Florestan Biaux
Alexandre Laval
Salomé Quoy
Emma Zglinicki

Rapport NF18

Sommaire

Introduction

Nous avons travaillé sur une application destinée à gérer des parkings :

Contexte

Une société de parkings souhaite mettre en place une base de données afin de gérer ses différents parkings dans une ville. Chaque parking est composé de plusieurs places, en plein air ou couvertes, adaptées à certains types de véhicules (deux_roues; camion; véhicule simple). Les prix des parkings varient selon la zone dans laquelle ils se situent dans la ville (Centre-ville, zone industrielle, zone d'activités commerciales, etc.).

Chaque utilisateur de ces parkings est soit un occasionnel, soit un abonné chez cette société. Un utilisateur peut avoir un ou plusieurs véhicules. Les occasionnels paient des tickets à l'heure, dans des guichets ou sur des automates avec sa carte bancaire. Les abonnés disposent des cartes mensuelles renouvelables chaque mois. La société souhaite gérer les transactions ayant lieu lors de chaque paiement.

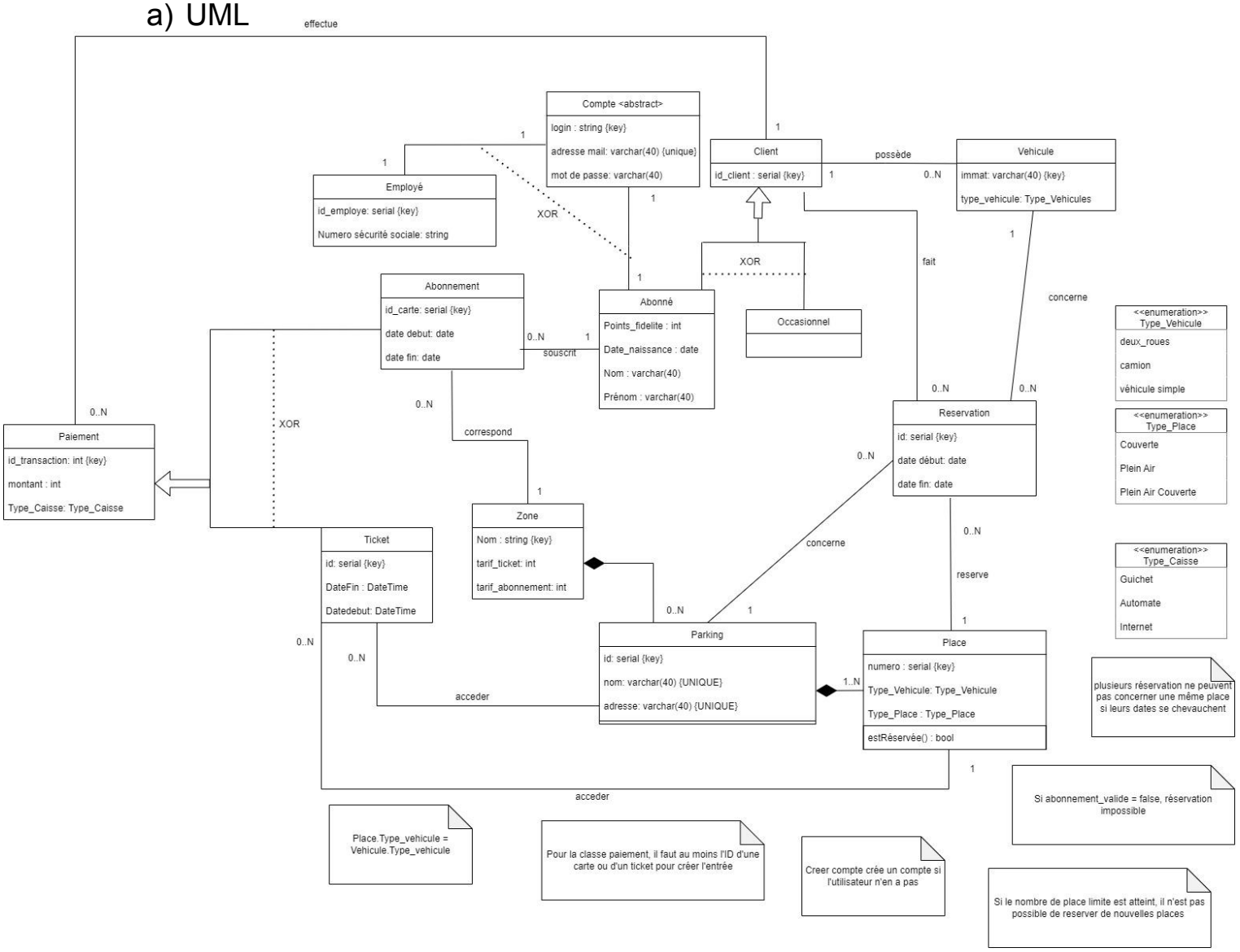
Pour attirer plus de clients, la société veut mettre en œuvre un système de fidélité pour ses clients. Pour chaque nouvel abonné, un compte lui est créé afin de sauvegarder ses différents abonnements.

Besoins

- Gérer l'ensemble des clients
- Gérer la réservation de places pour les abonnés et occasionnels
- Permettre la modification des tarifs des zones et permettre à un parking de changer de zone
- Gérer le système de fidélité de la société
- Afficher le nombre de places disponibles dans l'ensemble des parking
- Permettre de réaliser des études statistiques

I - Structure de la base de données

a) UML



b) MLD

Client(#id_client:serial) //classe mère de Abonne et Occasionel //héritage par référence

Vehicule(#immat:int, type_vehicule:enum:{deux_roues,camion,vehicule_simple}, proprietaire=>Client.id_client) avec type_vehicule, proprietaire NOT NULL

//Paiement classe mère de Abonnement et Tickets, héritage par référence

Paiement(#id_transaction:serial, montant:int, type_caisse:enum:{Guichet, Automate, Internet}, client => Client.id_client) avec montant, type_caisse, client NOT NULL

Zone(#nom:string, tarif_ticket:int, tarif_abonnement:int) avec tarif_ticket, tarif_abonnement NOT NULL

Parking(#id_parking:serial, zone=>Zone.nom, nom:string, adresse:string) avec nom, adresse NOT NULL

Place(#id_parking=>Parking.id_parking, #numero:int, type_vehicule:enum:{deux_roues,camion,vehicule_simple},type_place:enum:{couvert,plein air, plein air couverte}) avec type_vehicule, type_place NOT NULL, (id_parking, numero) KEY

Reservation(#id_reservation:serial, debut:Date, fin:Date, vehicule=>Vehicule.inmat, client=>Client.id_client, numero_place=>Place.numero, parking_place=>Parking.id_parking) avec debut, vehicule, client, numero_place, parking_placeNOT NULL and (debut<fin if (fin not null))

Employé(#id_employe: serial,num_secu:int) avec num_secu UNIQUE NOT NULL

Occasionel (#id_client => Client.id_client)

Abonne(#id_client => Client.id_client, points_fidelite : int, nom:string, prenom:string, date_naiss:Date) avec nom, prenom, date_naiss NOT NULL avec date_naiss < currentdate-17

Compte(#login: string, mail:string, mdp:string, employe => Employe.id_employe, abonne=> Abonne.id_client) avec mdp, employe, abonne NOT NULL and Mail, Employe, abonne UNIQUE et if employe null then abonne not null else abonne null

Abonnement(#id_carte:serial, id_transaction:Paiement.id, debut:Date, fin:Date, abonne=>Abonne.id_client, zone=>Zone.nom) avec debut, fin, abonne, zone NOT NULL et debut < fin

Ticket(#id_ticket:int, id_transaction:paiement.id_transaction, debut:date, fin:date, id_parking=>Parking.id_parking) avec type_vehicule, debut, parking.zone, parking.id NOT NULL avec debut < fin

Contraintes de cardinalité minimale : (les not null ont déjà été mis plus haut)

- Projection (Client, id_client) = Projection (Paiement, client)
- Projection (Abonne, id_client) = Projection (Abonnement, abonne)
- Projection (Reservation, vehicule) = Projection (Vehicule, immat)
- Projection (Client, id_client) = Projection (Vehicule, proprietaire)
- Projection (Abonnement, zone) = Projection (Zone, nom)
- Projection (Reservation, place.num) = Projection (Place, numero) et Projection (Reservation, place.park) = Projection (Place, parking)
- Projection (Ticket, parking.id) = Projection (Parking, id_parking) et Projection (Ticket, parking.zone) = Projection (Parking, zone)

Contrainte des héritages par références :

- INTERSECTION (PROJECTION(Abonnement,id_transaction), PROJECTION(Ticket,id_transaction)) = {}
- INTERSECTION (PROJECTION(Abonne,id_client), PROJECTION(Occasionnel,id_client)) = {}

c) Justification de nos choix

UtilisateursBDD :

- Nouvel utilisateur pour se créer un abonnement (et donc un compte)
- Abonné pour gérer ses abonnements, consulter ses paiements et réserver une place
- Employé pour consulter des statistiques et effectuer des actions au guichet (réservations, tickets, paiements, …)

Requêtes types :

- consulter les statistiques
- consulter le nombre de place restantes sur un parking
- réserver une place
- changer un parking de zone
- changer le tarif d'une zone
- se créer un compte
- requêtes associées aux users décrites précédemment
- consulter son solde de fidélité
- afficher le nombre de places disponibles dans l'ensemble des parkings
- …

***Compte** :* (classe mère)

Se compose d'un login (unique not null), d'un mot de passe(not null), et d'une adresse mail (unique not null). Un compte correspond soit à un abonné, soit à un employé

Employé :

Se compose d'un numéro de sécu social (unique not null)

- il correspond à un unique compte

***Abonné** :* (classe fille)

Se compose d'un nom(not null), d'un prénom(not null), d'une date de naissance (not null), et d'un solde de fidélité qui permettra de lui calculer des avantages.

- il hérite de client par référence
- il possède un compte
- il souscrit à autant d'abonnements qu'il le souhaite, sachant qu'il s'agit d'un abonnement par zone

Occasionnel *:* (classe fille)

Héritage complet par référence de la classe client

- si il souscrit à un abonnement il devient un abonné

Client :

Se compose d'un id_client (unique & not null). Classe mère de occasionnel et abonné

- il peut effectuer des paiements
- il peut effectuer des réservations
- il possède 0 ou plusieurs véhicules

Véhicule :

Se compose d'une immatriculation (unique not null), d'un type (not null): deux roues, camion, véhicule simple.

Il est nécessaire de connaître le type du véhicule pour lui attribuer une place adaptée dans le parking.

- Il peut être associé à une réservation
- Il appartient à un client

Réservation :

Se compose d'un id (unique not null), d'une date de début (not null) et d'une date de fin (not null), et d'une fonction permettant de vérifier si la réservation est possible date de fin > Date de début. Dans notre modélisation, un client peut réserver à l'avance afin d'être sûr qu'une place sera disponible pour lui. C'est gratuit pour celui qui a un abonnement et sinon, le jour de la réservation, l'occasionnel devra régler sur place. En fait cette classe est une assurance d'avoir une place disponible entre le début et la fin de la réservation.

- Elle est associée au véhicule qui a réservé la place
- Elle est faite par un client
- Elle réserve une place en particulier

Paiements : (classe mère abstraite)

Se compose d'un id de transaction (unique et non null), d'un montant (non null), d'un type de caisses (guichet, automate et internet) et d'une fonction permettant de calculer le montant. Classe mère de ticket et abonnement.

- Un paiement est effectué par un client

Abonnement *:* (classe fille)

Se compose d'un id_carte (unique not null), d'une date de début (not null) et d'une date de fin (not null) pour l'abonnement, et d'une fonction permettant de vérifier si l'abonnement est valide

- hérite par référence de paiements
- Il est souscrit par un abonné
- Il est relié à une zone. Dans notre modélisation, un abonnement est relié à une zone. Il peut donc y avoir autant d'abonnements que de zones
- Il est obtenu via un paiement

Tickets *:* (classe fille)

Se compose d'un id (unique et non null), varie en fonction du type de véhicules(non null), est lié à une date de début (non null) et une date de fin (quand le client s'apprête à payer)(non null) avec date de fin > Date de début

- varie en fonction du parking (car pas le même prix selon zone du parking), un ticket permet d'accéder à un parking d'une certaine zone
- hérite par référence de paiements
- Est effectué par un occasionnel ou un abonné ne possédant pas d'abonnement valide pour la zone du parking
- sont créés uniquement s'il reste de la place dans le parking

Parking :

Se compose d'un id(unique et non null), d'une adresse (unique et non null), d'un nom(unique et non null) et d'une fonction permettant de connaître le nombre de place limite et le nombre de places restantes en fonction du type de véhicule

- correspond à une zone (qui elle même correspond à un prix)
- est composé de places
- si on veut se garer sans être abonné, on doit prendre un ticket

Place:

Se compose d'un numéro (unique et non null), est adaptée à un type de véhicule (non null), est d'un certain type (couverte ou plein air) (non null), et possède une fonction avant de vérifier si elle est réservée ou non

- appartient à un parking, elle compose le parking
- peut être réservée ou non

Zone:

Se compose d'un nom (centre ville, zone industrielle, zone d'activités commerciales...)(unique et non null), d'un prix ticket à l'heure (non null), d'un prix abonnement mensuel pour une certaine zone (non null)

- peut être composée de parkings
- peut donner lieu à plusieurs abonnements

II - Application Python

Accueil :

```
Bienvenue chez Park'Auto, votre gestionnaire de parkings
Que voulez-vous faire ?
   1. Se connecter
   2. Se créer un compte
```

L'utilisateur peut se connecter ou créer un compte. Dans le cas d'une connexion, le menu affiché dépend ensuite de la nature du compte (employé ou client).
La création de compte ne concerne que les clients.

Connexion :

```
Quel est votre login ? employe

Quel est votre mot de passe ? e
```

La connexion se fait à l'aide d'un login et d'un mot de passe. La nature du compte est détectée automatiquement.

Requête utilisée : "SELECT mdp, employe, abonne FROM compte WHERE login='%s'"

```
dbnf18a060=> SELECT mdp, employe, abonne FROM compte WHERE login='employe';
 mdp | employe | abonne
-----+---------+--------
 e   |       1 |
(1 ligne)
```

On vérifiera ensuite que le mot de passe correspond bien à ce que la requête retourne. Si le requête ne retourne rien alors l'utilisateur n'existe pas.

Se créer un compte :

Lorsque ce menu est sélectionné, le programme demande le login souhaité par l'utilisateur et le compare à ceux déjà existants. Ensuite, il demande le reste des informations.

Requête utilisée :
"SELECT login FROM compte;"

```
Bienvenue chez Auto-loc, votre location auto-instantanée
Que voulez-vous faire ?
   1. Se connecter
   2. Se créer un compte
2
2
Quel login voulez-vous ? biauxflo
biauxflo
Saisissez votre mot de passe : azerty
azerty
Quelle est votre adresse mail ? bia.flo@etu.utc.fr
bia.flo@etu.utc.fr
```

Vient ensuite le choix du statut (employé ou client) et la vérification du numéro donné pour que chaque compte soit relié à un client existant ainsi qu'un client ne soit relié qu'à un seul compte (de même pour les employés).

"INSERT INTO compte VALUES(%s,%s,%s,%s)"
*"SELECT * FROM employe"*
"SELECT employe FROM compte"
*"SELECT * FROM abonne"*
"SELECT abonne FROM compte"

```
Êtes-vous employé ?
 0-Oui
 1-Non
1
1
Quel est votre identifiant client ? 1
1
Compte créé
```

Menu employé :

```
---Bienvenue dans l'espace employé---
Que voulez-vous faire?
[1] - Changer un parking de zone
[2] - Changer le tarif d'une zone
[3] - Voir le nombre d'abonnements par zone
[4] - Voir le nombre de parkings par zone
[5] - Nombre de Places sur un parking donné pour un type de véhicule
[6] - Voir nb de paiement pour chaque type
[7] - Voir le nombre de clients abonnés et occasionnels
[8] - Signaler l'entrée d'un véhicule
[9] - Signaler la sortie d'un véhicule
[10] - Terminer

Entrez votre choix :
```

On peut voir ici toutes les fonctionnalités implémentées pour les employés.

Changer un parking de zone :

Permet de changer la zone d'un parking.

Requête utilisée : "UPDATE Parking SET zone='%s' WHERE nom='%s';"

```
dbnf18a060=> UPDATE Parking SET zone='Industriel' WHERE nom='Mairie';
UPDATE 1
dbnf18a060=> SELECT * From parking;
 id_parking |     zone     |   nom    |                    adresse
------------+--------------+----------+---------------------------------------------
          2 | Centre-ville | Chateau  | 4 place du chateau, 60200 Compiegne
          3 | Industriel   | Usine    | 12 avenue du g├®n├®ral Leclerc,60200 Compiegne
          4 | Industriel   | Centrale | 1 Square du 8 mai 1945, 60200 Compiegne
          5 | Commercial   | Magasin  | 12 boulevard vivian,60200 Compiegne
          6 | Commercial   | Capitole | 135 rue des Bateliers, 60200 Compiegne
          1 | Industriel   | Mairie   | 1 place de la mairie,60200 Compiegne
(6 lignes)
```

Changer le tarif d'une zone :

Permet de changer le tarif abonnement ou ticket pour une zone donnée.

Requêtes utilisées :

"UPDATE Zone SET tarif_abonnement='%s' WHERE nom='%s';"

"UPDATE Zone SET tarif_ticket='%s' WHERE nom='%s';"

```
dbnf18a060=> UPDATE Zone SET tarif_ticket='6' WHERE nom='Industriel';
UPDATE 1
dbnf18a060=> SELECT * from zone;
     nom      | tarif_ticket | tarif_abonnement
--------------+--------------+-----------------
 Centre-ville |            2 |              15
 Commercial   |            6 |              20
 Industriel   |            6 |               5
(3 lignes)
```

Nombre d'abonnements par zone :

Parmi les statistiques proposées , on peut demander le nombre d'abonnements par zone.

Requête utilisée : "SELECT zone, Count(id_carte) FROM Abonnement GROUP BY zone;"

```
dbnf18a060=> SELECT zone, Count(id_carte) FROM Abonnement GROUP BY zone;
     zone     | count
--------------+-------
 Industriel   |     1
 Commercial   |     1
 Centre-ville |     1
(3 lignes)
```

Nombre de parkings par zone :

Requête utilisée : "SELECT zone, Count(id_parking) FROM Parking GROUP BY zone;"

```
dbnf18a060=> SELECT zone, Count(id_parking) FROM Parking GROUP BY zone;
     zone     | count
--------------+-------
 Industriel   |     3
 Commercial   |     2
 Centre-ville |     1
(3 lignes)
```

Nombre de places dans un parking pour un type de véhicule :

Requête utilisée : "SELECT Count(*) FROM Place WHERE id_parking = '%s' and type_vehicule = '%s';"

```
dbnf18a060=> SELECT Count(*) FROM Place  WHERE id_parking = '2' and type_vehicule = 'deux roues';
 count
-------
     1
(1 ligne)
```

Nombre de paiement pour chaque type :

Requête utilisée : "SELECT Count (type_caisse), type_caisse FROM Paiement GROUP BY type_caisse;"

```
dbnf18a060=> SELECT Count (type_caisse), type_caisse FROM Paiement GROUP BY type_caisse;
 count | type_caisse
-------+-------------
     4 | internet
     4 | automate
(2 lignes)
```

Nombre d'abonnés et d'occasionnels :

Requête utilisée : "SELECT Count (Abonne.id_client) FROM Abonne;"
Requête utilisée : "SELECT Count (Occasionnel.id_client) FROM Occasionnel;"

```
dbnf18a060=> SELECT Count (Occasionnel.id_client) FROM Occasionnel;
 count
-------
     2
(1 ligne)
```

Entrée dans un parking :

Un employé peut signaler l'entrée d'un véhicule dans un parking. Pour cela il renseigne l'immatriculation, le type de place désiré, le type du véhicule, l'id du parking et si le client est un abonné de la zone.
Avec toutes ces informations, on va commencer par identifier le véhicule. Si il n'est pas répertorié alors on va l'ajouter à la base de données et l'associer à un id occasionnel.
Sinon on va récupérer son propriétaire dans notre table de véhicules.
Si il y a des places libres on laisse entrer le véhicule en créant un ticket ou une réservation avec une fin nulle.
Requête utilisée : "INSERT INTO Ticket VALUES
(NULL,DEFAULT,'%s','%s','%s','%s',NULL);"

Edition d'un paiement :

Lors de la création d'un abonnement ou d'un ticket, le programme créé un nouveau paiement dont on calcule le montant en fonction du type (abonnement ou ticket) à l'aide de la fonction calculer_montant().
Ce montant est ensuite communiqué à l'utilisateur qui peut l'accepter ou le refuser.
Si le paiement est accepté, un nouveau paiement est créé et associé au ticket / abonnement en cours et lié au client.
Requête utilisée : "SELECT tarif_ticket FROM Zone where nom='%s'"
"SELECT debut,fin FROM Ticket where id_ticket='%s'"
"SELECT tarif_abonnement FROM Zone where nom='%s'"
"INSERT INTO paiement values (Default,'%s','%s','%s')"
"SELECT id_transaction FROM paiement where client='%s'"

Consultation des paiements :

Dans le menu client, il est possible de consulter les paiements effectués par le dit client.
Requête utilisée : "SELECT abonne from compte where login='%s'"
"SELECT id_transaction,montant,type_caisse from paiement where client='%s'"

III - Modifications NO-SQL

1) MongoDB

Pour modifier notre base de données, nous nous sommes basées sur la façon de coder du JSON sur MONGODB. La grande différence réside dans le fait que le JSON ne permet pas de construire une base de données relationnelle. Ainsi, il n'y a pas de schéma, de structure interne. Cela implique qu'on ne peut pas faire de référence entre table et donc que le volume de données est significativement plus important.
Prenons pour exemple la table parking.
En relationnel, il suffit d'expliquer que la zone référence la clé étrangère de Zone. Ainsi il suffira d'une jointure pour avoir accès à tous les attributs des deux tables.

```
CREATE TABLE Parking(
    id_parking serial primary key,
    zone varchar(40),
    nom varchar(40) unique not null,
    adresse varchar(100) unique not null,
    foreign key (zone) references zone(nom) on delete cascade on update cascade
);
```

Ce n'est pas le cas en Json sous MongoDb, ainsi pour cette même table :

```
db.parking.insert (
    {"id" : 1},
    {"zone" : [ {"nom" : "Centre-ville"},        //plus de référence alors on doit inclure les informations de la zone
                {"tarif_ticket" : 2},
                {"tarif_abonnement" : 15}]},
    {"nom" : "Mairie"},
    {"adresse" : "1 place de la mairie,60200 Compiegne"}
)
```

On doit ajouter les informations concernant la zone à l'intérieur de la table parking.

2) Json avec Postgres

Ainsi, les manipulations faites plus tôt ont permis de déterminer les modifications intéressantes à effectuer. Il semble donc intéressant de modifier les tables, de manière à ce que, à chaque fois qu'une référence sur une table étrangère doit être faite sur une table unique, on peut la remplacer par un attribut JSON.
Par exemple, un parking compose les places alors il semble intéressant de mettre en place un attribut JSON.
La structure de place devient alors : {"numero" : valeur}, {"type_vehicule" : valeur}, {"type_place" : valeur}

```
CREATE TABLE Parking(
    id_parking serial primary key,
    place json,
    zone varchar(40),
    nom varchar(40) unique not null,
    adresse varchar(100) unique not null,
    foreign key (zone) references zone(nom) on delete cascade on update cascade
);
```

On peut également penser à mettre véhicule en json dans client car il peut s'en occuper, il compose ces dernières :

```
CREATE TABLE Client(
    id_client serial primary key
    vehicule json
);
```