PLONGÉE AU COEUR DE L'HISTOIRE D'UN RÉSEAU SOCIAL

PANDAMONIUM

SOMMAIRE

Chapitre 1 - La Genèse
Chapitre 2 - La Répartition des Tâches
Chapitre 3 - La Base de Données
Chapitre 4 - Explication d’une classe *by Arthur*
Chapitre 5 - Explication d’une classe *by Gaëtan*

CHAPITRE 1 - LA GENÈSE

Internet : quel outil merveilleux ! Utilisé par tous et pour tout de nos jours, permettant des prouesses qui, il y a ne serait-ce que 30 ans de cela, auraient paru inimaginables, telles que :

- l'accès instantané à toute la connaissance de l'humanité depuis n'importe quel endroit de la planète, ce qui est par exemple utile lorsque vous vous interrogez sur la fréquence d'émission du phare de Punta Panul ;

- la rencontre de personnes plus intéressantes les unes que les autres, ce qui engendre très souvent la confrontation d'idées et ipso facto un enrichissement culturel, politique, social, historique, etc ;

- le contact quotidien avec des connaissances des quatre coins du globe, ce qui s'avère important lors d'une relation à distance ou lorsqu'un membre de votre famille décide de s'installer à Santa Clara, voire plus loin encore en Meurthe-et-Moselle ;

- la propagation quasiment instantanée des informations les plus diverses, allant de la naissance des trois chatons angora de Caroline, petite fille de 8 ans vivant en Corrèze, jusqu'à la déclaration d'indépendance du Soudan du Sud le 9 juillet 2011 ;

En bref, plus rapide que Cher Ami, le pigeon voyageur héros de la Première Guerre mondiale, plus rapide encore qu'un hélicoptère Sikorsky VS300, Internet à travers ses interfaces de transmission Wi-Fi, à travers les réseaux câblés sillonnant notre planète, traversant même les océans, ce réseau informatique mondial est une ressource incroyablement précieuse. Rendu possible grâce aux avancées scientifiques et en évolution constante, notamment en ce moment avec l'intelligence artificielle ou le développement de la technologie quantique en cours, Internet est une technologie dont beaucoup d'entre nous ne sauraient se passer aujourd'hui.

Or d'après l'écrivain français du XVIII^{ème} siècle Louis de Bonald, « l'homme n'existe que par la société et la société ne le forme que pour elle. »

Ainsi il semblerait que l'une des caractéristiques déterminantes de l'être humain soit que celui-ci ne peut vivre sans une société, c'est-à-dire sans la compagnie de ses semblables.

L'une des facettes de ce bijou de technologie qu'est Internet, repose donc sur cette caractéristique, celle du besoin de contact social. Vous l'aurez compris, nous parlons ici des réseaux sociaux.

En effet, ces plateformes ont été créées avec le dessein d'apporter une socialisation à leurs utilisateurs. De fait, personne ne peut dire qu'il n'a pas été transformé, changé d'une certaine façon par une rencontre faite sur

l'Internet, ou bien une amitié épanouie grâce à la communication régulière sur des plateformes de « chat » en ligne.

Malheureusement il existe un « mais ». Et un « mais » de taille si nous osons dire. En effet, c'est un euphémisme de déplorer que tout n'est pas tout rose sur les réseaux sociaux. Et lorsque nous écrivons cela, nous ne faisons pas allusion aux internautes qui se croient amusants lorsqu'ils tentent systématiquement de faire cliquer leurs amis sur le lien du clip vidéo-musical de Rick Astley ayant pour titre « Never Gonna Give You Up » pour les piéger en leur faisant croire qu'il s'agit de quelque chose d'autre, non c'est bien pire que cela. Nous parlons bel et bien de félonies qui dans le cadre pénal ne sont rien d'autre que des délits, allant parfois jusqu'à des crimes.

Dans une démarche scientifique où notre unique but est d'être factuels et objectifs – et non pas de promouvoir notre projet – nous laisserons parler les chiffres.

Le premier problème est bien souvent la haine qu'envoient certains internautes à des communautés ou des personnes. Cela a un nom : un tel acharnement représente tout simplement du cyberharcèlement. De nombreux internautes pensent rester impunis lorsqu'ils envoient de la haine à une personne depuis leur appareil numérique. Bien souvent cachés derrière des pseudonymes, les harceleurs crachent leur venin, et espèrent trouver par cette voie un but à leur misérable existence. Si de leur point de vue un tel comportement peut paraître anodin, les conséquences du cyberharcèlement sont dramatiques, pouvant aller jusqu'à la dépression, voire dans des cas extrêmes le suicide des victimes. Cela est d'autant plus tragique lorsque les personnes visées sont des adolescents.

En outre, en faisant appel à des mécanismes psychologiques créant à la fois le besoin et le comblant, les réseaux sociaux fidélisent leurs utilisateurs en les rendant addicts. Bilan de cela: une cyberdépendance, phénomène de société qui inquiète les psychologues, des milliers d'heures perdues chaque année pour des millions de jeunes.

Arnaque au sentiment, fraude en utilisant les cryptomonnaies, sextorsion, usurpation d'identité, etc. Les arnaques sur les réseaux sociaux ont été multipliées par 18 en seulement cinq ans selon une étude. Il semblerait que les plateformes soient impuissantes face à ce fléau, ou que du moins elles le laissent se propager.

Et enfin, bien pire encore. Un pervers caché derrière le profil d'Emilia, 16 ans, qui donne rendez-vous à la petite Zoé, 12 ans, car elle souhaite rencontrer "dans le monde réel" sa nouvelle amie rencontrée grâce à Internet. La suite, vous la connaissez malheureusement. Zoé n'est pas seule. Tous les jours,

des milliers de prédateurs sont présents sur les réseaux sociaux. Ils cherchent la moindre occasion de s'en prendre à un enfant. Là encore, malgré des efforts de la part des plateformes sociales, il semblerait que celles-ci soient relativement impuissantes.

Forts de ce constat, déçus, dégoûtés et tristes à la fois de tant de malveillance, mais pas pour autant résignés face à celle-ci – bien au contraire – nous, Arthur Barbera et Gaëtan Cathelain, avons pris la décision mûrement réfléchie de faire notre part, d'apporter notre modeste contribution avec une tentative de faire du monde – du moins du monde « virtuel » du Web – un endroit où il fait un peu plus bon vivre.

La conception de A à Z d'une telle plateforme d'échanges en ligne représentait évidemment pour deux lycéens tels que nous un projet ambitieux, mais comme le disait Napoléon « l'art d'être tantôt très ambitieux et tantôt très prudent est l'art de réussir. » C'est pourquoi nous avons choisi de ne pas reculer devant l'ampleur de la tâche, d'être ambitieux, car demain ne sera pas trop tard pour être prudents.

Alors nous avons agi. Dans un élan de courage, peut-être aussi de folie, mais de folie saine et inhibitrice, nous avons créé PANDAMONIUM.

Notre but étant bien évidemment de trouver des solutions à ces problèmes qui gangrènent les réseaux sociaux, nous avons mis au point la théorie suivante, notamment comme suit :

- En trouvant des solutions efficaces pour bannir les utilisateurs et supprimer les posts malveillants ;
- En n'incluant pas à notre application l'"infinite scroll", mécanisme qui rend addict les utilisateurs.

Le nom du réseau social s'appuie sur le terme de pandémonium, contraste parfait avec la définition de ce terme représentant un endroit désorganisé que nous appelons Internet, et, en s'appuyant sur un style de forêts japonaises respirant paix et calme, nous avons fait le choix de construire notre réseau social autour du thème du panda roux, vivant dans ces mêmes zones.

Ainsi, les utilisateurs peuvent se joindre sur le même *bambou*, un serveur de messagerie commun à plusieurs utilisateurs. Chaque bambou contient donc une branche sur laquelle ils peuvent se balader, comme pour changer de pièce, de salon, et ainsi naviguer entre plusieurs discussions. Lesdites discussions seront modérées par des modérateurs très actifs et assidus, ne laissant passer aucun des risques évoqués plus au-dessus.

CHAPITRE 2 - LA RÉPARTITION DES TÂCHES

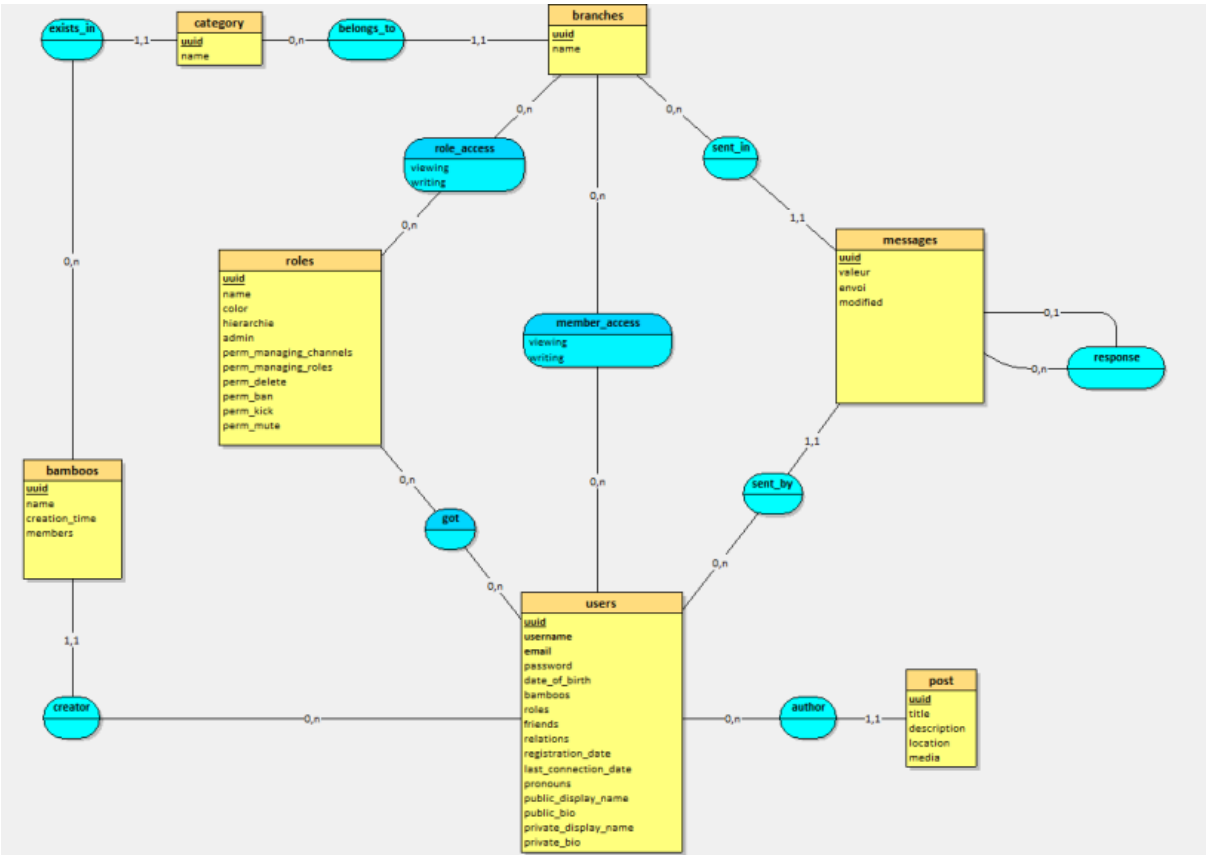
FAIT PAR ARTHUR :

- Communication client-serveur (serveur de messagerie) (**app.py**) ;
- Inscription/connexion de l'utilisateur (**auth.py**) ;
- Moitié du frontend, non délimité ;
- Modules :
    - **database** : Module servant à mieux gérer et implémenter les interactions entre le code et la base de données. Sont présents :
        - Classes représentant tables et colonnes, respectivement **Entity** et **Column** ;
        - Fonctions relatives aux dites classes (à voir sur le code) ;
        - Fonctions permettant d'ouvrir, obtenir et fermer la connexion à la base de données, en s'appuyant sur le module Flask.
    - **security** : Module dont le but est principalement de gérer des entrées utilisateurs de manière sécurisée et de garantir une bonne manipulation de plusieurs types de données. Sont présents :
        - Fonctions relatives à la mise en cache de messages d'erreur à afficher à l'utilisateur ;
        - Fonctions dédiées à la manipulation de mots de passe, de dates, d'UUID, et une dernière fonction pour des données plus générales.

FAIT PAR GAËTAN :

- Mise en place de la base de données (**Schéma relationnel & SQL**) ;
- Implémentation de classes pour communiquer avec celle-ci : **Bamboo**, **Branch** ;
- Programmation du blueprint **bamboo** ;
- Mise en place des formulaires de *création de bambou* et *de branche* à l'intérieur de ceux-ci ;
- Participation au développement de différents templates (fichiers HTML) et plus globalement à la moitié du frontend.

CHAPITRE 3 - LE SCHÉMA DE LA BASE DE DONNÉES


exists_in
1,1
category
uuid
name
0,n
belongs_to
1,1
branches
uuid
name
0,n
role_access
viewing
writing
0,n
0,n
sent_in
1,1
messages
uuid
valeur
envoi
modified
0,1
0,n
response
0,n
roles
uuid
name
color
hierarchie
admin
perm_managing_channels
perm_managing_roles
perm_delete
perm_ban
perm_kick
perm_mute
0,n
member_access
viewing
writing
0,n
bamboos
uuid
name
creation_time
members
0,n
got
0,n
1,1
sent_by
0,n
users
uuid
username
email
password
date_of_birth
bamboos
roles
friends
relations
registration_date
last_connection_date
pronouns
public_display_name
public_bio
private_display_name
private_bio
1,1
creator
0,n
0,n
author
1,1
post
uuid
title
description
location
media

CHAPITRE 4 - ANALYSE D’UNE CLASSE PAR ARTHUR

La classe Entity représente l'instance d'une table quelconque de la base de données, ayant pour clé primaire un UUID. Un UUID est un "identifiant universel unique", ou pour faire simple, une chaîne de caractères construite en se basant grandement sur le timestamp de sa création et respectant toujours le même schéma, le tout encodé en hexadécimal. Ainsi, de nombreuses tables sont représentées par des classes Entity dans le code : **User**, **Bamboo**, **Message** et **Role**. Elle comporte tous les attributs ainsi que toutes les méthodes relatives à l'insertion et à la mise à jour de données dans la base de données.

```
class Entity(abc.ABC):

    """Classe représentant une table de la base de données dont les instances ont
besoin d'être différenciée des autres

    par un UUID."""



    def __init__(self, name: str, uuid: str | None, **columns):

        """Constructeur de la classe.



        :param name: Nom de la table.

        :param uuid: UUID (clé primaire) de la première colonne.

        :param columns: Noms des colonnes de la table, associés à leur valeur ou à
une paire valeur-contrainte (sous

            forme de tuple)."""

        self.name = name

        self.valid = True

        self.__columns = {'uuid': Column('uuid', uuid if uuid is not None else
str(uuid4()), 0)}



        for index, (name, column) in enumerate(columns.items(), 1):

            if isinstance(column, tuple):

                self.__columns[name] = Column(name, column[0], index, column[1])

            else:

                self.__columns[name] = Column(name, column, index)



            if not self.__columns[name].valid:
```

```
                self.valid = False

                break
```

Premier point important : la classe est une fille de ABC, un module Python, dont la signification est simplement “Abstract Base Class”. Son utilité est diverse, mais il sera notamment ici question d’une implémentation polymorphique dans la structure du code, avec des méthodes abstraites. Comme cette classe a pour but d’être implémentée par de futures classes filles dont les méthodes seront radicalement changées, il ne ferait pas sens de pouvoir l’instancier librement dans un code Python, puisque toutes les méthodes abstraites de cette même classe n’ont pas de comportement prédéfini.

Nous remarquons donc, dans le constructeur de cette classe, que l’on attend un nom de table ainsi que des colonnes dont chaque clé est un nom de colonne et chaque valeur une instance de Column, représentant la colonne d’une table (nous n’entrerons ici pas dans les détails de cette classe). Ces deux arguments doivent être communiqués à cette classe par les classes filles. UUID, quant à lui, devra être répété dans les classes filles car il sera entré, à l’avenir, par les utilisateurs des classes filles, et non par les classes filles elles-mêmes.

Comme on peut le voir, ses attributs sont donc **name** (le nom de la table), **valid** (un état valide ou invalide de la classe, varie selon les données passées aux colonnes de la table, données qui peuvent être détectées comme valides ou invalides) et **__columns** (stockant toutes les colonnes de la table, attribut rendu privé car il est encapsulé par des méthodes que nous verrons plus tard).

Pour initialiser ce dernier attribut, une boucle for est utilisée. Chaque valeur est soit la valeur de la colonne elle-même, soit une paire (sous forme de tuple) de valeur et de sa contrainte, dont l’objectif sera de gérer la validité des données entrées par l’utilisateur et qui influera sur l’attribut **valid** précédemment évoqué. Dans ces deux cas, les valeurs feront l’objet d’une instanciation de Column, stockée dans **__columns**. Cette propriété pourra être accédée de plusieurs manières, dont la manière brute que voilà :

```
@property

def columns(self):

   return self.__columns
```

Cette méthode ne contient pas de docstring car, lors de son appel, elle ne se comportera pas comme une méthode mais comme on voudrait le faire avec un attribut, et cela se fait grâce au décorateur **@property**, qui transforme ce getter en attribut. Cela ne doit être utilisé qu’en cas d’itération à travers les attributs de la classe dans un but de lecture uniquement, en évitant la

modification de données. On a aussi une méthode plus directe qui permet d'accéder directement à la valeur d'une colonne, que voilà :

```
def get_column(self, name: str) -> tp.Any | None:

   """Obtenir la valeur d'une colonne à partir de son nom.



   :param name: Nom de la colonne.



    :return La valeur de la colonne portant le nom donné en argument, ou None si
elle n'existe pas."""

   return self.__columns[name].value if name in self.__columns else None
```

Comme on le remarque, si le nom donné en arguments est une colonne qui n'existe pas dans la classe, on ne renvoie rien (None).

Une autre méthode permet à son utilisateur de remplacer directement la valeur de la colonne, sans avoir à recréer d'instance de Column et sans avoir à gérer la validité de la valeur donnée :

```
def set_column(self, name: str, value):

   """Écrase la valeur de la colonne portant le nom donné en argument.



   :param name: Nom de la colonne.

   :param value: Valeur de la colonne."""

   if name in self.__columns:

       self.__columns[name].value = value



       if not self.__columns[name].valid:

           self.valid = False
```

Ces trois méthodes ont pour objectif d'*encapsuler* l'attribut **__columns**. Sans trop entrer dans les détails, l'encapsulation est un concept de la programmation orientée objet. Elle permet de protéger les attributs de sa classe à l'aide de méthodes dans l'objectif de faire des vérifications ou des actions supplémentaires, contrairement à ce que feraient de simples getters et setters. En l'occurrence, la méthode **set_column()** permet non seulement d'éviter une KeyError (le fait de référencer une clé qui n'existe pas dans un dictionnaire qu'est l'attribut **__columns**), mais aussi d'éviter de donner n'importe quelle valeur (initialement potentiellement invalide) à la colonne en question, de gérer, si la valeur ne peut être transmise à la colonne par son invalidité, l'attribut **valid** de la classe tout en conservant l'instance de Column

donnée lors de l'initialisation de la classe. Ces méthodes sont donc des incontournables, afin d'éviter toute nuisance ou erreur au fonctionnement de la classe.

S'en suivent deux méthodes aux décorateurs intéressants :

```
@classmethod
@abc.abstractmethod
def instant(cls, *args):
    """Constructeur créant à la fois une nouvelle instance de la classe actuelle tout en la créant en base de
   données."""
   pass


@classmethod
@abc.abstractmethod
def fetch_by(cls, *args):
      """Constructeur créant une instance de la classe actuelle à partir d'informations qui seront récupérées en base
   de données via une requête SQL de type SELECT."""
   pass
```

Premièrement le décorateur **@classmethod**. Pour faire simple, c'est exactement la même chose que la méthode **__init__()** trouvable dans n'importe quelle classe Python ayant un constructeur propre à elle. Or, les méthodes possédant ce décorateur sont accessibles d'office sans avoir besoin de passer par une instance de la classe à laquelle on se réfère. En l'occurrence, je pourrais faire **Entity.instant(arg1, arg2, arg3, …)** (toutefois je ne peux pas le faire car le deuxième décorateur m'en empêche, explication juste en-dessous). Le paramètre **cls** visible sur la signature de cette méthode est invisible pour l'utilisateur externe à la classe et n'a d'intérêt que dans le corps de la méthode décorée. Son unique intérêt est d'appeler le constructeur (**__init__()**) de la classe courante.

Deuxièmement, le décorateur **@abc.abstractmethod**. De part son nom et sa provenance du package **abc** au même titre que la classe **ABC**, on comprend que son but est de définir la méthode qui s'ensuit comme étant "abstraite". Mais qu'est-ce qu'une méthode abstraite ? C'est une méthode qui n'a aucun corps et qui ne doit pas en posséder, car cela n'aurait aucun sens, comme expliqué au début de la présentation de cette classe. Elle devra donc obligatoirement être redéfinie lorsqu'une classe fille sera créée, auquel cas elle ne pourrait fonctionner.

Pour en revenir à l'utilité de ces "constructeurs nommés et abstraits", **instant()** permet non seulement d'obtenir une instance de la classe, mais en plus de l'insérer sur place en base de données, c'est pourquoi cette méthode porte ce nom (instantané). **fetch_by()**, quant à elle, a un intérêt "inverse" ; son but est d'initialiser une instance de la classe à partir d'une valeur de la base de données.

On peut remarquer que ces deux méthodes ont l'argument ***args**. Sans entrer dans les détails une fois de plus, cela permet aux classes filles qui devront redéfinir cette méthode de changer le nombre d'arguments de la méthode en question.

Une fois de plus, on a une autre méthode abstraite mais qui n'est, cette fois ci, pas un constructeur nommé :

```
@abc.abstractmethod

def _update(self, **values):

    """Méthode permettant de mettre à jour certaines valeurs de l'instance de la
table actuelle.

   Cette méthode ne doit être utilisée que par les classes filles, qui doivent la
redéfinir.



   :param values: Nouvelles valeurs à attribuer aux colonnes de la table."""

   pass
```

Cette méthode a pour but de mettre à jour des données dans la base de données. L'argument ****values** sera, comme l'attribut **__columns** de la classe, identifié comme un dictionnaire contenant, pour chaque clé, le nom de la colonne visée, et en valeur la nouvelle valeur que devra prendre la colonne. Le préfixe **_** est présent afin de dire que cette méthode ne doit être vue que par les classes filles, et cette visibilité a été choisie car des noms de colonnes donnés pourraient être erronés, c'est pourquoi une méthode publique a fait son apparition dans le but de n'appeler la méthode **_update** que dans le cas où toutes les colonnes demandées existent :

```
def update(self, **values):

    """Méthode permettant de mettre à jour certaines valeurs de l'instance de la
table actuelle.



   :param values: Paires de clés-valeurs à assigner aux colonnes, où la clé est le
nom de la colonne attachée à

       sa valeur."""

   for key, value in values.items():
```

```
        if key not in self.__columns:

                raise ValueError(f"La colonne '{key}' n'existe pas dans la table
{self.name}.")



    self._update(**values)
```

Ainsi, si une colonne demandée n’existe pas, une exception de type ValueError sera lancée (car cela vient donc du développeur et non de l’utilisateur).

CHAPITRE 5 - ANALYSE D'UNE CLASSE PAR GAËTAN

La classe Bamboo permet l'implémentation en langage objet des données d'un enregistrement de la table bamboos dans la base de données, ainsi que sa gestion et sa modification.

```
class Bamboo:

    """Classe représentant un serveur unique du réseau social.

    Par "bambou", nous parlons d'un endroit virtuel créé sur notre réseau social pour discuter de façon communautaire.

    Différentes "branches" de discussion peuvent être créése, des rôles et permissions peuvent être

    attribués aux différents membres par le créateur ou les administrateurs du bambou."""


    def __init__(

            self,

            bamboo_uuid: str = None,

            name: str = None

    ):

        """Ctor d'un bambou. Instancie le bambou à partir de la base de données si son uuid est donné en argument.

        Sinon, crée le bambou dans la base de données si le nom est indiqué en argument.

        Paramètres :

            bamboo_uuid STR

                L'UUID du bambou existant à aller chercher dans la base de données et à instancier.

            name STR

                Le nom du bambou à créer et à instancier
```

```
        Les différents attributs donnés à l'instance sont
: nom, date de création, uuid du créateur et membres
(sous la

       forme d'une liste)."""


       if bamboo_uuid is not None:

           self.uuid = bamboo_uuid

           db = get_db()

           with db.cursor() as curs:

               curs.execute(

                                   'SELECT name, creation_date,
owner_uuid, members FROM bamboos WHERE uuid = %s',

                   [self.uuid]

               )

               bamboo = curs.fetchone()

               self.name = bamboo[0]

               self.creation_time = bamboo[1]

               self.creator = bamboo[2]

               self.members = uuid_split(bamboo[3])
```

Le constructeur de cette fonction prend deux possibilités : la première où elle est construite à partir de l'uuid d'un bambou déjà existant dans la base.

Dans ce cas les données récupérées, correspondant aux données des attributs name, creation_time, creator et members sont récupérées dans la base de données et indiquées dans les attributs du même nom de l'objet créé.

```
elif name is not None:

   self.uuid = str(uuid4())

   self.name = name

   self.creator = fk.g.user

   self.creation_time = date_to_string(datetime.now())
```

```
    db = get_db()

    db.cursor().execute(

            'INSERT INTO bamboos(uuid, name, creation_date,
owner_uuid, members) VALUES (%s, %s, %s, %s, %s)',

                (self.uuid, self.name, self.creation_time,
self.creator.get_column('uuid').value,

             self.creator.get_column('uuid').value)

    )
```

L'autre possibilité est de créer un serveur inexistant dans la base à partir du nom demandé par l'utilisateur.

Les données sont alors attribuées automatiquement :

- uuid est un nouvel uuid généré automatiquement ;
- creator prend la valeur de l'username de l'utilisateur connecté au moment de la création du bambou ;
- creation_time devient la chaîne de caractères de la date et de l'heure du moment de création.

Ces valeurs sont insérées dans la base de données, dans la table bamboos, avec le nom name entré en argument.

```
def update(

        self,

        name: str,

):

    """Méthode permettant de modifier les informations """


    if self.name != name:

        self.name = name


        db = get_db()
```

```
        db.cursor().execute(
            'UPDATE bamboos SET name = %s WHERE id = %s',
            (self.name, self.uuid)
        )
```

La méthode update quant à elle, permet de modifier tant dans l'instance objet que dans la base de données, le nom du bambou.

REMARQUE : Seul le nom est modifiable car les autres attributs sont remplis automatiquement de façon arbitraire.

```
def get_branches(self):
    """Méthode qui renvoie une liste contenant les uuid de
toutes les branches faisant partie de l'instance."""
    db = get_db()
    with db.cursor() as curs:
        curs.execute(
             'SELECT uuid FROM branches WHERE parent_bamboo
= %s',
            [self.uuid]
        )
        result = curs.fetchall()

    return [Branch(uuid[0]) for uuid in result]
```

La méthode suivante permet de récupérer toutes les branches dépendant d'un bambou.

Pour ce faire, nous faisons une requête de type SELECT dans la base de données pour récupérer dans une liste tous les UUIDs des branches concernées.

La valeur renvoyée est une liste contenant les implémentations en objet de chacune des branches.