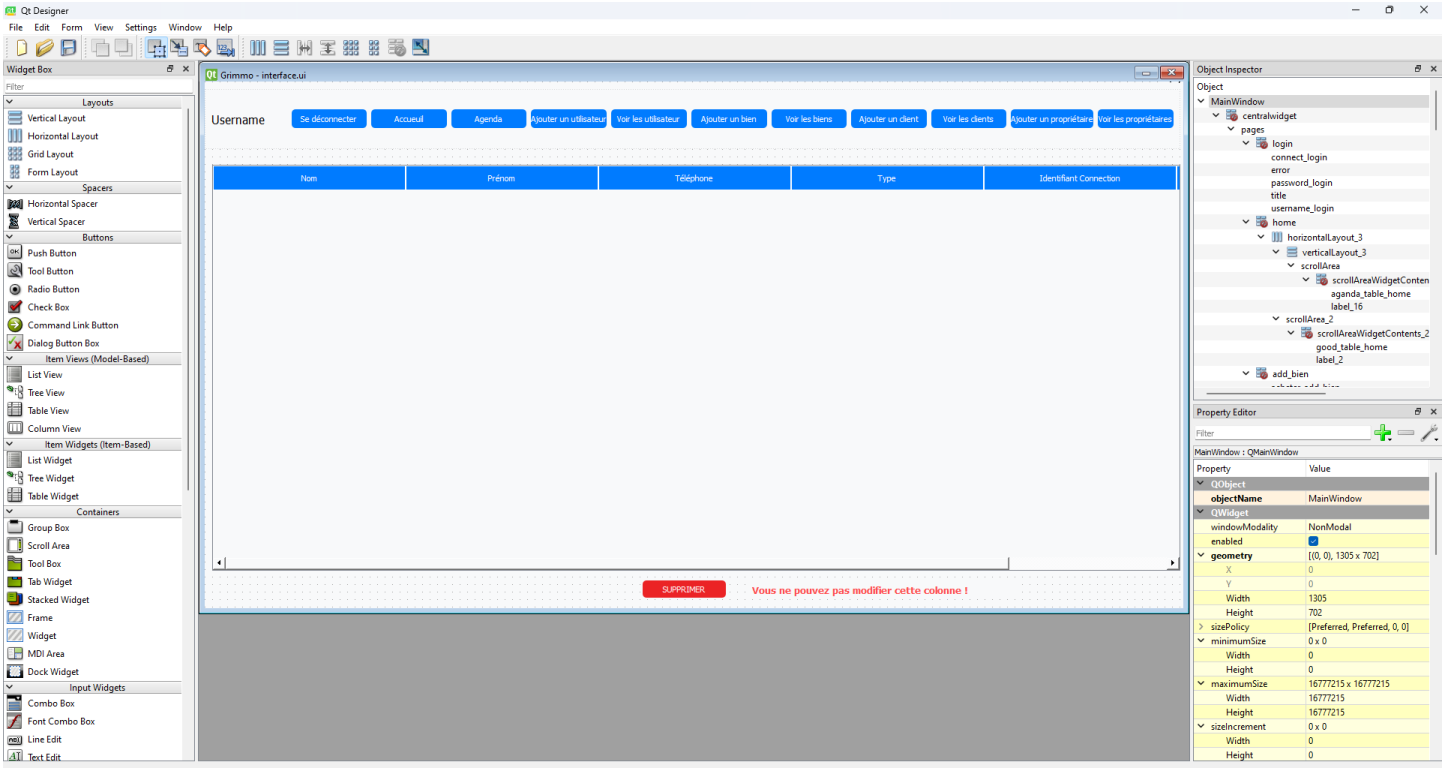


*Figure 18 : Interface QT Designer*

Ce logiciel est très pratique, car cela permet de créer une interface graphique beaucoup plus rapidement qu'avec du code comme nous le propose la librairie standard à Python nommé **Tkinter** ou encore **Kivy**.

Après avoir créé l'interface graphique, le logiciel nous permet soit de directement la convertir en fichier « .py « (Python) ou alors de nous générer un fichier avec l'extension « .ui » qui peut être interprété par un loader fourni par le Framework PyQt5.

Voici un exemple d'un fichier .ui qui peut être interprété par le loader sur Framework :

```
<widget class="QLabel" name="mois_label">
 <property name="geometry">
  <rect>
   <x>1050</x>
   <y>442</y>
   <width>71</width>
   <height>21</height>
  </rect>
 </property>
 <property name="font">
  <font>
   <pointsize>12</pointsize>
   <weight>50</weight>
   <bold>false</bold>
  </font>
 </property>
 <property name="text">
  <string>/mois</string>
 </property>
</widget>
```

*Figure 19 : Fichier .ui*

Sur cette image nous pouvons voir que l'élément est de type widget, ses coordonnées, sa taille, le de la police d'écriture, s'il est en gras, son texte, etc.

Les fichiers en .ui sont des fichiers qui définissent tous les paramètres de chaque composant d'une interface graphique.

Voici le loader qui permet de charger un fichier en .ui en Python :

```
uic.loadUi("core/GUI/interface.ui", self)
```

*Figure 20 : Loader UI*

Une fois que l'application est lancée et que le loader à charger l'interface graphique, nous pouvons récupérer chaque widget de l'interface graphique via leurs classes et leurs noms de classe (identifiant) pour les stocker dans des variables et pour utiliser leur propriété et méthode.

```
self.connect_to_ldap_btn = self.findChild(QPushButton, "connect_login")
```

*Figure 21 : Récupération d'un widget de l'interface graphique*

Avec ces variables qui stockent les widgets de l'interface graphique, on peut les utilisées pour par exemple leur ajouter un « Event Listener » (écouteur d'événement) sur l'événement de clique pour faire en sorte à ce que quand un clique est détecté, il  appel une fonction ou autre.

```
self.add_user_home.clicked.connect(lambda : self.pages.setCurrentIndex(6))
```

*Figure 22 : Écoute de l'événement click sur un bouton*

Dans ce cas-ci, quand le bouton « add_user_home » est cliqué, on va changer la page actuelle du widget « pages » qui contient toutes les pages de l'application.

# Développement des fonctionnalités principales

Les fonctionnalités principales sont développées dans le contrôleur lié à l'interface graphique comme nous sommes basés sur l'architecture MVC.

Le contrôleur possède une trentaine de fonctionnalités toutes avec un rôle et une utilité bien précise au fonctionnement de l'application.

Par exemple la méthode d'ajout de propriété :

```
def add_good(city, address, cp, type_good, surface, nbr_ro
    """
    If buy is checked commendable_purshasable = 1
    If rent is checked commendable_purshasable = 0
    """
    global user

    if city.text() == "" or address.text() == "" or cp.tex
        handle_message(error)
        return False

    if type_good.currentText() == "Appartement":
        type_good = 0
    elif type_good.currentText() == "Maison":
        type_good = 1
    else:
        type_good = 2 #Terrain

    uuid4 = str(_generate_uuid4())

    img_bytes = "null"

    if(url.text() != ""):
        with open(url.text(), 'rb') as b:
            img_bytes = base64.b64encode(b.read()).decode(

    data = {"uuid":uuid4,
            "address":address.text(),
```

*Figure 23 : Méthode d'ajout de propriété*

Cette méthode est appelée quand l'utilisateur clique sur le bouton « ajouter » de la page qui permet d'ajouter une nouvelle propriété à la base de données. Cette méthode va faire plusieurs vérifications, faire un appel API, et ensuite ajouter la propriété dans la base de données.

Ensuite nous avons développé une fonctionnalité qui permet d'ajouter un nouvel utilisateur au serveur AD.

```
@private
def _add_user_to_ad(last_name, first_name, password, gp_name, uuid4) -> bool:
    return ldap_server.add_user(last_name, first_name, password, gp_name, uuid4)
```

*Figure 24 : Fonction d'ajout d'utilisateur AD*

Cette fonction va appeler la méthode « add_user » de notre instance LDAP et retourner la réponse de l'appel de la méthode.

La méthode « add_user » de l'instance LDAP va créer un dictionnaire de donnée avec toutes les informations du compte à créer, puis venir utiliser la méthode de « add » de la connexion initialement instanciée pour envoyer les données d'ajout au serveur LDAP.

```python
def add_user(self, last_name, first_name, password, gp_name, uuid4)
    try:
        full_name = f"{last_name.upper()} {first_name.upper()}"
        user_id = f"{first_name[0].upper()}{last_name[0].upper()}{la

        print(password)
        # Attributes of new user
        user_attributes = {
            'objectClass': ['person', 'top', 'organizationalPerson',
            'cn': full_name,
            'sn': last_name,
            'givenName': first_name,
            'userPassword': str(password),
            'sAMAccountName': user_id,
            'userPrincipalName': f"{user_id}{self.DOMAIN}",
            'displayName': full_name,
            'uid' : str(uuid4),
            'pwdLastSet' : -1,
            'userAccountControl' : 544
        }

        # User DN
        user_dn = f"CN={full_name},OU=Users,OU=Grimmo,{self.BASE}"
        # Add user to AD
        self.CONN.add(user_dn, attributes=user_attributes)
```

*Figure 25 : Méthode add_user de l'instance LDAP*

Beaucoup d'autres fonctionnalités comme celles-ci existent dans ce contrôleur. Je ne vais donc pas toutes les présenter, car cela sera beaucoup trop fastidieux et ennuyant à lire.

# La Dataclass User

L'application utilise une autre classe, qui cette fois-ci est une classe uniquement composée de données. Elle ne contient aucune méthode, seulement des variables. On appelle ça une Dataclass. Cette Dataclass nous permet de stocker toutes les informations de l'utilisateur connecté. C'est une sorte de cookie de session, comme sur les sites internet.

```python
from dataclasses import dataclass

@dataclass
class User:
    cn : str
    groups : list
    uid : str
    last_name :str
    first_name : str
    phone : str
    type_u : int
    access_token : str
```

*Figure 26 : Dataclass USER*

Cette Dataclass est instanciée au début de la fonction de connexion.

```python
user = User(LDAP_CNUSER, groups, uid, None, None, None, 1, None)
```

*Figure 27 : Instance de la Dataclass User*

Lors de l'instanciation de la Dataclass, on l'instancie avec des valeurs nulles « None » pour certains paramètres pas nécessaires au début. Ensuite, si la connexion au serveur LDAP et au serveur de la base de données se déroule correctement, donc, que les informations de connexion sont correctes. On modifie chaque valeur nulle de l'instance par les vraies valeurs, qui sont les informations de l'utilisateur, comme sont uuid, nom, prénom, téléphone, token d'accès à l'api, etc.

```python
user.last_name = values[0][0]
user.first_name = values[0][1]
user.phone = values[0][2]
user.type_u = values[0][3]
user.access_token = values[0][4]
```

*Figure 28 : Modification des valeurs nulles de l'instance de la Dataclass*

Cette Dataclass nous permet de pouvoir utiliser toutes les informations de l'utilisateur connecté, très facilement. Cela nous évite de devoir les stocker dans du JSON ou bien dans un dictionnaire et de devoir utiliser de multitude de clé pour retrouver une valeur ou encore de devoir parcourir des listes, etc.

Par exemple sur la fonction « add_user » on vient utiliser la variable « type_u » de notre instance pour vérifier que l'utilisateur a bien le type requis pour utiliser cette méthode.

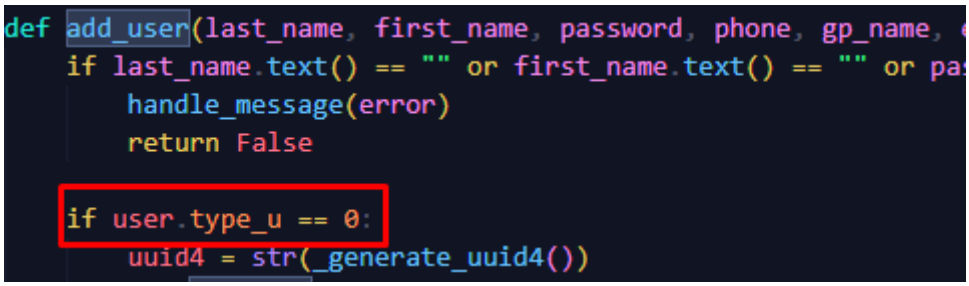


*Figure 29 : Vérification avec l'instance de la dataclass*

Cette variable a été affectée lors de la connexion à l'application, et l'on peut l'utiliser comme bon nous semble pendant toute la durée de vie du processus, grâce à la Dataclass.

Dans la fonction « _add_user_to_db » qui permet d'ajouter un nouvel utilisateur à la base de données. On fait un appel à l'API. Cette API possède un système d'authentification, avec les tokens « BEARER ». Ce système d'authentification permet d'éviter que n'importe qui puisse utiliser des méthodes de l'API. Comme cette API nécessite une autorisation. Lors de l'envoi de la requête, on utilise le token « BEARER » ou « access_token » de l'utilisateur connecté, précédemment récupéré et stocké dans l'instance de la Dataclass lors de la connexion initiale, pour authentifier la requête.

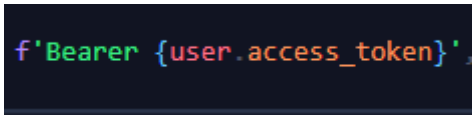


*Figure 30 : Access token*

Cette approche facilite grandement la manipulation des données d'un utilisateur connecté. On a juste à appeler l'instance et choisir la variable qui nous intéresse.

# Table des illustrations