UNIVERSITÉ LIBRE DE BRUXELLES

Rapport : Villo !

Pierre Gérard, Titouan Christophe

INFO-H-303 Base de données

Esteban Zimányi, Michaël Waumans

Année académique 2014 - 2015

Table des matières

1 Diagramme entité association

1.1 Diagramme

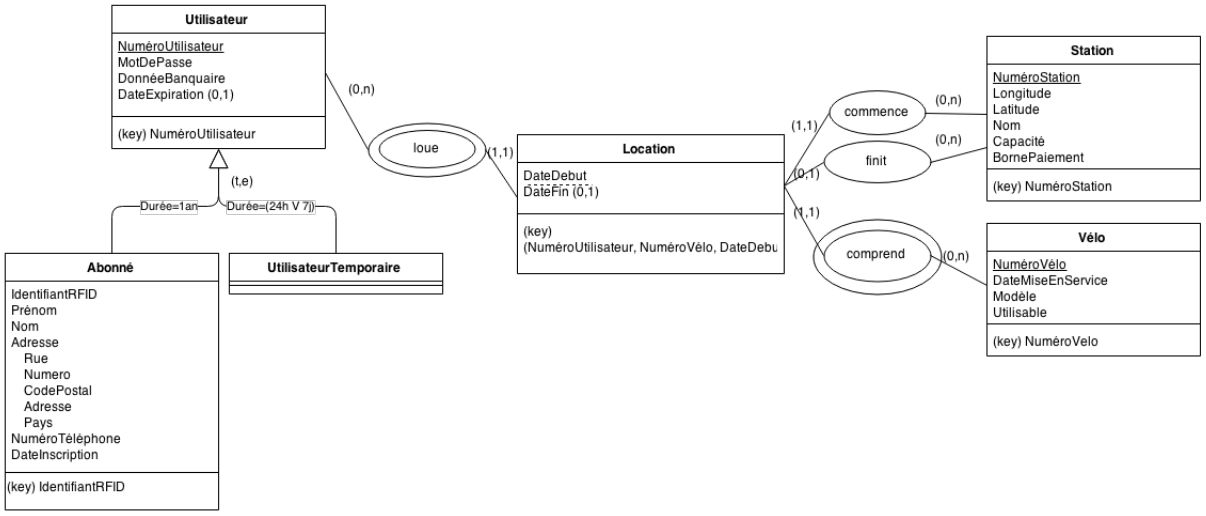

FIGURE 1 – Diagramme entité association

1.2 Contraintes

Les contraintes sont les suivantes :

- La DateDebut d'une Location doit précéder la DateFin,
- La DateMiseEnService d'un Vélo doit précéder la DateDebut de chacune de ses Locations,
- La DateExpiration d'un Utilisateur doit être postérieure à la DateDebut de toutes ses Locations,
- Le couple (Longitude, Latitude) est unique,
- La DateExpiration d'un Utilisateur doit être postérieure à la DateFin de toutes ses Locations,
- Un utilisateur qui a une date d'expiration non nul ne peut pas avoir plus d'une Location ayant une DateFin nul,
- Pour une Station, le nombre de vélo dont la dernière Location finit dans cette station ne doit pas dépasser sa capacité,
- Un Vélo ne peut pas avoir de déplacement disjoints. C'est a dire que la Station de départ du trajet n doit être la même que la Station d'arrivée du trajet $n-1$ pour $n \geq 0$,
- Un Utilisateur ayant une DateExpiration non nulme ne peut pas prendre un Vélo si Usable est faux.

2 Modèle relationnel

2.1 Modèle

Utilisateur(NuméroUtilisateur, MotDePasse, DonnéeBanquaire, *DateExpiration*)

Abonné(NuméroUtilisateur, IdentifiantRFID, Nom, Rue, Numero, CodePostal, Adresse, Pays DateInscription, NuméroTélephone)

Abonné.NuméroUtilisateur référence Utilisateur.NuméroUtilisateur

Location(NuméroUtilisateur, NuméroVélo, DateDebut,*DateFin*, NuméroStationDépart , NuméroStationFin)

Location.NuméroUtilisateur référence Utilisateur.NuméroUtilisateur

Location.NuméroVélo référence Vélo.NuméroVélo

Location.NuméroStationDépart référence Station.NuméroStation

Location.NuméroStationFin référence Station.NuméroStation

(NuméroUtilisateur, DateDebut) est unique

(NuméroVélo, DateDebut) est unique

Station(NuméroStation, Longitude, Latitude, Nom, Capacité, BornePaiement)

Vélo(NuméroVélo, DateMiseEnService, Modèle, Utilisable)

2.2 Contraintes

Les contraintes sont les suivantes :

— Une Location a au plus une station d'arrivé,
— Une Location a une et une seule Station de départ,
— Une Location a un et un seul Utilisateur,
— Une Location a un et un seul Vélo,
— La DateDebut d'une Location doit précéder la DateFin,
— La DateMiseEnService d'un Vélo doit précéder la DateDebut de chacune de ses Locations,
— Le couple (Longitude, Latitude) est unique,
— La DateExpiration d'un Utilisateur doit être postérieure à la DateDebut de toutes ses Locations,
— Un utilisateur qui a une date d'expiration non nul ne peut pas avoir plus d'une Location ayant une DateFin nul,
— Pour une Station, le nombre de vélo dont la dernière Location finit dans cette station ne doit pas dépasser sa capacité,
— Un Vélo ne peut pas avoir de déplacement disjoints. C'est a dire que la Station de départ du trajet n doit être la même que la Station d'arrivée du trajet $n-1$ pour $n \geq 1$,
— Un Utilisateur ayant une DateExpiration non nulle ne peut pas prendre un Vélo si Usable est faux.

3 Hypothèses sur le modèle

Il existe des utilisateur "admin", ce sont ces derniers et uniquement eux qui n'ont pas de date d'expiration.

Si un Abonné a son abonnement qui expire, il peut re-utiliser le NuméroUtilisateur et MotDePasse dans le futur, l'entité n'est pas supprimée.

Dans le cas ou des employés villo déplacent un vélo la nuit, alors ce déplacement doit être enregistré dans la base de donnée par un utilisateur admin.

Dans le cas ou un vélo serait cassé et devrait sortir du circuit de location, un Utilisateur admin vient le chercher et la Location ne finit jamais, c'est à dire pas de DateFin.

Dans le cas ou la société villo achète des nouveaux vélo et les met en circulation, un utilisateur admin fait une location de ce nouveau vélo qui a une date de départ égale à la date d'arrivée et une station de départ égale à la station d'arrivée.

Le champs MotDePasse contient un hash cryptographique du mot de passe et non le mot de passe lui-même.

4 Justification du modèle

Afin d'éviter la redondance et de garantir la cohérence du modèle, nous avons choisi de :

— Ne pas mettre d'attribut PlaceUtilisé dans Station,
— Ne pas mettre d'attribut Endroit dans Vélo,
— ect ...

En effet, ces informations peuvent être déduites à partir des locations.

Pour obtenir une clé primaire à l'entité Location, nous avons rendu obligatoire le champ DateDebut et c'est pour cela que la mise en circulation et la mise à la retraite des vélos sont différentes.

Pour la généralisation nous avons choisi une solution permettant d'avoir une relation à une seule table depuis la location et une solution permettant de mettre une contrainte d'existence sur tous les champs exceptés la DateExpiration.

5 Script DDL de création de la base de données

```
PRAGMA foreign_keys = ON;

CREATE TABLE IF NOT EXISTS station (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    payment BOOLEAN NOT NULL,
    capacity INTEGER NOT NULL
        CHECK(capacity >= 0),
    latitude REAL NOT NULL
        CHECK(-90<=latitude AND latitude<=90),
    longitude REAL NOT NULL
        CHECK(-180<=longitude AND longitude<=180),
    name VARCHAR(32) NOT NULL,

    UNIQUE(latitude, longitude)
);

CREATE TABLE IF NOT EXISTS bike (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    entry_date VARCHAR(20) NOT NULL
        CHECK(entry_date IS strftime(entry_date)),
    model VARCHAR(32) NOT NULL,
    usable BOOLEAN NOT NULL
);

CREATE TABLE IF NOT EXISTS user (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    password VARCHAR(64) NOT NULL,
    card VARCHAR(64) NOT NULL,
    expire_date VARCHAR(20)
        CHECK(expire_date IS strftime(expire_date))
);

CREATE TABLE IF NOT EXISTS subscriber (
    user_id INTEGER PRIMARY KEY,
    rfid TEXT UNIQUE NOT NULL,
    firstname TEXT NOT NULL,
    lastname TEXT NOT NULL,
    address_street TEXT NOT NULL,
    address_streenumber INTEGER NOT NULL,
    address_zipcode INTEGER NOT NULL,
    address_city TEXT NOT NULL,
    address_country TEXT NOT NULL,
    entry_date VARCHAR(20) NOT NULL
        CHECK(entry_date IS strftime(entry_date)),
    phone_number VARCHAR(20),

    FOREIGN KEY(user_id) REFERENCES user(id)
);

CREATE TABLE IF NOT EXISTS trip (
    departure_station_id INTEGER NOT NULL,
    departure_date VARCHAR(20) NOT NULL
        CHECK (departure_date IS strftime(departure_date)),
```

```
    arrival_station_id INTEGER,
    arrival_date VARCHAR(20)
56      CHECK(arrival_date IS strftime(arrival_date))
        CHECK(arrival_date >= departure_date),
    user_id INTEGER NOT NULL,
    bike_id INTEGER NOT NULL,

61  PRIMARY KEY(user_id, bike_id, departure_date),

    FOREIGN KEY(departure_station_id) REFERENCES station(id),
    FOREIGN KEY(arrival_station_id) REFERENCES station(id),
    FOREIGN KEY(user_id) REFERENCES user(id),
66  FOREIGN KEY(bike_id) REFERENCES bike(id)
);
```

6 Requêtes

6.1 Les utilisateurs habitant Ixelles ayant utilisé un Villo de la station Flagey

6.1.1 SQL

```sql
-- Les utilisateurs habitant Ixelles ayant utilise un Villo de la station Flagey

SELECT DISTINCT subscriber.user_id,
                subscriber.firstname,
                subscriber.lastname
FROM   subscriber
       INNER JOIN trip
               ON subscriber.user_id = trip.user_id
       INNER JOIN station
               ON trip.departure_station_id = station.id
WHERE  station.NAME = "FLAGEY"
       AND subscriber.address_zipcode = 1050;
```

6.1.2 Algèbre relationnelle

$$\Pi_{subscriber.user\_id,subscriber.firstname,subscriber.lastname}(((\sigma_{subscriber.address\_zipcode=1050}(subscriber)) \tag{1}$$

$$\bowtie_{subscriber.user\_id=trip.user\_id} (trip \bowtie_{trip.departure\_station_id=station.id} (\sigma_{station.name="FLAGEY"}(station)))) \tag{2}$$

6.1.3 Calcul relationnel

$$\{sb.user_id, sb.firstname, sb.lastname | subscriber(sb) \wedge \exists tp \exists st(trip(tp) \wedge station(st) \wedge tp.departure\_station\_id = st.id \tag{3}$$

$$\wedge sb.user\_id = tp.user\_id \wedge st.name = "FLAGEY" \wedge tp.departure\_station_id = st.id)\} \tag{4}$$

6.2 Les utilisateurs ayant utilisé Villo au moins 2 fois

6.2.1 SQL

```sql
-- Les utilisateurs ayant utilise Villo au moins 2 fois

SELECT DISTINCT trip1.user_id
FROM   trip AS trip1
       INNER JOIN trip AS trip2
               ON trip1.user_id = trip2.user_id
                  AND trip1.departure_date != trip2.departure_date;
```

6.2.2 Algèbre relationnelle

$$\Pi_{trip1.user\_id}(trip1 \bowtie_{trip1.user\_id=trip.user\_idANDtrip1.departure\_date!=trip2.departure\_date} trip2)$$

6.2.3 Calcul relationnel

$$\{t1.user_id | trip(t1) \wedge \exists t2(trip(t2) \wedge trip1.user\_id = trip2.user\_id \wedge trip1.departure\_date! = trip2.departure\_date)\}$$

6.3 Les paires d'utilisateurs ayant fait un trajet identique

6.3.1 SQL

```sql
-- Les paires d'utilisateurs ayant fait un trajet identique

SELECT DISTINCT t1.user_id,
                t2.user_id
FROM   trip AS t1
       INNER JOIN trip AS t2
               ON t1.departure_station_id = t2.departure_station_id
                  AND t1.arrival_station_id = t2.arrival_station_id
                  AND t1.user_id < t2.user_id;
```

6.3.2 Algèbre relationnelle

$$\Pi_{t1.user\_id,t2.user\_id}(\tag{5}$$

$$t1 \bowtie_{t1.departure\_station_id=t2.departure\_station_idANDt1.arrival\_station_id=t2.arrival\_station\_idANDt1.user\_id<t2.user\_id;} t2) \tag{6}$$

6.3.3 Calcul relationnel

$$\{t1.user\_id, t2.user\_id | trip(t2) \wedge trip(t1) \wedge t1.departure\_station_id = t2.departure\_station_id \tag{7}$$

$$\wedge t1.arrival\_station_id = t2.arrival\_station_id \wedge t1.user\_id < t2.user\_id \tag{8}$$

$$\wedge trip1.departure\_date! = trip2.departure\_date\} \tag{9}$$

6.4 Les vélos ayant deux trajets consécutifs disjoints (station de retour du premier trajet différente de la station de départ du suivant)

6.4.1 SQL

Voici, deux requêtes qui semblent prendre un temps équivalent à s'exécuter.

```
-- Les velos ayant deux trajets consecutifs disjoints (station de retour du
-- premier trajet differente de la station de depart du suivant)

SELECT DISTINCT bike_id FROM (
    SELECT t1.bike_id AS bike_id,
           t1.arrival_station_id AS s1,
           t2.departure_station_id AS s2,
           t1.arrival_date AS arrival,
           t2.departure_date AS departure
    FROM trip AS t1
    INNER JOIN trip AS t2 ON t1.bike_id=t2.bike_id AND t1.arrival_date<t2.departure_date
    WHERE t1.arrival_date NOT NULL --AND t1.bike_id=5
    GROUP BY t2.departure_date
    ORDER BY t1.departure_date,t2.departure_date)
WHERE s1 != s2;

-- Les velos ayant deux trajets consecutifs disjoints (station de retour du
-- premier trajet differente de la station de depart du suivant)


SELECT DISTINCT t1.bike_id
FROM   trip AS t1
       INNER JOIN trip AS t2
               ON t1.bike_id = t2.bike_id
                  AND t1.arrival_date <= t2.departure_date
                  AND t1.arrival_station_id != t2.departure_station_id
       LEFT OUTER JOIN trip AS t3
                    ON t1.bike_id = t3.bike_id
                       AND t1.arrival_date < t3.arrival_date
                       AND t3.arrival_date < t2.departure_date
WHERE  t3.bike_id IS NULL;
```

6.4.2 Algèbre relationnelle

$$\Pi_{t1.bike\_id}(\sigma_{t1.arrival\_station_id!=t2.departure\_station_idANDt1.arrival\_date<=t2.departure\_dateANDt3.bike_idISNULL}(\tag{10}$$

$$(t1 \bowtie_{t1.bike_id=t2.bike_id} t2) \rtimes_{t1.bike\_id=t3.bike\_idANDt1.arrival\_date<t3.arrival\_dateANDt3.arrival_date<t2.departure\_date} t3)) \tag{11}$$

6.4.3 Calcul relationnel

$$\{t1.user\_id | trip(t1) \wedge \exists t2(trip(t2) \wedge t1.arrival\_station\_id! = t2.departure_station_id \wedge \tag{12}$$

$$t1.arrival\_date <= t2.departure\_date \wedge \nexists t3(trip(t3) \wedge t1.bike\_id = t3.bike\_id \wedge t1.arrival\_date < t3.arrival\_date \wedge \tag{13}$$

$$t3.arrival\_date < t2.departure\_date))\} \tag{14}$$

6.5 Les utilisateurs, la date d'inscription, le nombre total de trajet effectués, la distance totale parcourue et la distance moyenne parcourue par trajet, classés en fonction de la distance totale parcourue

```
-- Les utilisateurs, la date d'inscription, le nombre total de trajet effectues,
-- la distance totale parcourue et la distance moyenne parcourue par trajet,
-- classes en fonction de la distance totale parcourue

-- SQLite ne dispose pas de fonction de la fonction sqrt(), ou de fonctions
-- trigonometriques, necessaires au calcul de la distance sur terre. Nous avons
-- donc implemente la formule de Haversine en tant qu'extension SQLite a
-- charger au demarrage, et qui integre la fonction
-- geodistance(lat1,long1,lat2,long2) -> km a l'environnement SQL.
-- Pour ce faire, compiler l'extension geodistance.c a l'aide du Makefile,
-- puis charger l'extension dans l'interpreteur SQLite avant d'executer la requete

SELECT   subscriber.firstname,
         subscriber.lastname,
         subscriber.entry_date,
         COUNT(trip.arrival_station_id) AS trip_count,
         SUM(geodistance(from_.latitude, from_.longitude, to_.latitude, to_.longitude))
    AS total_km,
         AVG(geodistance(from_.latitude, from_.longitude, to_.latitude, to_.longitude))
    AS avg_km
FROM     trip
INNER JOIN station AS from_ ON trip.departure_station_id = from_.id,
           station AS to_ ON trip.arrival_station_id = to_.id,
           subscriber ON trip.user_id = subscriber.user_id
WHERE    trip.arrival_station_id NOT null
GROUP BY subscriber.firstname,
         subscriber.lastname
ORDER BY count(trip.arrival_station_id);
```

6.6 Les stations avec le nombre total de vélos déposés dans cette station (un même vélo peut-être comptabilisé plusieurs fois) et le nombre d'utilisateurs différents ayant utilisé la station et ce pour toutes les stations ayant été utilisées au moins 10 fois.

```
-- Les stations avec le nombre total de velos deposes dans cette station
-- (un meme velo peut-etre comptabilise plusieurs fois) et le nombre
-- d'utilisateurs differents ayant utilise la station et ce pour toutes les
-- stations ayant ete utilisees au moins 10 fois.


SELECT station.NAME,
       Count(trip.arrival_station_id),
       Count(DISTINCT trip.user_id)
FROM   station
       INNER JOIN trip
               ON trip.arrival_station_id = station.id
GROUP  BY station.id
HAVING Count(trip.arrival_station_id) >= 10;
```

7 Implémentation

7.1 Langages et bibliothèques

Nous avons écrit l'application Villo ! en Python 2, en utilisant
— Flask (Python) comme framework web,
— Jinja2 comme moteur de template pour flask,
— Bootstrap et JQuery comme frameworks de frontend,
— **Sqlite3**, avec le module `sqlite3` de la librairie standard Python comme moteur de base de donnée.

7.2 Script d'insertion des données

```
# -*- coding: utf-8 -*-

from csv import DictReader
from sqlite3 import Connection
from sys import argv
import xml.etree.ElementTree as ET

from dbutils import hash_password

def create_tables(db, create_file="createDB.sql"):
    # Create tables using createDB.sql
    # (sqlite3 module cannot run multiple statements at once)
    creator = open(create_file).read().split(';')[:-1]
    for create_table in creator:
        db.execute(create_table)

def insert_bikes(db, input_file="data/villos.csv"):
    q = 'INSERT INTO bike (id,entry_date,model,usable) VALUES (?,?,?,?)'
    bikelist = (
        (
            int(bike['nume\xcc\x81ro']),
            bike['mise en service'],
            bike['mode\xcc\x80le'],
            bike['fonctionne'] == 'True'
        ) for bike in DictReader(open(input_file), delimiter=';')
    )
    with db:
        db.executemany(q, bikelist)

def insert_stations(db, input_file="data/stations.csv"):
    q = 'INSERT INTO station (id,payment,capacity,latitude,longitude,name) VALUES
    (?,?,?,?,?,?)'
    stationlist = (
        (
            int(station['nume\xcc\x81ro']),
            station['borne de paiement'] == 'True',
            int(station['capacite\xcc\x81']),
            float(station['coordonne\xcc\x81e Y']),
            float(station['coordonne\xcc\x81e X']),
            station['nom']
        ) for station in DictReader(open(input_file), delimiter=';')
    )
    with db:
        db.executemany(q, stationlist)

def insert_users(db, input_file="data/users.xml"):
    users_dom = ET.parse(input_file).getroot()
    users, subscribers = [], []

    for subscriber in users_dom.find('subscribers'):
        userdata = {f.tag: f.text for f in subscriber if f.tag != "address"}
```

```
            addr = subscriber.find('address')
            users.append((
                int(userdata['userID']),
                hash_password(userdata['password']),
                userdata['card'],
                userdata['expiryDate']
            ))
            subscribers.append((
                int(userdata['userID']),
                userdata['RFID'],
                userdata['firstname'],
                userdata['lastname'],
                addr.find('street').text,
                addr.find('number').text,
                addr.find('cp').text,
                addr.find('city').text,
                "Belgique",
                userdata['subscribeDate'],
                userdata['phone']
            ))

        for tmpuser in users_dom.find('temporaryUsers'):
            userdata = {f.tag: f.text for f in tmpuser}
            users.append((
                int(userdata['userID']),
                userdata['password'],
                userdata['card'],
                userdata['expiryDate'],
            ))

        with db:
            db.executemany(
                'INSERT INTO user (id,password,card,expire_date) VALUES (?,?,?,?)',
                users)
            db.executemany(
                'INSERT INTO subscriber (user_id,rfid,firstname,lastname,address_street,
        address_streenumber,address_zipcode,address_city,address_country,entry_date,
        phone_number) VALUES (?,?,?,?,?,?,?,?,?,?,?)',
                subscribers)

    def insert_trips(db, input_file="data/trips.csv"):
        with db:
            assert db.execute('INSERT INTO user (password,card,expire_date) VALUES ("%s
        ","","")' % (hash_password('admin'))).rowcount == 1
            res = db.execute('SELECT id FROM user WHERE password="%s" AND expire_date=""' % (
        hash_password('admin')))
            admin_id = res.next()[0]
            assert db.execute('INSERT INTO subscriber (user_id,rfid,firstname,lastname,
        address_street,address_streenumber,address_zipcode,address_city,address_country,
        entry_date) VALUES (?,?,?,?,?,?,?,?,?,?)',
                (admin_id, "0", "Administrateur", "De Villos", "Street","1","1","Bruxelles","
        Belgique","2010-01-01T00:00:01")).rowcount == 1

        print "Admin user has id=%d and password=admin" % admin_id

        def extract_trip(trip):
            if trip['depart'] == 'None':
                trip['depart'] = trip['arrive\xcc\x81e']
            if trip['heure de\xcc\x81part'] == 'None':
                trip['heure de\xcc\x81part'] = trip['heure arrive\xcc\x81e']
            if trip['utilisateur'] == 'None':
                trip['utilisateur'] = int(admin_id)
            return (
                int(trip['depart']),
                trip['heure de\xcc\x81part'],
```

```
            int(trip['arrive\xcc\x81e']) if trip['arrive\xcc\x81e'] != 'None' else None,
            trip['heure arrive\xcc\x81e'] if trip['heure arrive\xcc\x81e'] != 'None' else
    None,
            trip['utilisateur'],
            int(trip['ve\xcc\x81lo'])
        )

    q = 'INSERT INTO trip (departure_station_id,departure_date,arrival_station_id,
    arrival_date,user_id,bike_id) VALUES (?,?,?,?,?,?)'
    with db:
        db.executemany(q,
            map(extract_trip, DictReader(open(input_file), delimiter=';')))

def initDB(db_file=":memory:"):
    def do_nothing(*args):
        pass

    TABLES = [
        ('bike', 'data/villos.csv', insert_bikes, 2000),
        ('station', 'data/stations.csv', insert_stations, 179),
        ('user', 'data/users.xml', insert_users, 1096),
        ('subscriber', 'data/users.xml', do_nothing, 552),
        ('trip', 'data/trips.csv', insert_trips, 46717),
    ]

    db = Connection(db_file)

    def count_all(table):
        return db.execute("SELECT COUNT(*) FROM %s;" % (table)).next()[0]

    create_tables(db)
    for table, input_file, insert, expected_rows in TABLES:
        insert(db, input_file)
        assert count_all(table) == expected_rows, "Should insert %d rows in '%s'" % (
    expected_rows, table)

if __name__ == "__main__":
    from config import DATABASE
    initDB(argv[1] if len(argv) > 1 else DATABASE)
```

7.3 Modèle

Nous avons écrit un ORM minimal (fichier `models.py`), qui définit les classes **User**, **Bike**, **Station** et **Trip**. Ces classes contiennent la logique pour construire des objets depuis, et enregistrer des objets dans la base de données, ainsi que des requêtes sur mesure pour récupérer les objets selon leurs relations. Toutes les classes concrètes du modèle héritent de la classe **Model**, qui définit les méthodes d'accès basiques.

7.3.1 Model.create

Insère un objet dans la table. Le nom de la table et ses colonnes sont enregistrés dans un attribut de classe, et les valeurs `(?)` sont passées en paramètre.

```
INSERT INTO <table> (<colonnes>) VALUES (?);
```

7.3.2 Model.all

Récupère tous les objets de la table

```
SELECT <colonnes> FROM <table>;
```

7.3.3 Model.count

Compte tous les objets de la table

```
SELECT COUNT(*) FROM <table>;
```

7.3.4 Model.get

Récupère un objet en fonction de son id. N'est pas implémenté pour `Trip`, pour lequel la clef primaire est composite, mais qui peut être récupéré par ses associations.

```
SELECT <colonnes> FROM <table> WHERE id=?;
```

7.4 Relations

7.4.1 Bike.location

Renvoie la localisation actuelle d'un vélo (une station ou `None` si le vélo est en déplacement)

```
SELECT DISTINCT <colonnes> FROM station
INNER JOIN trip ON station.id=trip.arrival_station_id
WHERE bike_id=?
ORDER BY departure_date DESC;
```

D'autres requêtes permettent, similairement à `Model.get` et `Model.all` avec l'adjonction d'une clause `WHERE`, de récupérer tous les trajets d'un vélo, tous les vélos utilisables, et tous les vélos inutilisables.

7.4.2 User.is_subscriber

Renvoie `True` si l'utilisateur est un abonné. Cette méthode n'exécute pas de requête en elle-même, mais illustre le fonctionnement un peut particulier de la classe `User`. Puisque nous pouvons avoir des utilisateurs abonnés, et des temporaires, et que les informations d'abonnés sont dans une autre table, à la construction d'un objet on initialize les attributs d'abonnés à `None`, et on vérifie s'il existe dans la table `subscriber`. Si c'est le cas, ces attributs sont initialisé avec le contenu de la table.

```
SELECT id, password, card, expire_date, rfid,firstname, lastname,
       address_street, address_streenumber, address_zipcode, address_city,
       address_country, entry_date, phone_number
FROM user
LEFT JOIN subscriber ON user.id=subscriber.user_id
WHERE id=? LIMIT 1;
```

7.4.3 User.active_trips, User.current_trip

Un administrateur peut avoir plusieurs locations en cours, récupérées par la requête

```
SELECT <colonnes> FROM trip WHERE user_id=? AND arrival_station_id IS NULL;
```

On peut récupérer le voyage actuel d'un utilisateur (un seul en cours) en ajoutant une clause `LIMIT 1` pour un utilisateur normal.

7.4.4 User.billable_trips

Renvoie toutes les locations facturables (dépassant 30 minutes) d'un utilisateur, pour une période donnée.

```
SELECT <colonnes> FROM <trip>
WHERE ?<=departure_date AND
      arrival_date<=? AND
      user_id=? AND
      strftime('%s',arrival_date)-strftime('%s',departure_date) >= 1800;
```

7.4.5 Trip

Pour toutes les clefs étrangères dans la table `trip`, il exite un accesseur analogue à `Model.get` renvoyant l'objet lié.

7.4.6 Trip.distance

Renvoie la distance à vol d'oiseau entre la station de départ et d'arrivée de la location. Cette distance est calculée grâce à une extension C pour sqlite3 que nous avons du implémenter, à cause du manque de fonctions mathématiques dans le moteur de bases de données.

```
SELECT geodistance(s1.latitude,s1.longitude,s2.latitude,s2.longitude)
FROM trip
INNER JOIN station AS s1 ON s1.id=trip.departure_station_id,
           station AS s2 ON s2.id=trip.arrival_station_id
WHERE trip.departure_date=? AND
      trip.user_id=? AND
      trip.bike_id=?;
```

7.4.7 Station.bikes

Permet de connaître le nombre de vélos à une station. On cherche tous les vélos dont le dernier déplacement a cette station pour arrivée. Cette requête peut être déclinée en plusieurs version, pour obtenir les objets vélos à cette station (on remplace le `COUNT(*)` par les colonnes de `bike`), ou en ajoutant une clause `WHERE` pour compter seulement les vélos utilisable, cassés, ou pour connaître le nombre de places libres (en retranchant le nombre de vélos à la capacité de la station).

```
SELECT COUNT(bike_id) FROM (
    SELECT user_id,bike_id,arrival_station_id,MAX(departure_date)
    FROM trip
    GROUP BY bike_id)
JOIN bike ON bike.id=bike_id
WHERE arrival_station_id=?
```

8 Apports personnels

8.1 Intégration de OpenStreetMap

Pour rendre l'interaction avec notre application plus visuelle nous avons ajouté des vues cartes à notre application (Figure 2). Celles-ci sont affichées grâce à Leaflet [1], une bilbliothèque Javascript libre, affichant des données d'OpenStreetMap (données cartographiques libres). Des popups s'ouvrant sur la map permettent de voir le nombre de vélos et places utilisables dans une station en un coup d'oeil.

1. http ://leafletjs.com/

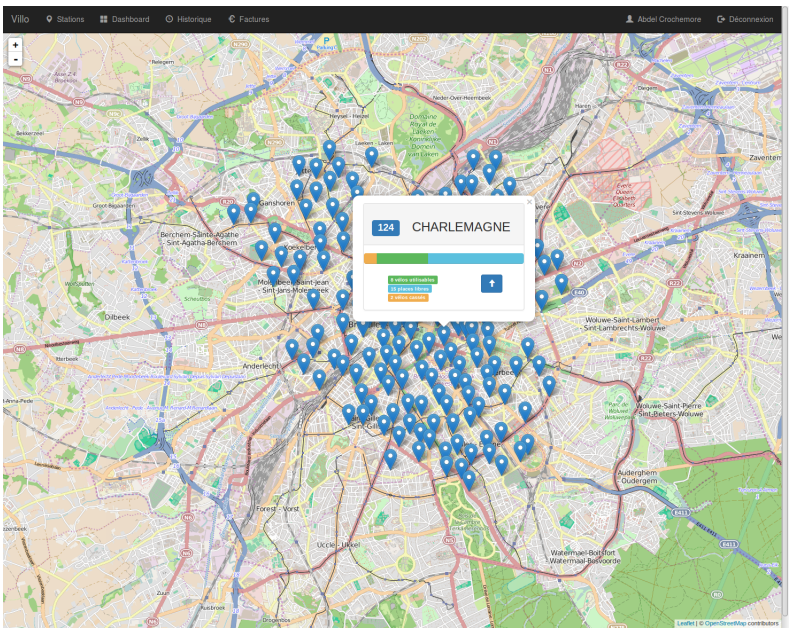

FIGURE 2 – Vue carte

8.2 Récapitulatifs/Factures téléchargeable en CSV

Pour faciliter l'utilisation de villo par le client, nous lui offrons la possibilité de télécharger une facture et un historique détaillés et complets au format CSV (Figure 3). De cette manière il pourra ouvrir ces derniers dans son tableur (Microsoft Excel, LibreOffice, Google Spreadsheet, ...).

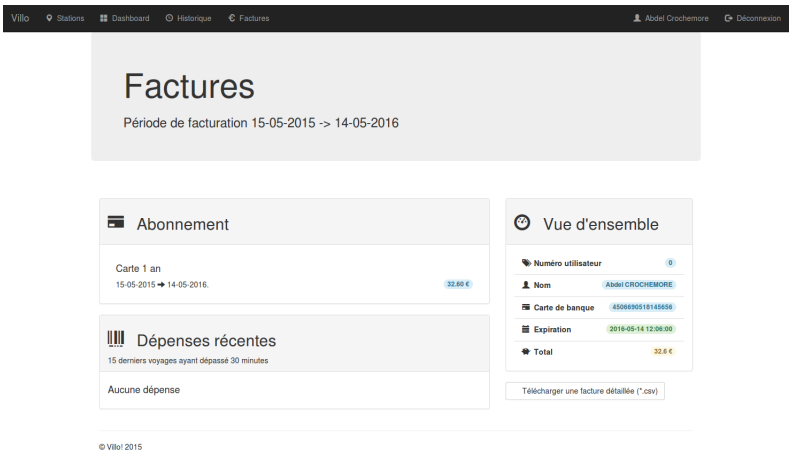

FIGURE 3 – Vue facturation

8.3 Utilisateur administrateur

La gestion d'une flotte de villo n'est pas aisé. C'est pourquoi nous avons implémenté des facilités pour les gestionnaires (Figures) : un compte administrateur qui peut faire l'ensemble des opérations suivantes :

— Ajouter des nouveaux vélos dans le circuit a une station,

— Ajouter des nouvelles stations directement sur la carte,
— Sortir un vélo du circuit,
— Réparer un vélo cassé (si l'administrateur l'a en location ou que le villo est à une station).

Notons que la réparation des vélos n'aurait aucun sens si les utilisateurs ne peuvent marquer un vélo comme cassé, cette dernière fonctionnalité est donc elle aussi implémentée.

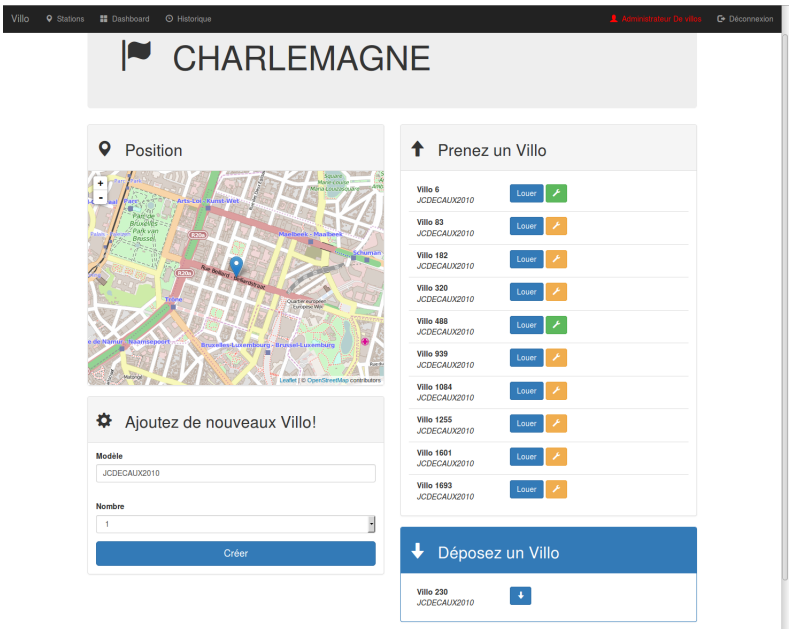

FIGURE 4 – Vues d'administration d'une station : ajout de villos, possibilité d'en marquer comme réparés ou de déposer n'importe lequel de ses vélos en cours d'emprunt

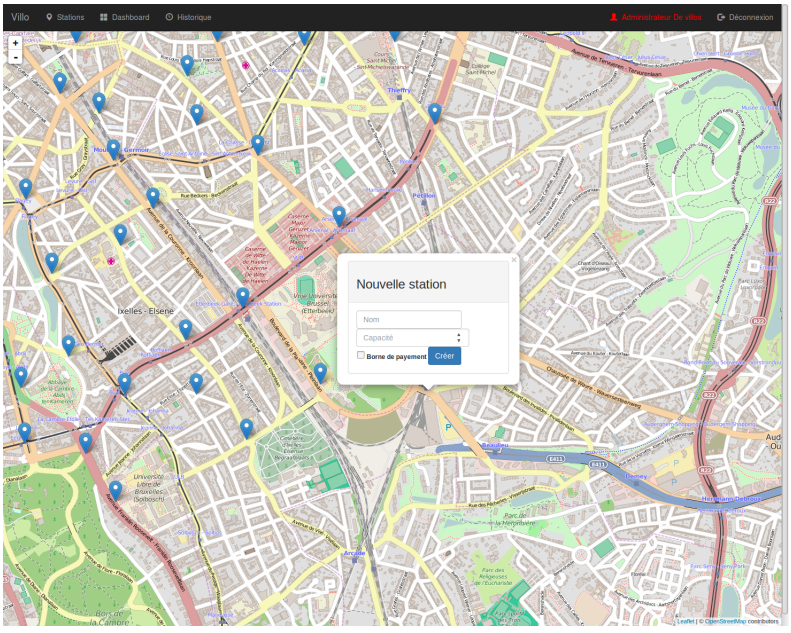

FIGURE 5 – Création d'une station

8.4 Utilisateur temporaire

Notre implémentation prend en compte les utilisateurs temporaires. En effet il est possible de s'inscrire pour 1 jour ou une semaine (Figure 6).

Un utilisateur non-connecté peut louer un villo dans une station disposant d'un point de payement en créant un compte d'utilisation temporaire.

FIGURE 6 – Inscription temporaire (l'autre onglet permet de souscrire à un abonnement)

8.5 Calcul de distances

Dans l'historique des déplacement, la distance à vol d'oiseau entre la station de départ et d'arrivée s'affiche, dans les vues tableau de bord (Figure 10), historique (Figure 8) et factures (Figure 3).

8.6 Interface mobile-friendly

La tendance actuelle est à l'utilisation mobile. Nous avons donc créé un site qui fonctionne sur tout type d'appareil, y compris les téléphones intelligents et tablettes.

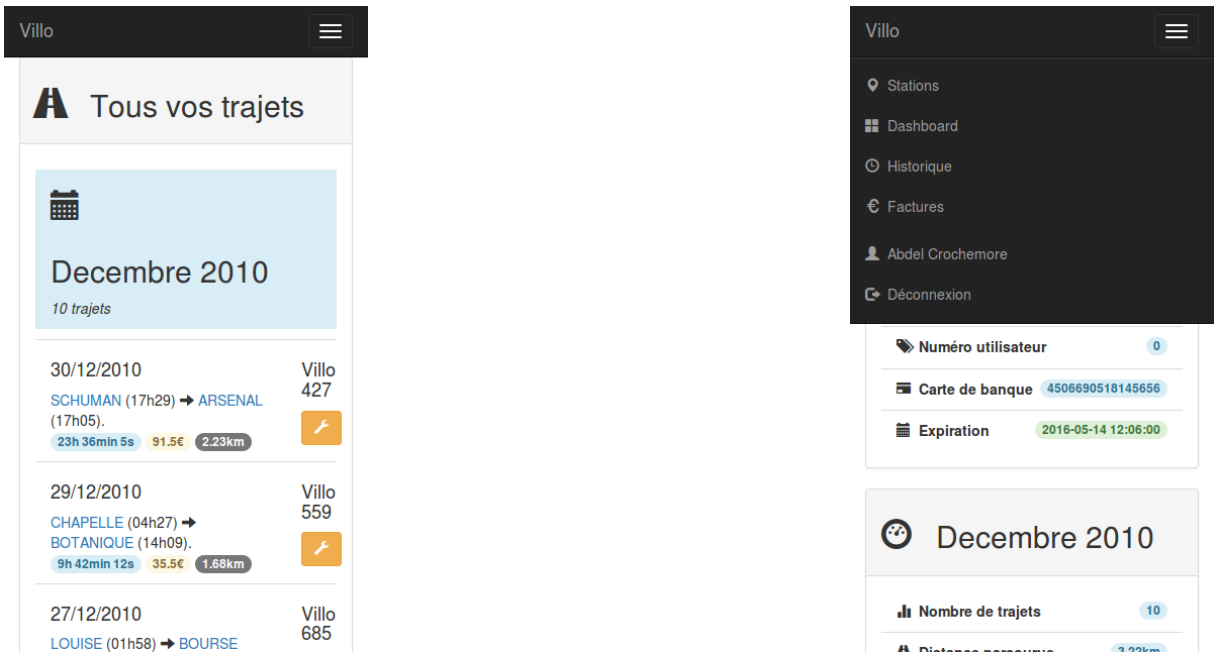

FIGURE 7 – Vues sur mobile

8.7 Récapitulatif par mois

Pour faciliter la lecture de l'historique et de la facturation, les trajets sont regroupés par mois

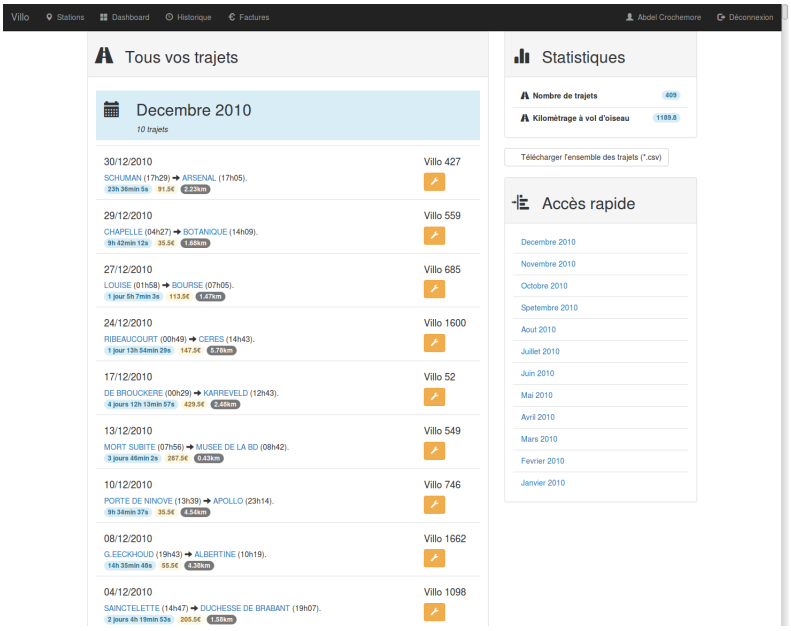

FIGURE 8 – Vue historique

8.8 Tableau de bord

L'utilisateur souhaite accéder immédiatement aux informations importantes, c'est pourquoi lorsqu'il se connecte, il est tout de suite re-dirigé sur un tableau de bord, comprenant, les 10 derniers trajets, ses dépenses du mois, ses informations personnelles, etc ..

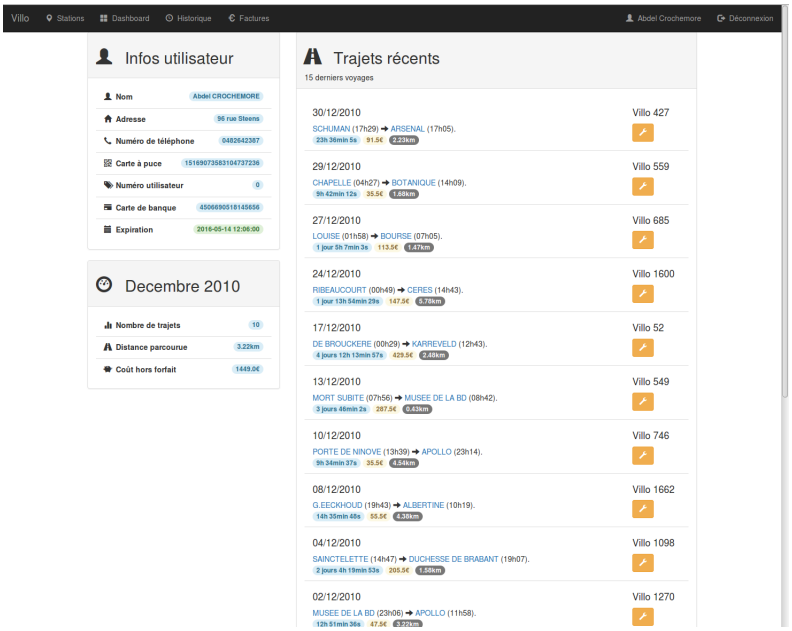

FIGURE 9 – Vue tableau de bord d'un abonné

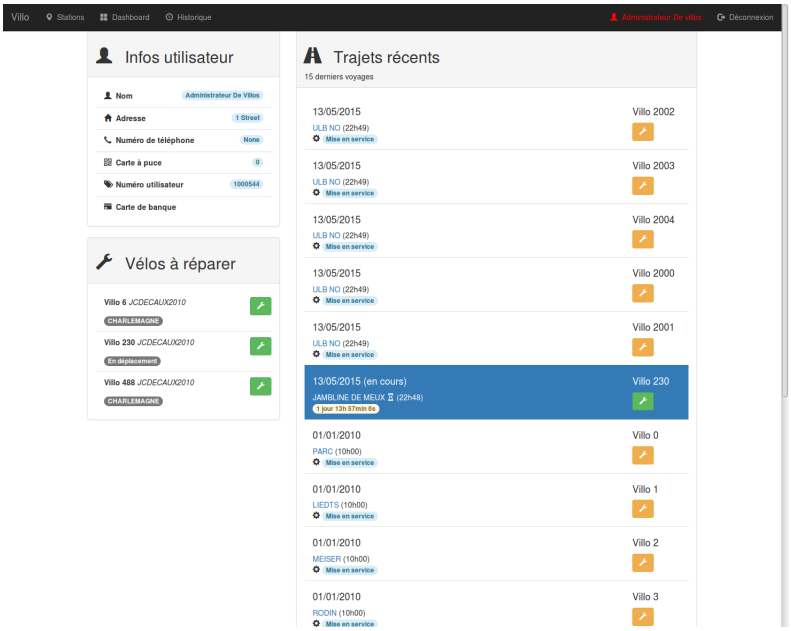

FIGURE 10 – Vue tableau de bord d'un administrateur