Check-List Manager

Dossier de conception technique

Version 1.0

Auteur
LE MAGOROU Jean-Martial

Nothing-consulting
www.nothing.net

55 rue du Faubourg St-Honoré – 75 008 Paris – jmlm74@gmail.com

SARL au capital de 000,00 € enregistrée au RCS de Paris – SIREN 999 999 999 – Code APE : 6202A

Table des matières

1 - Versions

Auteur	Date	Description	Version
JMLM	29/09/20	Création	V0.1
JMLM	15/10/20	Domaine fonctionnel	V.01
JMLM	04/11/20	Architecture Technique et déploiement	V.01
JMLM	05/11/20	Architecture Logicielle	V.01

2 - Introduction

2.1 - Objet du document

Le présent document constitue le volet « conception technique » de la réponse au P13 du parcours Développeur d'Applications Python d'OPENCLASSROOMS.

Il a pour but de présenter les différents acteurs et fonctionnalités afin de :

- Modéliser les objets du domaine fonctionnel et ainsi définir le modèle physique de données qui permettra de concevoir la base de données de ce qui sera le futur logiciel Chek-List Manager.
- Définir les composants internes et externes afin de définir leurs interactions avec le logiciel.
- Décrire le déploiement des composants
- Décrire l'architecture logicielle de l'application ainsi que les modules nécessaires à sa bonne exécution.

Les éléments du présent dossiers découlent

- de l'expression du besoin
- de l'analyse complète du besoin suite à mon expérience personnelle récente.

Dans la suite du document, nous ferons références aux sociétés et non plus à l'utilisateur. Un utilisateur particulier n'est rien d'autres qu'une société avec un unique utilisateur.

2.2 - Références

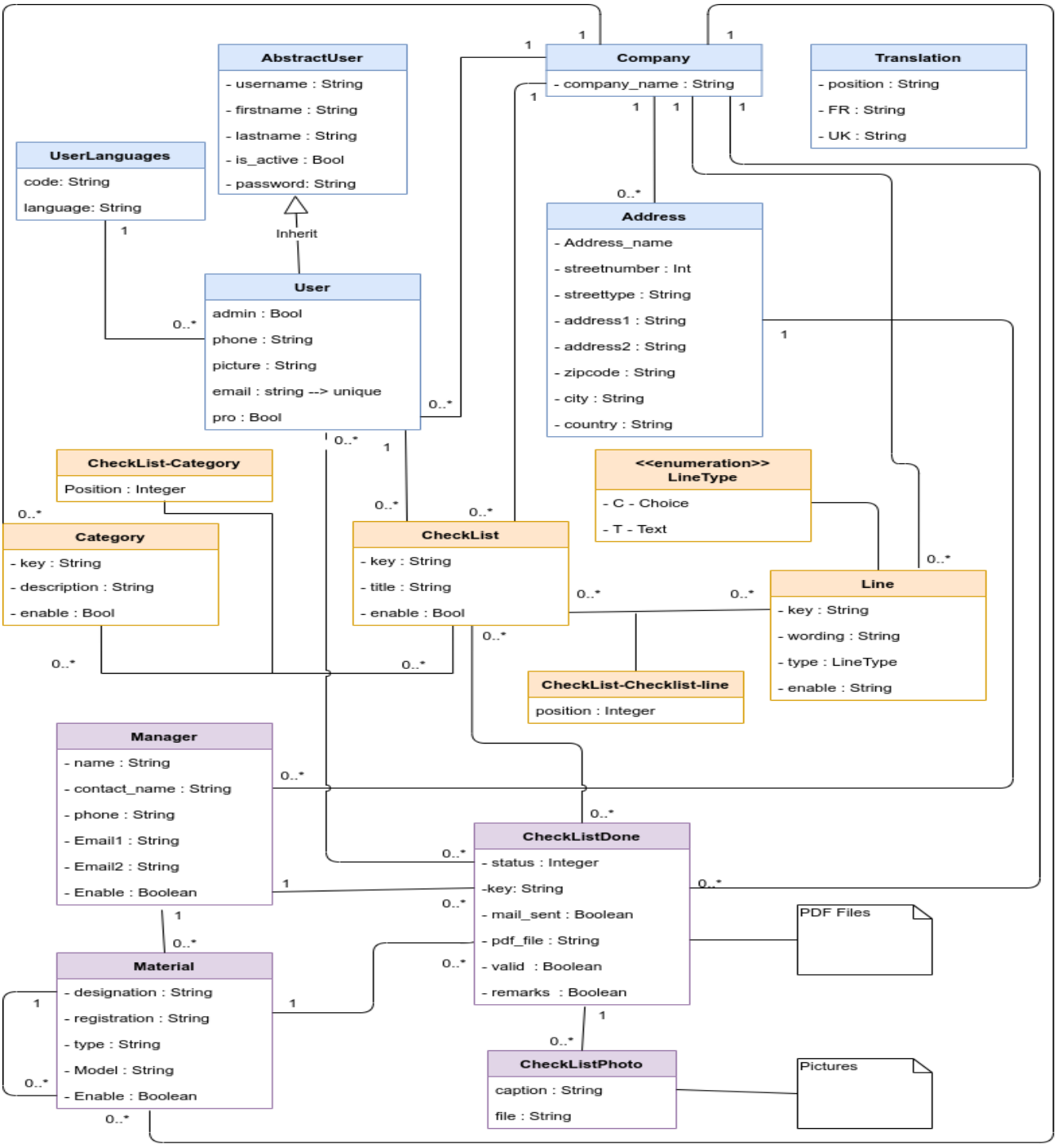
Pour de plus amples informations, se référer également aux éléments suivants :

1. **P13 – Dossier Conception Fonctionnelle** : Dossier de conception fonctionnelle de l'application.
2. **P13 – Dossier Exploitation :** Dossier d'exploitation de l'application.

3 - Le domaine fonctionnel

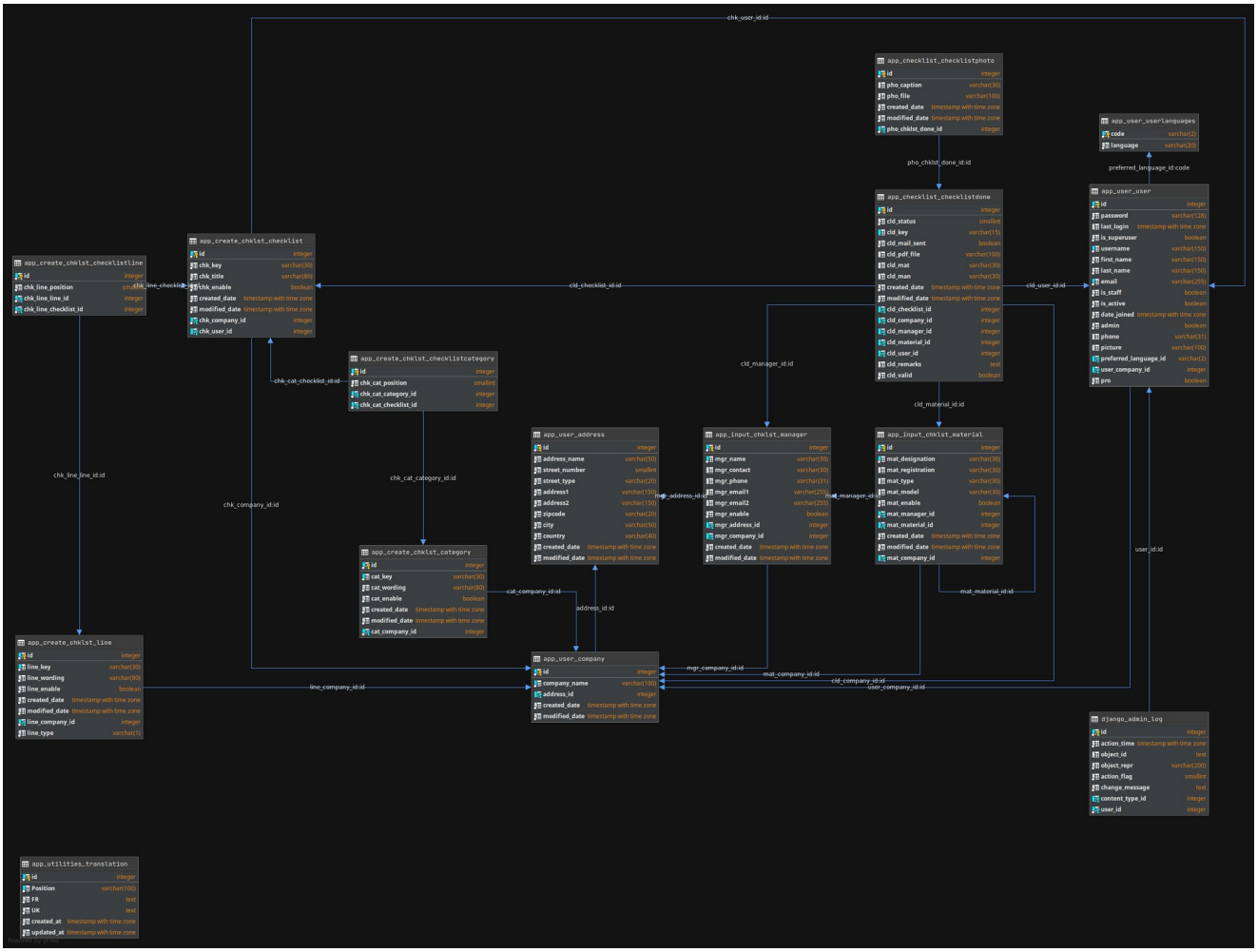
3.1 - Référentiel

Le domaine fonctionnel permet d'établir les différentes classes interférant dans le S. I. ainsi que les relations qui les lient. Le premier schéma est appelé le diagramme de classes. Il servira de socle à la création du Modèle physique de données et donc à la création de la base de données.



Le diagramme de classes UML ci-dessus fournit une représentation précise des informations manipulées et permet ainsi d'avoir une vision globale du système et de l'organisation des données.

Du diagramme de classes découle un second diagramme plus détaillé qui est la modélisation exacte de la base de données : le Modèle physique de données.



Nous décrirons ensuite chaque table/classe, les différentes colonnes/attributs qui les composent ainsi que les liens/associations qui les relient entre-elles.

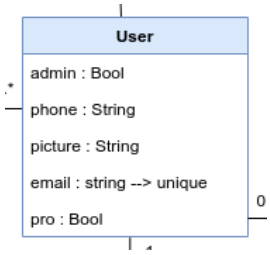
3.1.1 - Règles de gestion

Nous allons voir que les classes et les tables sont étroitement liées, car les secondes émanent des premières. Dans la section suivante nous allons décrire les classes/tables ainsi que les liens qui les relient entre-elles. Les classes ont été classées en 3 groupes : Ces 3 groupes correspondent à peu de chose près aux 3 grandes parties de l'application. La partie « Utilisateurs » qui va concerner les utilisateurs et les sociétés. La partie « Création » qui va plutôt s'occuper de la partie création d'une check-list. Et enfin, la partie Saisie qui elle comprend tout ce qui concerne la saisie d'une check-list. En effet, nous verrons qu'une check-list saisie

peut être finalement assez éloignée de son « squelette » car apparaît les notions d'exploitant et de matériel ainsi que des photo…
Dans les descriptifs qui vont suivre, quand nous parlerons de cardinalité et pour une meilleure compréhension, nous partirons toujours de la classe/table dont nous faisons la description. Ce sera donc toujours la table qui contiendra la clé étrangère (ou foreign key) notée *nomtable_fk* (ex : address_fk).

3.1.2 - Les classes « Utilisateurs »

3.1.2.1 - Classe/table User



La Classe User décrit l'utilisateur de l'application. Elle hérite de la classe standard AbstractUser et récupère ainsi tous les attributs et méthodes standard d'un utilisateur classique d'une application Django (prénom, nom, mot de passe… pour les attributs et login/logout, is_authenticated… pour les méthodes.

À noter que l'utilisateur est répertorié dans la base par un identifiant unique mais qu'il est tout à fait possible d'avoir 2 utilisateurs avec le même nom-prénom. Une contrainte d'unicité a été rajoutée sur l'e-mail pour conserver la méthode standard fournie par Django pour le reset des mots de passe. D'autres attributs ont eux aussi été ajoutés. Il s'agit de la société (foreign key avec la table company » qui permet de lier un utilisateur à une société. À noter que les utilisateurs qui s'enregistrent directement (les particuliers) ont comme société leur Id dans la table User (ex l'utilisateur n° 195 aura comme société une société qui s'appellera 195. Il a aussi été rajouté un avatar ou une photo de l'utilisateur d'une taille maxi de 100X100 ainsi qu'un numéro de téléphone. Enfin, denier ajout : La langue de l'interface. Un utilisateur peut donc choisir la langue dans laquelle l'interface du site s'affichera. À noter qu'à aucun moment, un utilisateur ne peut choisir sa société. La société lui est, soit attribué d'office à son auto-enregistrement, soit attribué par l'administrateur de site à la création de l'administrateur de la société et ensuite l'administrateur en le créant lui attribuera automatiquement sa société.

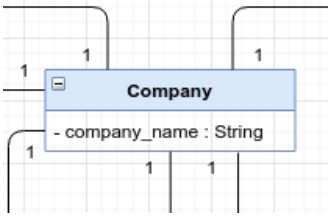
3.1.2.1.1 Relations

Classe : Company – Cardinalité : de 1 à plusieurs
La classe User est reliée à la classe Company en gardant une relation de 1 côté User et plusieurs côté company. En effet, un utilisateur doit avoir une société (au moins son Id User) alors qu'une société peut-avoir plusieurs utilisateurs.

Classe : UserLanguages – Cardinalité : de 1 à plusieurs

Chaque utilisateur a une langue dans laquelle il souhaite voir s'afficher l'application. Cette relation permet donc de définir cette langue. Le choix d'une relation et non pas d'un attribut à la table User s'explique par le fait qu'en cas d'ajout d'une nouvelle langue, il n'y aura pas besoin de modifier la classe User qui est une classe assez sensible.

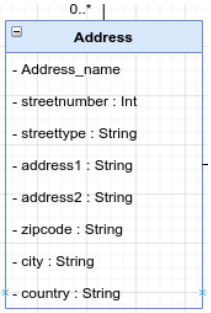
3.1.2.2 - Classe/table Company



Cette classe (qui peut paraître subsidiaire) est en fait la classe la plus référencée. En effet, la plupart des autres classes s'y rapportent directement, car il faut à chaque phase conserver le cloisonnement strict entre les utilisateurs ou sociétés.

Les relations seront traitées au fur et a mesure du document, mais il me paraissait important de souligner l'importance de cette classe.

3.1.2.3 - Classe/table Address



La classe address contient toutes les adresse que ce soit celles d'une société exploitant ou d'une société inscrite sur le site, une adresse reste une adresse. La classe adresse est la seule qui ne soit pas cloisonnée.

3.1.2.3.1 Relations

Classe : Company – Cardinalité : de 0,1 à plusieurs

A une adresse correspond de 0 à plusieurs sociétés et/ou exploitants. En effet, il est possible de :

- Créer une société/exploitant sans adresse (pourquoi pas ?) si l'utilisateur souhaite juste référencer le matériel et lui assigner un exploitant.
- Créer une adresse sans assigner de société dans le but de la réutiliser plus tard.
- D'avoir plusieurs sociétés/exploitant à la même adresse.

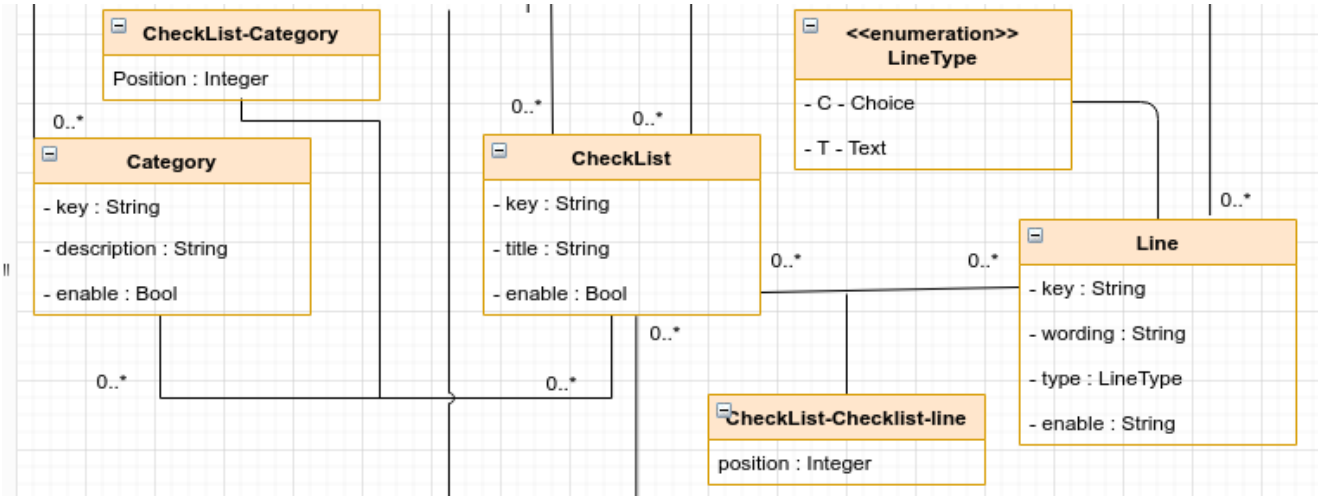
3.1.2.4 - Classes/tables UserLanguages et Translation

L'utilisation de la classe UserLanguage a déjà été expliquée lors de la description de la classe User. Elle va servir à déterminer pour chaque utilisateur sa langue d'affichage du site.

La classe Translation va quant à elle contenir pour chaque mot, chaque bouton, chaque phrase affichés sur le site sa traduction en autant de langues que souhaitées (ici FR et UK). Elle est donc accédée plusieurs fois a chaque affichage de page que ce soit un page « statique » ou non à travers un template tag. En fait, a chaque affichage, on va rechercher un mot cle (position) dans la table et afficher sa traduction (FR ou UK). À noter qu'afin de limiter tout de même les accès à la base, la langue de l'utilisateur est stockée dans une variable de session dès sa connexion et que le langage par défaut est UK.

3.1.3 - Les classes « Création »

3.1.3.1 - Classes/tables CheckList – Category, Line



Les classes « Création » sont les classes utilisées à la conception des checklists, c'est à dire à la création du masque de saisie.

Les classes « Création » se résument en fait à 1 classe centrale (CheckList) reliée par une relation de plusieurs à plusieurs a 2 autres classes : Category et Line à travers leurs positions respectives dans la check-list grâce aux tables intermédiaires CheckList-Category et CheckList-Line.

Exemple : La check-list A est composée de la catégorie C et des lignes l1, l2, l3.
La relation entre CheckList et Category va donc être effectuée au travers d'une table intermédiaire qui va contenir la position de la catégorie dans la check-list (ici 1). Par le même principe, les tables CheckList et Line sont reliées entre-elles par le même principe et ayant respectivement les positions 2, 3 et 4 pour l1, l2 et l3.
Cela permet aussi bien entendu qu'une ligne ou une catégorie soient utilisées par plusieurs

check-lists.

À noter qu'une ligne ou une catégorie n'est pas nécessairement reliée à une check-list. Elle peut bien entendu avoir été créée en attente de…

3.1.3.1.1 Relations

Classe: Company - cardinalité : de 0,1 à plusieurs

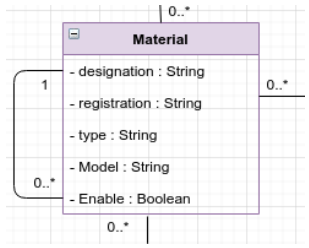
Les 3 tables précédemment citées sont reliées a la table Company. Cette relation permet de conserver le cloisonnement entre les sociétés. Il va de soi qu'une check-list, ligne ou catégorie appartient à une société et une seule et qu'une société en a bien entendu un grand nombre.

Cette relation permet aussi d'avoir une contraint d'unicité avec la société : Il est possible d'avoir plusieurs check-lists du même nom tant que ce n'est pas sur la même société (cohérence du cloisonnement).

3.1.4 - Les classes « Saisie »

Les classes « Saisie » sont les classes utilisées lors de la saisie de la Check-list, c'est-à-dire lors de l'utilisation du masque précédemment créé. Les 2 classes Manager et Material ne sont pas utilisées par les utilisateurs non-pro (qui se sont auto-enregistrés) sur le site.

3.1.4.1 - classe/table Material



La table Material va donc contenir les matériels sur lesquels vont être effectuées les check-lists. Cela n'est pas obligatoire car une check-list peut aussi être une simple liste de course ou autre et ne s'applique donc pas à un matériel.

3.1.4.1.1 Relations

Classe : Material – cardinalité : de 0,1 à plusieurs –> Relation sur elle même

Une notion à prendre en compte est la notion de matériel primaire et secondaire. Un matériel peut très bien être rattaché à un autre matériel : Prenons par exemple une voiture. Vous pouvez avoir une check-list : « Vidange » qui va être directement rattaché au véhicule, mais vous pouvez avoir une check-list qui va concerner les moteurs. Vous avez donc un matériel (le moteur) qui sera relié au véhicule. Le véhicule sera donc le matériel primaire et le moteur le matériel secondaire. Il est donc possible d'avoir plusieurs matériels secondaires sur un matériel primaire.

Classe : Manager – Cardinalité : de 0,1 à plusieurs

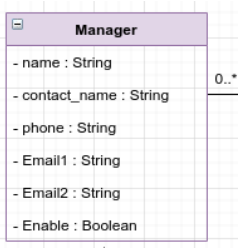
Un matériel peut avoir un exploitant. Cet exploitant permettra d'envoyer un mail à la fin de la saisie s'il est renseigné. Cela permet aussi une recherche et un classement plus aisé des

matériels mais aussi des check-lists saisies.

Classe : Company – Cardinalité : de 0,1 à plusieurs

Il s'agit de la société ayant créé l'exploitant. Cela va donc servir au cloisonnement ainsi que pour une contrainte d'unicité (Company-name).

3.1.4.2 - Classe/table Manager



La classe manager va contenir le contact exploitant d'un matériel. Tout comme la classe Material, la classe Manager n'est pas utilisée par les utilisateurs non-pro. À noter qu'un exploitant va la plupart du temps être une société, une personne pouvant être spécifiée pat l'attribut contact_name

3.1.4.2.1 Relations

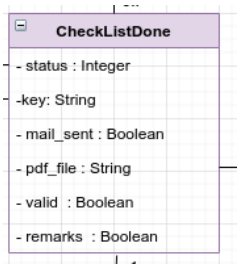
Classe : Address – Cardinalité : de 0,1 à plusieurs

Il est possible de créer un exploitant avec ou sans adresse. Et bien entendu, une adresse peut être l'adresse de plusieurs exploitants.

Classe : Company - Cardinalité : de 0,1 à plusieurs

Il ne s'agit pas ici de la société de l'exploitant (cf ci-dessus) mais de la société ayant créé l'exploitant. Cela va donc servir au cloisonnement ainsi que pour une contrainte d'unicité (Company-name).

3.1.4.3 - Classe/table CheckListDone



Cette classe contient toutes les check-lists terminées. Il ne s'agit pas ici de sauvegarder toutes les rubriques/réponses… mais de sauvegarder le fichier PDF issue de la saisie, le statut (valide ou non –> permet de coloriser la liste sur la page d'entrée), les remarques globales et le statut (0 : encours – 1 Terminé).

3.1.4.3.1 Relations

Classe : Material - Cardinalité : de 0,1 à plusieurs

Il est évident qu'une check-list peut pointer sur un matériel mais qu'un matériel peut être concerné par plusieurs check-lists.

Classe : Manager – Cardinalité : de 0,1 à plusieurs

Le principe est ici le même que pour Material. À noter qu'il est tout à fait envisageable de saisir une check-list sans matériel mais avec un exploitant (Vérification des consignes de sécurité dans un bâtiment par exemple). Cela empêcherait par exemple de remonter du manager en passant par le matériel et vice-versa.

Classe : Company – Cardinalité : de 0,1 à plusieurs

Comme pour toutes les autres classes reliées à Company, dans un souci de cloisonnement et d'unicité.

Classe : CheckList – Cardinalité : de plusieurs à plusieurs

Cette relation permet de savoir quel modèle de check-list a été utilisé pour telle ou telle checklist. Elle permet aussi de créer le fichier PDF qui sera, lui, conservé.

3.1.5 - Nota Bene

3.1.5.1 - Le mot de passe

Concernant le mot de passe, je voudrais insister sur le fait qu'il est crypté directement par une librairie python (crypt en mode SHA256) et que si sa longueur en texte est de maximum 30 caractères alphanumériques, le mot de passe crypté fait plus de 100 caractères de long. De plus, il est impossible de récupérer un mot de passe perdu, il faudra en recréer un !La procédure pour le recréer est la procédure standard fournie par Django.

3.1.5.2 - Les langues

L'ajout d'une langue est relativement simple :
- Créer un code – langage dans la table UserLanguages
- Créer une colonne dans la table Translation dont le nom sera le code précédemment créé.
- Traduire tous les messages dans la nouvelle langue dans la table Translation.
- La nouvelle langue est prête à l'utilisation. Il suffit alors de créer/modifier un utilisateur avec la nouvelle langue.

4 - Architecture Technique

L'application sera une application de type web elle-même basée sur plusieurs briques. Ces briques mises côte à côte forment le serveur d'application. C'est cette application qui communiquera avec les utilisateurs d'un côté et avec les composants externes de l'autre

Les composants externes sont décrits par les diagrammes de composants. Ceux-ci mettent en évidence les dépendances entre eux et les composants internes et décrivent les interfaces nécessaires à leur utilisation.
Ces composants sont au nombre de 3 : OpenStreetMap pour la géolocalisation, OpenWeather pour la prévision météo et MAILGUN pour les envois de mails.

4.1 - L'application WEB

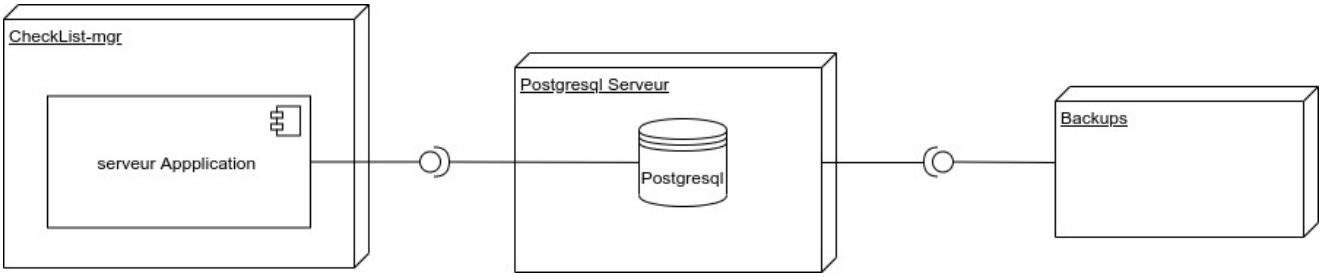
L'application web sera une application web responsive. Cela veut dire que cette application pourra être utilisée sur différents médias (PC, tablette, smartphone). Cette application devra être développée sur une plateforme qui permette une évolution simple à mettre en œuvre tout en restant robuste et résistante à la charge. Toutefois, la complexité de saisie de la partie utilisateur et création des matériels, lignes, Check-lists… fonctionnera sur les écrans de type smartphone mais sera difficilement utilisable. La partie « Saisie des check-lists » sera parfaitement utilisable sur smartphone. L'application dans son intégralité sera complètement utilisable sur les crans de type tablettes et plus grands.
L'application sert les 2 interfaces décrites dans le cahier des charges fonctionnel. Le serveur sera donc le serveur frontal quel que soit le type d'utilisateur et quelle que soit la demande de l'utilisateur.

L'application sera composée des briques logicielles suivantes :
- Système d'Exploitation : Linux DEBIAN 10 (Sortie Juillet 2019)
La distribution Linux Ubuntu existe maintenant depuis plus de 20 ans. Cette distribution est très certainement la plus utilisée dans le monde pour les serveurs. Le support des distributions Debian est de 5 ans
- Langage utilisé : Python 3.8 (sortie Octobre 2019)
Le langage python est depuis quelques années déjà l'un des langages les plus utilisés. Il est reconnu pour sa rapidité tant au niveau de l'exécution et de développement ainsi que pour le grand nombre de bibliothèques externes qui lui offrent un grand nombre de fonctionnalités.
- Framework + ORM : Django 3.1 (sortie Août 2020)
L'utilisation d'un framework tel que Django couplé à des librairies telles que Bootstrap et Jquery donne des applications « full responsive » avec une excellente ergonomie. De plus, l'utilisation d'un framework web permet un gain de temps en termes de développement. Cela permet aussi une organisation du code source standardisée qui est donc facilement maintenable.
- Serveur Web : Gunicorn (sortie Novembre 2019) + Nginx (sortie
Le serveur Gunicorn est un serveur d'application WEB. C'est-à-dire qu'il va exécuter les programmes. La première version est sortie il y a environ 10 ans.
Le serveur NGINX date lui aussi d'environ 10 ans. Son rôle va être de servir les pages WEB. A chaque fois qu'un programme devra être executé, il enverra la requête a Gunicorn qui avec le

moteur Django exécutera la tâche et renverra la réponse à Nginx .

- Serveur de base de données : Postgresql 12 (sortie Juin 2019)
Là encore un « dinosaure » de 1996 et qui a eu le temps de démontrer toutes ses qualités. La donnée est le cœur du système. Il n'est pas concevable de confier ses données à un serveur qui n'a pas toutes les qualités requises : Rapidité, fiabilité, sécurité. Postgresql rassemble toutes ses qualités et dispose aussi d'outils avancés tels que de la réplication en temps réel par exemple.

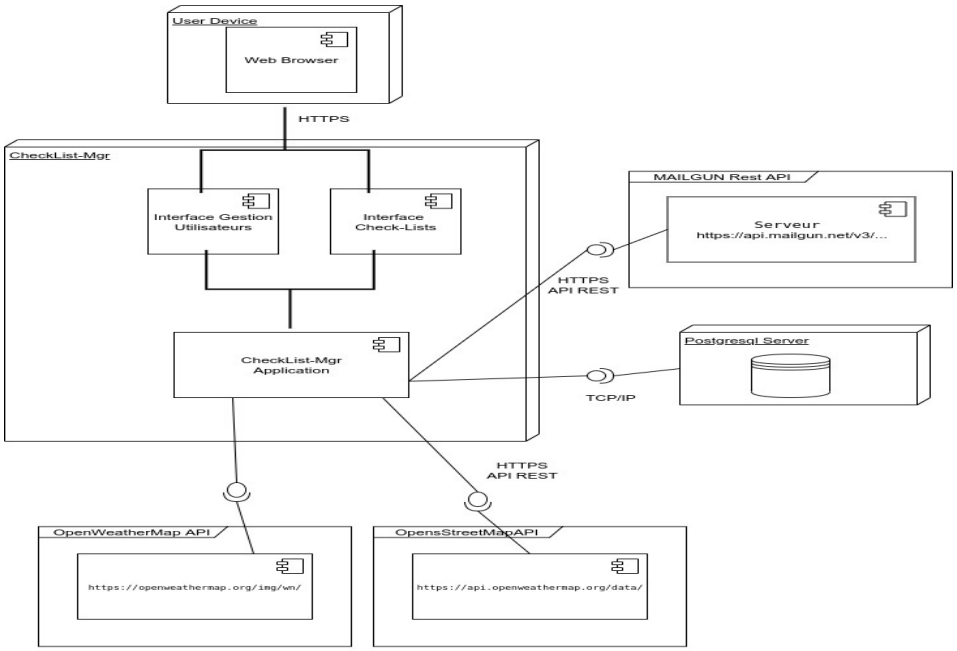
4.2 - La base de données



La base de données sera donc un serveur Postgresql. Le système d'exploitation de ce serveur sera Linux. Les sauvegardes et réorganisations de la base de données seront effectuées de nuit en dehors des heures d'ouverture.

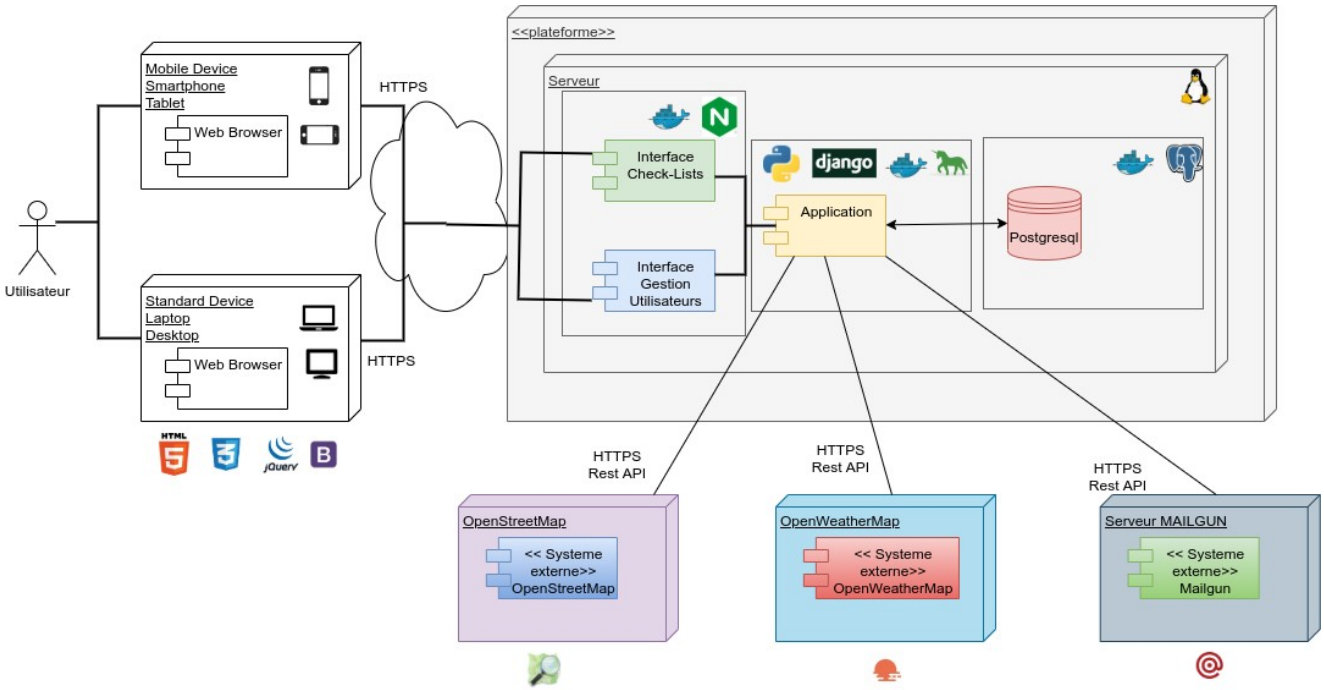
4.3 - Les Composants externes

Le diagramme suivant montre les différents composants externes nécessaires à l'application.



5 - Architecture de Déploiement

Après avoir vu chaque composant, le diagramme de déploiement UML permet aisément d'identifier sur quel matériel physique se place chaque composant ainsi que les différentes connexions qui les relient. Il permet donc de bien comprendre comment sera déployée l'application, et par quel moyen chaque acteur primaire ou secondaire devra y accéder.



On peut donc voir sur le graphique que l'architecture est découpable en 3 grandes parties :
- Les matériels utilisateurs
- La plateforme
- Les acteurs secondaires

5.1 - Les Matériels utilisateurs

L'application fonctionne sur tous les types d'écrans. Par contre, la partie Gestion Utilisateurs et la création des matériels, exploitants... sera difficilement utilisable sur un écran trop petit, car il y a beaucoup de saisie à faire. Par contre, la saisie des check-lists est parfaitement opérationnelle sur tous les types d'écran.

5.2 - La plateforme

L'application sera déployée sur un serveur de l'hebergeur Digital Ocean. Cet hébergeur permet

de disposer d'un monitoring intégré et d'un serveur performant à un tarif très convenable. Comme on peut le voir sur le Schéma d'architecture de déploiement chaque partie sera en fait un conteneur Docker. La gestion de ces conteneurs sera effectuée par Docker-compose.

6 - Architecture logicielle

6.1 - Principes généraux

Les sources et versions du projet sont gérées par **Git**, les dépendances sont-elles gérées par **pip**.

L'application se présentera donc sous la forme d'un projet Django divisé en 6 parties ou applications :
- app_user :Gestion utilisateurs
- app-utilities : Utilitaires (transportables d'un projet à l'autre)
- app_create_chklst: Création Check-lists (Catégories, Lignes, Adresses, Check-lists)
- app_input_chklst : Création Exploitants, Matériels
- app_home : Partie statique du site
- app_chklst : Saisie des check-lists + génération PDF

Ce mode de fonctionnement en applications permet de sortir relativement facilement une application pour la réutiliser ailleurs (Gestion utilisateurs par exemple) mais aussi de rajouter une application (statistiques par exemple) sans avoir à tout refaire.

Chaque application respecte le pattern Model-Vue-Template. Ce modèle est inhérent à Django et est suivi maintenant par une très grande majorité de développements et d'outils.

6.1.1 - Les couches

Comme dit ci-dessus, L'architecture applicative est du type MVT:

- La couche **Model :** Responsable de la représentation des données. Pour schématiser, elle va représenter les données et les fonctions (méthodes) qui leur sont liées.
- La couche **Template** : Responsable de l'affichage des données.
- La couche **Vue :** Responsable de l'interface client c'est en fait la logique du programme qui est géré par cette couche. Elle reçoit des demandes (requêtes) demande si besoin à la couche **model** de lui renvoyer des données et les renvoie une fois traitées à la couche **template** pour qu'elle les affiche correctement. J'ai préféré explique cette couche en dernier, mais on voit bien qu'elle se situe au milieu des 2 autres.

Exemple pour l'application app_create_chklst:
- La couche **Model** va représenter les données : Category, Material, CheckList…
- La couche **Vue** va être la logique : Un utilisateur est sur le site et va sur la page « Matériels ». La couche **Vue** reçoit alors une demande d'affichage de la page. Elle va donc demander à la couche Model les données à afficher. Elle recoit alors ces données brutes, les traite puis les envoie à la couche **Template**

- La couche **Template** va alors afficher les données reçues en respectant les chartes graphiques, les taille d'écran...

6.1.2 - Structure des sources

La structuration des répertoires du projet suit la logique suivante :

- Les répertoires et fichiers ci-dessous ne sont indiqués qu'à titre d'exemple. L'ordre n'est pas respecté et d'autres pourront être créés au fur et à mesure du développement et des besoins.

```
Checklistmgr
├── app_checklist
│   ├── migrations
│   ├── static
│   │   └── app_checklist
│   │       ├── css
│   │       ├── img
│   │       └── js
│   ├── templates
│   │   └── app_checklist
│   └── test
├── app_create_chklst
│   ├── migrations
│   ├── static
│   │   └── app_create_chklst
│   │       ├── css
│   │       ├── img
│   │       └── js
│   ├── templates
│   │   └── app_create_chklst
│   │       ├── dialogboxes
│   │       └── partials
│   └── test
├── app_home
│   ├── migrations
│   │   └── __pycache__
│   ├── __pycache__
│   ├── static
│   │   └── app_home
│   │       ├── css
│   │       ├── img
│   │       └── js
│   ├── templates
│   │   └── app_home
│   │       └── partials
│   └── test
│       └── __pycache__
├── app_input_chklst
│   ├── migrations
│   │   └── __pycache__
```

```
│   ├── __pycache__
│   ├── static
│   │   └── app_input_chklst
│   │       ├── css
│   │       ├── img
│   │       └── js
│   ├── templates
│   │   └── app_input_chklst
│   │       ├── dialogboxes
│   │       └── partials
│   └── test
│       └── __pycache__
├── app_user
│   ├── migrations
│   │   └── __pycache__
│   ├── __pycache__
│   ├── static
│   │   └── app_user
│   │       ├── css
│   │       ├── img
│   │       └── js
│   ├── templates
│   │   └── app_user
│   │       ├── dialogboxes
│   │       └── registration
│   └── test
│       └── __pycache__
├── app_utilities
│   ├── migrations
│   │   └── __pycache__
│   ├── __pycache__
│   ├── static
│   │   └── app_utilities
│   │       ├── css
│   │       ├── img
│   │       └── js
│   ├── templates
│   │   └── app_utilities
│   ├── templatetags
│   │   └── __pycache__
│   └── test
├── checklistmgr
│   └── __pycache__
├── media
│   ├── checklists
│   │   └── 2020
│   │       ├── 10
│   │       └── 11
│   ├── images
│   └── photos
│       └── 2020
```

```
│           └── 10
├── __pycache__
├── static
│   ├── css
│   ├── img
│   └── js
└── templates
    ├── errors
    └── partials
```

7 - Points particuliers

Aucun point particulier n'est à noter.

8 - Glossaire

Framework	Ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel
UML	Langage de modélisation graphique à base de pictogrammes conçu pour fournir une méthode normalisée pour visualiser la conception d'un système
GIT	Logiciel de gestion de version décentralisé
Pip	Gestionnaire de paquets Python (Gère les inter-dépendances)