NF18 P21

LABONNELOC

LOUER UNE VOITURE N'A JAMAIS ETE AUSSI SIMPLE

REDOUIN Noé
NGUYEN Bang
DUROCHER Violette
LERNER Thomas

Plan du rapport :

1. Le projet

Il y a quelques semaines, cher client, vous êtes venu solliciter nos services. Votre agence de location de voitures, située à Compiègne, commence à prendre de l'ampleur. Nombreux sont

les étudiants qui, attirés par vos prix de location incroyablement bas, viennent toquer à votre porte pour profiter d'une petite clio misérable le temps d'un semestre. Et ils sont ravis !

Vous œuvrez pour le bien de la communauté. Mais, face à votre succès, il vous fallait plus qu'une porte pour pouvoir répondre aux besoins de votre clientèle. Le temps de quelques semaines, vous avez fait confiance à notre petite start-up de développeurs passionnés pour façonner une application, à votre image, et qui vous ressemble, afin de pouvoir gérer les locations de façon totalement dématérialisée. Pour vous, nous avons relevé le défi.

Grâce à notre application, LBL (LaBonneLoc), enregistrer chaque nouvelle location sera un jeu d'enfant ! Sur notre application vous pouvez, entre autres :

- Avoir accès à la liste de tous les véhicules enregistrés dans votre agence,
- Enregistrer une nouvelle location (modifiable ou annulable à tout moment),
- Garder la trace de toutes les facturations et entretiens associés aux locations,
- Afficher un bilan par client, véhicule ou catégorie, ...

Notez bien que nous avons conçu notre application en gardant à l'esprit que votre agence était en développement exponentiel. Ainsi, s'il vous arrivait de multiplier les agences, dans Compiègne ou même ailleurs, vous pourriez préciser de quelle agence provient chaque véhicule ou agent (technique ou commercial), y travaillant.

Par ailleurs, nous savons que votre clientèle n'est pas exclusivement étudiante, et qu'il vous arrive également de faire affaire avec des entreprises, telles que Nike par exemple ! Notre application permet de prendre en charge le client, quel qu'il soit, de la façon la plus adaptée. Ainsi, une entreprise peut effectuer plusieurs locations, et être facturée une seule fois.

Enfin, pour assurer l'entière légalité de votre activité, notre application vérifie que vos clients ont tous 21 ans a minima, et demande un numéro de permis pour chaque conducteur, ainsi qu'une carte bancaire pour chaque location.

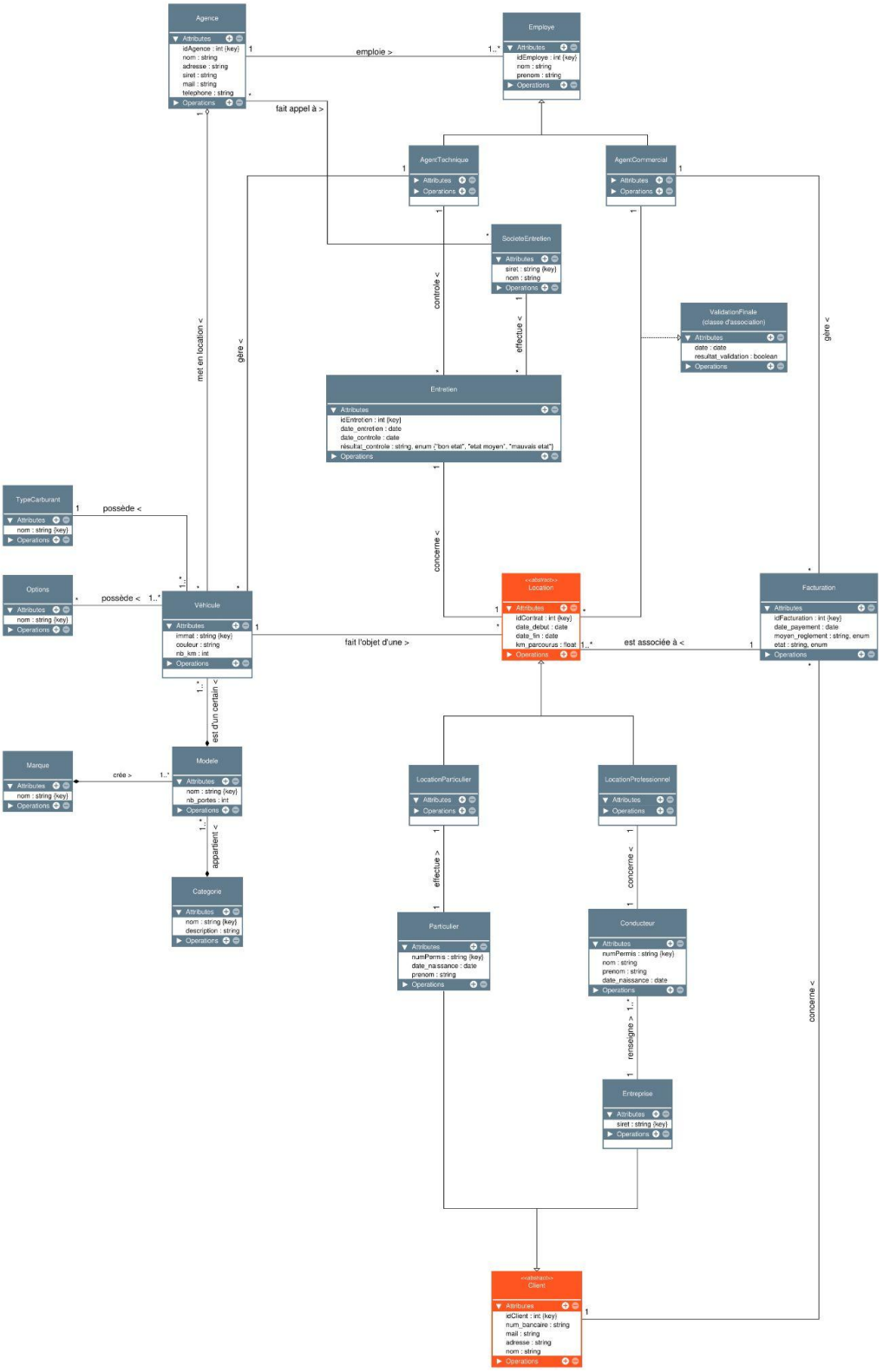
Vos employés, vos clients et vos associés (les agences d'entretien) seront tous ravis de cette application, agréable à l'utilisation et répondant à tous vos besoins !

Nous vous laissons découvrir son fonctionnement dans le descriptif qui suit,

Bien à vous, l'équipe LBL :)

2. Conceptualisation du problème

Modèle Conceptuel de Données en UML (détails en annexe cf partie 4) :



3. Fonctionnement de l'application

Pour stocker et gérer toutes vos données, nous avons créé une base de données que vous pourrez manipuler depuis une partie logicielle en Python permettant l'accès, la modification et l'affichage des données contenues dans cette BdD.

Notre base de données est conçue grâce aux fichiers suivants :

- create.sql (création des tables)
- data.sql (chargement d'un jeu de données)
- delete.sql (suppression des données des tables)
- drop.sql (suppressions des tables)

Notre application contient les fichiers suivants :

- app.py (gestion du menu)
- config.py (informations de connexion)
- utils.py (fonctions utilitaires)
- bang.py (gestion de quelques fonctionnalités du menu)
- violette.py (gestion de quelques fonctionnalités du menu)
- noe.py (gestion de quelques fonctionnalités du menu)
- thomas.py (gestion de quelques fonctionnalités du menu)

Comme vous pouvez le voir, nous avons nommé les fichiers d'après les membres de notre équipe, afin de vous laisser un souvenir de notre contribution :) Bien entendu, chaque membre de l'équipe a contribué à l'écriture de chaque fichier.

Voici comment utiliser notre application :

1) Remplissage de la base de données

Commençons par créer votre base de données dans PostgreSQL.

Passez par le vpn utc afin de pouvoir vous connecter au serveur tuxa.sme.utc. Normalement, vous devriez avoir un compte PostgreSQL associé à une base de données, auquel vous vous connectez depuis cette machine (grâce au client textuel sur linux ou au client graphique pgAdmin). Une fois dans PostgreSQL, exécutez les fichiers create.sql puis data.sql (pour avoir un jeu de données initial, qui pourra être modifié ensuite).

Votre base de données est maintenant créée et remplie.

2) Lancement de l'application

Chargez l'application dans un IDE python.

Notre application peut être utilisée par plusieurs utilisateurs. Pour vous connecter avec vos identifiants, rendez-vous dans le fichier config.py et rentrez vos informations de connexions.
(Rappel : Notre base de données a été conçue sur le serveur tuxa.sme.utc, port 5432.)

Super ! Vous n'avez plus rien à faire, à part lancer l'application.

Lancez le fichier app.py depuis le terminal. L'application se lance… Laissez-vous aller… C'est souple, c'est agréable, c'est LBL !

3) L'application

Vous y êtes ? C'est joli ? Si tout va bien, le menu suivant devrait s'afficher :

Sinon, vous avez le droit de venir nous taper sur les doigts.

Mais à priori, vous y voilà. Maintenant, voyons un peu ce qu'on vous propose.

L'application vous propose de sélectionner une option. Il suffit de taper le numéro correspondant à l'option qui vous intéresse.

1. Afficher la liste de véhicules

Dans le jeu de données fourni dans data.sql, des véhicules sont déjà pré-enregistrés. Vous pouvez tous les voir en tapant 1.
(NB : Pour une meilleure ergonomie, nous avons utilisé le paquet python ***tabulate****. Vous pouvez l'installer grâce à la commande* ***pip install tabulate****).*

2. Ajouter un véhicule

Rien de plus clair. Il vous suffira de rentrer :
- son numéro d'immatriculation (7 caractères),
- un modèle parmi les options proposées (attention à bien respecter l'orthographe du nom du modèle),
- le carburant (attention à l'orthographe à nouveau),
- la couleur (du vert canard au jaune citron, vous choisissez !),
- le nombre de km alors parcourus par le véhicules (un entier),

- l'agence à laquelle le véhicule appartient (dans votre cas, c'est toujours RENTACAR puisque vous n'avez qu'une seule agence pour l'instant),
- l'agent technique responsable du véhicule (vous pouvez ne rentrer que quelques lettres de son nom et de son prénom pour le retrouver, ou laissez vide pour voir toutes les options. Au final, il faudra rentrer son identifiant),

3. Ajouter une location

- Taper 1 ou 2 pour choisir le type du client. Vous devez alors sélectionner un client (par son identifiant), ou en ajouter un nouveau. Si vous ajoutez un nouveau client, celui-ci doit présenter un numéro de carte bancaire valable.

- Ensuite vous devez sélectionner une date de début et une date de fin de location. Attention au format, on est bilingues chez LBL ! Utilisez le format AAAA-MM-DD.

- Puis, vous devez sélectionner un véhicule. Par défaut, l'application vous demande un modèle (vous pouvez laisser cette donnée vide pour avoir accès à toutes les options), et vous propose les véhicules correspondant à ce modèle et qui sont disponibles pour les dates que vous avez indiquées (c'est bien fait, non ?). Rentrez (copiez-collez, même) l'immatriculation du véhicule choisi.

- Nous savons que votre entreprise fait appel à des sociétés pour réaliser l'entretien des véhicules. Une location doit donc toujours être associée à une société d'entretien. Vous pouvez taper le nom de la société d'entretien pour la retrouver ou laisser vide pour toutes les voir. Entrez ensuite le siret de la société. L'agent technique associé à l'entretien est celui associé au véhicule par défaut.

- Si le client est une entreprise et qu'il a déjà effectué une location facturée mais encore impayée chez vous, l'application propose d'ajouter la nouvelle location sur la facturation déjà existante. Si vous souhaitez créer une facturation différente, taper -1. Si le client est un particulier, une nouvelle facturation est créée par défaut.

- Ensuite vous pouvez choisir l'agent commercial responsable de la location et de la relation client.

- Si le client est une entreprise, la location doit être associée à un conducteur. Si aucun conducteur n'est déjà enregistré, vous pouvez en ajouter un. Attention ! Un conducteur doit avoir plus de 21 ans et un numéro de permis valable.

4. Annuler une location

La liste des locations s'affiche, à vous de sélectionner celle que vous voulez supprimer en rentrant son numéro de contrat.

5. Modifier une location

Vous pouvez modifier n'importe quel paramètre d'une location (la date_debut ou date_fin par exemple).

6. Valider une location

Selon ce que vous nous avez indiqué, seul un agent commercial peut décider ou non si une location est validée. Nous imaginons que l'agent se connecte à l'application, rentre son identifiant et voit la liste des locations auxquelles il est associé. Il peut alors en sélectionner une et décider du choix de la validation (1 pour ok, 0 pour non ok).

7. Payer une facturation

Parmi la liste de facturations proposées (classées par type de client), sélectionner l'identifiant de la facturation que vous souhaitez marquer comme payée. Vous pouvez spécifier le moyen de paiement.

8. Contrôler un entretien

Cette fois-ci, l'application prend le point de vue d'un agent technique. L'agent rentre son id, et peut contrôler les entretiens auquel il est associé.

9, 10, 11. Bilans

Vous voulez connaître les statistiques de votre agence ? A quel point un client vous est fidèle ? Quel modèle, ou encore quelle catégorie de véhicules connaît le plus de succès ?

Ces trois options permettent de jeter un œil aux bilans statistiques pour un client, un modèle de voiture ou une catégorie.

12. Trace des agents

Vous nous avez précisé que vous vouliez garder la trace de chaque opération effectuée par l'un de vos employés, et bien, c'est chose faite !

Toute opération réalisée par un agent technique ou commercial est enregistrée, qu'elle soit concluante ou non, et vous pouvez la retrouver grâce à cette dernière option.

4) Votre ressenti, cher client

Voilà, vous pouvez prendre en main notre application. Alors heureux ? N'hésitez pas à nous contacter si vous avez la moindre question, et à nous laisser une étoile sur l'AppleStore !

L'équipe LBL :)

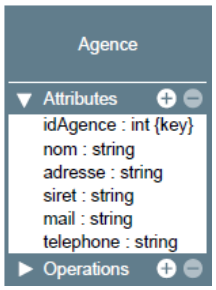
4) Annexe : détails sur l'Implémentation

1) MCD (UML)

- Le problème

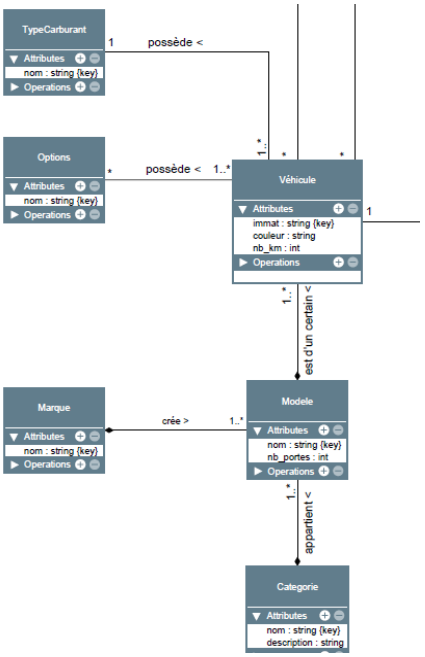
Comprendre et représenter les entités en jeu dans la base de données et les relations qu'elles entretiennent entre elles. Parmi ces entités : l'agence, ses employés, les locations, les clients, les véhicules et leurs composantes.
Au total, nous avons défini 20 classes pour modéliser l'entièreté du problème.

- Détails



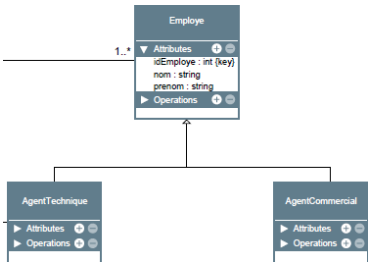
La classe Agence : sa présence donne la possibilité de répertorier plusieurs agences si à l'avenir cela s'avère utile. Chaque agence est identifiée par un numéro unique.
Liens:

- classe Véhicule (chaque véhicule appartient à une agence),
- classe Employé (chaque employé travaille dans une agence),
- classe Société Entretien (l'agence contacte des sociétés d'entretien).



La classe Véhicule et classes associées :
Un véhicule est identifié par son numéro d'immatriculation, et on note son nombre de kilomètres parcourus (modifié à chaque location).
Liens :

- Les classes Modèle, Catégorie et Marque permettent de référencer facilement chacun de vos véhicules. Il vous est également possible de spécifier pour chaque véhicule des options et son type de carburant,
- classe Agence,
- classe Location (une location est associée à un véhicule)
- classe Agent Technique (les agents techniques sont chargés de contrôler les entretiens des véhicules)



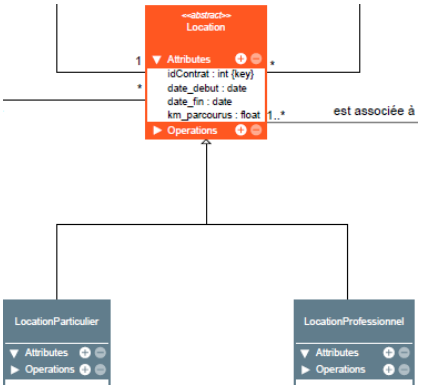
Les Employés : Héritage concernant les Agents Techniques et les Agents Commerciaux. La classe Employé n'est pas une classe abstraite : nous avons pensé que cette classe pourrait avoir d'autres instances futures différentes d'Agent Technique et Agent Commercial : un.e Directeur.ice par exemple.
Liens :

- classe Agence,
- Véhicule, Location, Entretien, Facturation.

Locations : Un particulier ne peut louer qu'un véhicule à la fois contrairement à un professionnel (une entreprise qui référence des conducteurs). De plus, une entreprise peut enregistrer plusieurs locations sous la même facturation, contrairement à un particulier.

Nous vous proposons deux classes : LocationParticulier et LocationProfessionnel. C'est cette fois-ci un héritage d'une classe abstraite.
Liens :

- classe Facturation (prendra en compte le type de location),
- classe Véhicule,
- classes Agent Technique et Agent Commercial (interviennent tous les deux dans la gestion d'une location).



La classe Validation Finale :
Vous avez bien précisé vouloir "garder la trace de toute opération effectuée par un agent". Il est donc nécessaire de pouvoir enregistrer toute validation, même non concluante. Nous avons donc choisi de représenter l'association Agent Commercial/Location par une classe d'association comportant les attributs date et résultat (la location étant validée ou non).

Les classes Client : Les classes Particulier et Entreprise héritent toutes les deux de la classe abstraite Client. Ils sont identifiés par un numéro idClient (clé artificielle).
Liens :

- selon le type du client LocationParticulier ou LocationProfessionel,
- Entreprise est liée à la classe conducteur,
- classe Facturation.

2) Modèle Relationnel de Données

● Formalisation en tables

Ce modèle précède le SQL et permet de traduire chaque entité et association définie en UML sous formes de tables. Chaque association devait se traduire ou bien en la création d'un nouvel attribut, ou bien en clé étrangère, et chaque relation donnait lieu à des contraintes.
(cf le rendu MLD pour plus de détails)

● Expressions des contraintes

Nous avons dû mettre en place un certain nombre de contraintes pour respecter le contrat et traduire le MCD dans son intégralité. Parmi ces contraintes : des contraintes de non nullité (ex : un employé appartient obligatoirement à une agence), des contraintes d'unicité (ex: entreprises différentes ne peuvent pas avoir le même numéro de carte bancaire), et des contraintes liées au contenu (ex: tous les conducteurs doivent avoir 21 ans ou plus).

3) SQL

Pour la gestion de la base, nous sommes passés par le système de gestion PostgreSQL, dans lequel nous avons créé maintes tables telles que Véhicule, Location, ou encore Agence. Pour les créer, nous avons utilisé un fichier create.sql contenant les requêtes SQL permettant la création de tables, avec la définition de leurs attributs ainsi que les contraintes y étant rattachées.

Voici un exemple de requête de création de table :

```sql
CREATE TABLE Agence(
  id_agence SERIAL PRIMARY KEY,
  nom VARCHAR,
  adresse VARCHAR,
  siret VARCHAR(14),
  mail VARCHAR,
  telephone VARCHAR(10),
  CONSTRAINT check_mel CHECK(mail SIMILAR TO '([a-zA-Z0-9.]+)@([a-zA-Z0-9]+).([a-zA-Z0-9]+)'),
  CONSTRAINT check_tel CHECK(telephone SIMILAR TO '[0-9]{10}'),
  CONSTRAINT check_siret CHECK(siret SIMILAR TO '[0-9]{14}')
);
```

On peut ici voir comment les attributs de la classe agence sont créés et définis par leur nom suivi de leur type (nombre entier, chaîne de caractères, ...). On notera aussi la définition de contraintes sur ces attributs. Prenons ici l'exemple de la contrainte posée sur mail : check_mel. Cette contrainte permet de vérifier que la chaîne de caractères entrée dans la table a bien la forme d'une adresse mail valide. Le critère SIMILAR TO permet de comparer la chaîne entrée avec le format classique d'une adresse mail. La contrainte définie [a-zA-Z0-9.] indique que cette partie peut être composée de lettres minuscules, majuscules et de chiffres entre 0 et 9, ainsi que du caractère '.' pouvant les séparer. On retrouve alors bien un format valide d'adresse mail. Cela empêche toute entrée invalide directement au niveau de la base de données.

4) Application python

- app.py :

Ce fichier est le point d'entrée de notre application qui est responsable d'afficher un menu à l'utilisateur. Nous avons mis l'accent sur l'évolutivité de l'application. Il est très facile d'implémenter de nouvelles fonctionnalités.

Pour l'affichage des menus, nous avons défini un dictionnaire, indexé par le numéro de chaque option. Chaque item dans le dictionnaire contient un couple, chaque couple possède un texte (qui sera affiché dans le menu) et une fonction qui sera exécutée lorsque l'utilisateur choisit cette option. Pour utiliser cette liste, nous la passons en argument de la fonction **menu**. La fonction parcourt la liste, affiche et retourne l'opération appropriée à chaque option.

L'affichage du menu au lancement de l'application >

- utils.py :

Les fonctions dans ce fichier sont des fonctions communes qui réalisent des opérations de base nécessaires pour les autres fonctions. Le fichier se charge aussi de la gestion de connexion à la base de données.

Quelques unes de ces fonctions:

- Connexion à la base de données :

```
conn = psycopg2.connect(dbname=cfg.DBNAME, user=cfg.DBUSER,
                        password=cfg.DBPWD, host=cfg.DBHOST, port=cfg.DBPORT)
curseur = conn.cursor()
```

Pour se connecter à la base de données depuis l'application python, nous passons par l'adaptateur python de PostgreSQL : psycopg2. Il vous suffit de rentrer vos informations de connexion dans le fichier config.py et tout fonctionnera !

- Affichage d'une table:

```
def afficher(text, table):
    header = [desc[0] for desc in curseur.description]
    print(text + "\n")
    print(tabulate(table, headers=header), end="\n\n")
```

Cette fonction prend en argument un texte et une table à afficher. Pour l'ergonomie de l'application, nous utilisons un paquet python **tabulate** (qui peut-être installé par la commande **pip install tabulate**). Voici un exemple d'affichage des véhicules sous forme d'un tableau :

```
Liste des vehicules

immat    modele    carburant     couleur      nb_km    agence    agent_tech
-------  --------  ------------  ---------  -------  --------  ------------
AA123AA  208       essence       bleu          3000         1             1
BB123BB  208       sans plomb98  gris           100         1             1
CC123CC  clio      sans plomb95  rouge        25000         1             2
DD123DD  clio      essence       blanc          400         1             2
EE123EE  clio      essence       rouge         5000         1             2
FF123FF  twingo    essence       noir         12000         1             2
GG123GG  golf      essence       noir          1200         1             1
HH123HH  golf      gasoil        gris          4000         1             2
ZZ123ZZ  twingo    gasoil        noir           100         1             2
```

- Sélectionner toutes les lignes d'une table :

```
def select_all(table):
    query = "SELECT * FROM %s" % table
    curseur.execute(query)
    return curseur.fetchall()
```

- Insérer des nouvelles valeurs dans une table:

```
def insert(table, colonnes, valeurs):
    query = "INSERT INTO %s(%s) VALUES(%s);" % (
        table, ",".join(colonnes), ",".join(["%s"] * len(colonnes)))
    curseur.execute(query, valeurs)
    conn.commit()
```

Cette fonction permet de simplifier l'insertion des données dans une table: il n'est pas nécessaire d'écrire une requête SQL à chaque fois. La fonction **insert** prend en arguments le nom de la table, la liste des colonnes et les valeurs correspondantes à insérer.

- Choix d'un tuple existant dans la base :

```
def choisir_agent(type_agent):
    print("Choix de l'agent", type_agent, ":")
    nom = input("\tNom (laissez vide pour ne pas prendre en compte) : ")
    prenom = input("\tPrenom (laissez vide pour ne pas prendre en compte) : ")
    type_agent_switcher = {
        "technique": choisir_agent_tech,
        "commercial": choisir_agent_com
    }
    return type_agent_switcher.get(type_agent)(nom, prenom)
```

Pendant l'exécution de l'application, nous demandons souvent à l'utilisateur de faire un choix parmi les propositions déjà enregistrées dans la base de données. Par exemple, il faut souvent sélectionner un agent technique ou commercial selon le besoin. Les agents sont identifiés par leur identifiant, mais l'utilisateur ne connaît pas à priori les identifiants de ses agents. Nous avons donc créé la fonction **choisir_agent** qui permet à l'utilisateur de chercher un agent par son nom ou prénom (ou les deux), puis affiche la liste filtrée des agents correspondants afin de faciliter le choix d'utilisateur. En fonction du type de l'agent, la fonction fait appel à une fonction **choisir_agent_tech** ou **choisir_agent_com** qui lancent des requêtes SQL appropriées. Nous avons défini des fonctions similaires pour tous les choix où l'utilisateur devait pouvoir visualiser les propositions.

- Choix d'un véhicule pour une location

La partie la plus compliquée et importante dans la gestion des locations est qu'on ne permet qu'une seule location pour chaque véhicule dans une période de temps, afin d'éviter le cas si un véhicule est réservé par deux ou plus clients en même temps. Cette contrainte est compliquée et difficile à implémenter par un CHECK dans le SQL car elle emploie les données dans les différentes lignes de la table. Nous avons décidé de le traiter par l'algorithme suivante :

- Demander la durée de la location (date de début et date de fin)
- Demander le modèle de véhicule à chercher (même principe avec la fonction **choisir_agent**)
- Retrouver tous les véhicules avec leurs informations de location.

```
def choisir_vehicule_nouvelle_location(date_deb_location, date_fin_location):

    date_deb_location = datetime.strptime(date_deb_location, "%Y-%m-%d").date()
    date_fin_location = datetime.strptime(date_fin_location, "%Y-%m-%d").date()

    modele = input("Modele (laissez vide si vous n'avez pas de preference) : ")

    #Retrouver tous les vehicules avec leurs locations
    query = """SELECT v.immat, v.modele, l.date_debut, l.date_fin FROM Vehicule v
                LEFT JOIN Location l ON v.immat = l.vehicule_immat
                WHERE v.modele LIKE %s
                ORDER BY v.immat, l.date_debut;"""
    curseur.execute(query, ("%" + modele.lower() + "%",))
    vehicules = curseur.fetchall()
```

- Les véhicules disponibles pour la location souhaitée (avec la date de début et la date de fin précisée) seront les véhicules qui, soit n'ont aucune location associée, soit n'ont pas été réservés pour une durée chevauchante à celle demandée.
    - Si le véhicule ne possède aucune location => Disponible
    - Sinon, vérifier liste des locations associées dans l'ordre chronologique si on peut insérer une nouvelle location sans violer la contrainte de non-chevauchement.

- bang.py:

Les fonctions définies dans ce fichier servent généralement aux statistiques des véhicules par catégorie. Nous voulons tout d'abord d'afficher un bilan des catégories. Dans ce bilan, nous voulons connaître le nom de la catégorie, le nombre total de véhicules, le nombre de véhicules en cours d'une location, la proportion (en pourcentage) des véhicules entrant dans cette catégorie et le montant total d'argent généré par cette catégorie.
Toutes ces informations sont obtenues avec une requête SQL qui renvoie une table, puis, en parcourant cette table, nous affichons les résultats à l'utilisateur. Pour éviter des informations redondantes, elle n'affiche que les catégories non-vide, c'est-à-dire les catégories qui possèdent au moins un véhicule.

```
SELECT  total.categorie, total.nb_vehicules, encours.nb_vehicules AS nb_vehicules_en_location,
        total.pourcentage AS pourcentage_total_vehicules, statistique.argent_rapporte
FROM
    (SELECT CategorieVehicule.nom as categorie, COUNT(Vehicule.immat) as nb_vehicules,
        (COUNT(Vehicule.immat) * 100.0 / SUM(COUNT(Vehicule.immat)) over ()) as pourcentage FROM CategorieVehicule
        JOIN Modele ON Modele.categorie = CategorieVehicule.nom
        JOIN Vehicule ON Modele.nom = Vehicule.modele
        GROUP BY CategorieVehicule.nom) AS total
JOIN (SELECT    CategorieVehicule.nom as categorie,
                SUM(CASE WHEN Location.date_debut <= current_date AND current_date <= Location.date_fin THEN 1 ELSE 0
      FROM CategorieVehicule
        JOIN Modele ON Modele.categorie = CategorieVehicule.nom
        JOIN Vehicule ON Modele.nom = Vehicule.modele
        JOIN Location ON Location.vehicule_immat = Vehicule.immat
        GROUP BY CategorieVehicule.nom) AS encours ON total.categorie = encours.categorie
JOIN (SELECT argent.categorie, SUM(argent.montant) as argent_rapporte FROM
        (SELECT DISTINCT CategorieVehicule.nom as categorie, Facturation.montant  FROM Location
            JOIN Facturation ON Facturation.idfacturation = Location.facturation
            JOIN Vehicule ON Vehicule.immat = Location.vehicule_immat
            JOIN Modele ON Vehicule.modele = Modele.nom
            JOIN CategorieVehicule ON CategorieVehicule.nom = Modele.categorie
            WHERE Facturation.etat_payement = TRUE) AS argent
      GROUP BY argent.categorie) AS statistique ON statistique.categorie = encours.categorie;
```

Liste de tous les véhicules et leur proportion

Trouver des véhicules avec locations encours

Le statistique du montant total d'argent par véhicule

Nous voulons aussi afficher les traces des opérations de chaque agent. Les différents types de traces seront : traces des facturations, traces des locations, traces des contrôles effectués. Pour chaque type de trace, nous avons défini une fonction correspondante, qui emploie une requête SQL de la table Employe en jointure avec les tables concernées. Par exemple cette fonction renvoie les traces des facturations de chaque agent :

```
def trace_des_facturation():
    query = """SELECT id_employe, nom, prenom, Facturation.* FROM Employe
                    JOIN Facturation ON Facturation.agent_com = Employe.id_employe
                    ORDER BY id_employe;"""
    curseur.execute(query)
    resultats = curseur.fetchall()
    afficher("\nLa trace des operations sur les facturations de chaque agent :", resultats)
```

- thomas.py :

Les fonctions définies sur ce fichier ont pour but d'afficher toutes les informations d'un client donné. Le programme demande à l'utilisateur s' il désire voir les informations d'un client particulier ou d'un client professionnel et exécute des fonctions différentes suivant la réponse. En effet, la structure de la base de données ne permettant pas de faire des requêtes similaires, il a fallu définir des fonctions d'affichage différentes. On prend ensuite en paramètre le numéro identifiant du client. On effectue ainsi une requête retournant la ligne de la BdD correspondant au client recherché.

```
try:
    choix = int(input("""Quel type de client recherchez-vous ?
                                1.Client Particulier
                                2.Client Professionnel (entreprise)
                    """))
    if choix == 1:
        client = input("Entrer le numéro de client : ")
        sql = "SELECT * FROM Particulier WHERE id_client = %s"
        curseur.execute(sql, (client,))
```

On souhaite alors afficher les informations contenues dans la base. On recherche le nombre de locations passées, en cours, à venir, ainsi que le total d'argent généré par les locations de ce client précis. Pour ce faire, nous avons défini des variables globales (nb_loc_encours, paye, paye_pro, ...). Les fonctions définies dans le fichier utilisent ces variables pour retourner la valeur d'une colonne dans le programme principal. Les fonctions ont la forme suivante :

```
def nombre_locations_passees(client):
    global nb_loc_passees
    sql = "SELECT COUNT(*) FROM Location JOIN LocationParticulier ON " \
          "Location.id_contrat=LocationParticulier.id_contrat WHERE LocationParticulier.particulier = %s AND " \
          "Location.date_fin::date < current_date "
    curseur.execute(sql, (client,))
    result = curseur.fetchone()
    nb_loc_passees = result[0]
    return nb_loc_passees
```

Les fonctions ayant toutes une forme similaire, nous pouvons prendre l'exemple de nombre_locations_passees qui retourne le nombre de locations passées d'un client particulier. On effectue la requête retournant le nombre de locations faites par le client et dont la date de fin est antérieure à la date du jour où on fait tourner le programme. On affecte cette valeur à une variable globale puis on appelle cette fonction dans le programme principal.

On affiche ensuite la valeur comme ceci :

```
if choix == 1:
    client = input("Entrer le numéro de client : ")
    sql = "SELECT * FROM Particulier WHERE id_client = %s"
    curseur.execute(sql, (client,))
    row = curseur.fetchone()
    afficher_client_particulier(row)
    nombre_locations_encours(row[0])
    nombre_locations_passees(row[0])
    nombre_locations_prevues(row[0])
    total_paye(row[0])
    print("Nombre de locations passées : ", nb_loc_passees)
    print("Nombre de locations en cours : ", nb_loc_encours)
    print("Nombre de locations prévues : ", nb_loc_prevues)
    print("Total payé par le client : ", paye)
```

On obtient ainsi cet affichage :

```
Quel type de client recherchez-vous ?
                                1.Client Particulier
                                2.Client Professionnel (entreprise)
                            1
Entrer le numéro de client : 1
Albert Pinot  :

    Date de naissance : 1998-07-07
    Adresse : None
    Numéro de téléphone : 0695605788
    Numéro de permis : 20AW24096
    ID client : 1
Nombre de locations passées :  1
Nombre de locations en cours :  0
Nombre de locations prévues :  0
Total payé par le client :  500,00 €
```

On a bien ici un bilan complet de l'activité du client, avec le nombre de locations effectuées et à venir, ainsi que ses informations personnelles.

- noe.py:

Les fonctions définies sur ce fichier ont pour but d'afficher toutes les informations d'un véhicule donné. Une fois l'immatriculation donnée, toutes les fonctions donnant les informations du véhicule sont lancées.

```
def bilan_vehicule():
    immat = input('Entrez l immatriculation d un véhicule pour le bilan. ')
    location_passee(immat)
    location_present(immat)
    location_future(immat)
    argent_total(immat)
    duree_moy_location(immat)
```

Location_passee et location_future donnent respectivement les locations qui ont eu lieu et se sont finies avant la date actuelle ainsi que celle n'ayant pas encore débutée en informant l'utilisateur. La fonction location_present informe l'utilisateur de la location en cours. Les informations données sont l'id_contrat, la date de début ainsi que la date de fin.

```
print ("idcontrat: ", raw[0], "date début:", raw[1], "date fin:", raw[2])
```

La fonction argent_total premet de calculer tout l'argent généré par les locations de la voiture grace à la fonction SUM(Facturation.montant).

```
def argent_total(immat):
    query = "SELECT SUM(Facturation.montant) FROM Location inner join Facturation ON Location.facturation=Facturation.
idFacturation inner join Vehicule on Location.vehicule_immat=Vehicule.immat WHERE etat_payement='TRUE' AND
date_fin::date<current_date and Vehicule.immat='%s'" % immat
    curseur.execute(query)
    raw1 = curseur.fetchone()
    print("voici l argent generee par la voiture:", raw1)
```

Enfin la fonction duree_moy_location donne la durée moyenne de location par véhicule, cela peut être utile afin de mieux gérer votre parc-automobile. Cette fonction utilise la fonction AVG(date_fin - date_debut) qui permet de donner la durée moyenne de la location , si votre SGBD la reconnait la fonction DATEDIFF() sera plus efficace et sûr pour faire la soustraction des dates.

- violette.py

Dans ce fichier se trouvent les fonctions permettant d'insérer notamment:

- les entretiens grâce à la fonction def ajouter_entretien() qui permet d'ajouter un agent technique et une société à la table grâce aux fonctions choisir_societe() et choisir_agent("technique").

- les locations, la fonction def ajouter_location():

  Cette fonction demande d'abord de choisir un client en commençant par demander si ce client est particulier ou professionnel. Ensuite la fonction fait appel à la fonctiongetIdClient(typeClient) cette fonction appel les fonctions def choisir_client_prof() ou def choisir_client_part() selon le type de client. Ces fonctions permettent de retrouver les clients existant et dans le cas où il n'existe pas encore

permet de créer leur profil avec les fonctions ajouter_client_part et ajouter_client_prof et renvoie finalement l'id client.

Elle demande ensuite les dates de locations, puis le véhicule parmis ceux disponibles sur ces dates grâce à la fonction choisir_vehicule_nouvelle_location (date_deb_location, date_fin_location) l'entretien est prévu puis insérer grâce à la fonction ajouter_entretien().

La facturation est rentré grâce aux fonctions ajouter_facturation_professionnel(idClient) et ajouter_facturation_particulier (idClient) selon le type de client.

Les fonctions ajouter_location_professionnel(id_contrat, idClient) et ajouter_location_particulier(id_contrat, idClient) permette de faire le lien avec les id_contrat et pour les locations professionnel permettent de choisir le conducteur concerné.

- La fonction modifier_location() permet de modifier les locations en choisissant l'attribut que l'on veut modifier.
- la fonction annuler_location() permet de choisir puis de supprimer de la BDD une location existante.
- la fonction payer_facturation() permet de choisir une facture impayée grâce à la fontion choisir_factu_impayee() et de renseigner le moyen de paiement utilisé et passé l'état de paiement de la facture a "payé".
- la fonction validation_finale_location() choisit un agent commercial, une location et si la location est bien validée ou non.
- la fonction controler_apres_location() affich toute les locations puis propose de remplir tous les attributs de la relation contrôle.