

# INFO-H-303 - Bases de Données - Projet Villo!

Florentin Hennecker - Magali Hublet

2014/2015

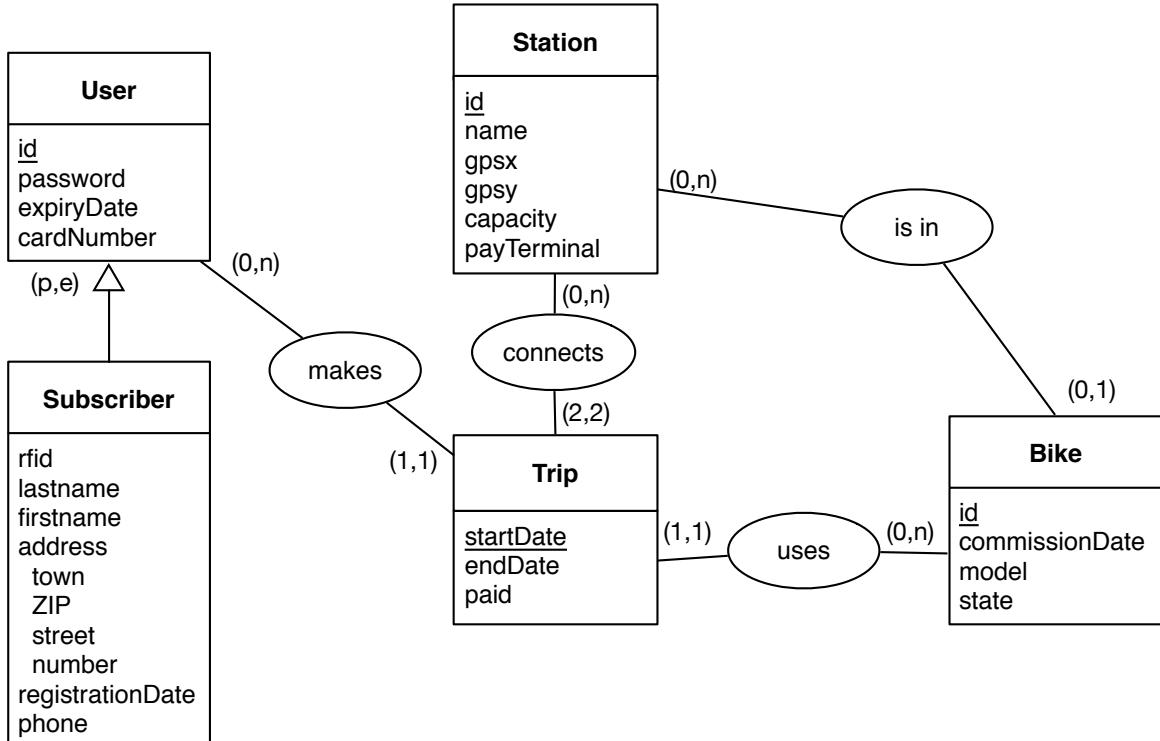
## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Modèle entité-relation</b>	<b>2</b>
2.1	Contraintes d'intégrité . . . . .	2
2.2	Remarques et hypothèses . . . . .	3
<b>3</b>	<b>Modèle relationnel</b>	<b>3</b>
3.1	Contraintes supplémentaires . . . . .	3
<b>4</b>	<b>Script SQL DDL</b>	<b>3</b>
4.1	Notes sur le script SQL DDL . . . . .	4
<b>5</b>	<b>Différentes requêtes</b>	<b>4</b>
5.1	Première requête . . . . .	4
5.2	Deuxième requête . . . . .	5
5.3	Troisième requête . . . . .	5
5.4	Quatrième requête . . . . .	6
5.5	Cinquième requête . . . . .	7
5.6	Sixième requête . . . . .	8
<b>6</b>	<b>Application</b>	<b>8</b>
6.1	Technologies utilisées et architecture . . . . .	8
6.2	Initialisation . . . . .	8
6.3	Fonctionnement . . . . .	9
6.3.1	Utilisateur non connecté . . . . .	9
6.3.2	Enregistrement d'un nouvel utilisateur . . . . .	9
6.3.3	Abonné connecté . . . . .	9
6.3.4	Effectuer un trajet . . . . .	9
6.3.5	Signaler un vélo . . . . .	9
6.4	Apports personnels . . . . .	15
6.4.1	Achat d'un ticket - utilisateur temporaire . . . . .	15
6.4.2	Internationalisation . . . . .	15
6.4.3	Google Maps API . . . . .	15
6.4.4	Dashboard administrateur . . . . .	16
6.4.5	Protection contre l'injection SQL . . . . .	16
6.4.6	Design . . . . .	16
<b>7</b>	<b>Conclusion</b>	<b>16</b>
<b>A</b>	<b>Extension C pour la distance</b>	<b>19</b>

# 1 Introduction

Pour ce projet de bases de données, il nous a été demandé de développer un système de gestion des Vilos pour la ville de Bruxelles, avec une interface graphique. Tous les Bruxellois doivent pouvoir utiliser ce système pour naviguer dans la ville à l'aide des vélos mis à disposition ou consulter diverses informations à propos de leur compte.

## 2 Modèle entité-relation



Notons que nous avons traduit le modèle rendu lors de la première partie pour mieux coller avec notre code. Nous avons également séparé l'ancien champ `gps-coord` en deux champs : `gpsx` et `gpsy`. La clé d'un Trip est `User.id + Trip.startDate`.

### 2.1 Contraintes d'intégrité

- l'heure de départ d'un déplacement est antérieure à l'heure d'arrivée
- un vélo est impliqué dans maximum un déplacement à un moment donné
- la date de mise en service d'un vélo est antérieure à toute date de déplacement qui implique ce vélo
- le nombre de vélos rangés dans une station n'excède pas la capacité de cette station
- la date d'inscription d'un utilisateur est antérieure à tout déplacement fait par celui-ci
- la date d'inscription d'un abonné est antérieure à la date d'expiration de son titre de transport
- la date de départ d'un déplacement est antérieure à la date d'expiration du titre de transport de l'utilisateur qui fait ce déplacement
- le RFID est unique pour chaque abonné
- la capacité d'une station est positive
- un vélo dont l'état spécifie qu'il ne fonctionne pas ne peut pas être utilisé dans un déplacement
- il ne peut exister plus de deux déplacements non payés impliquant le même utilisateur

## 2.2 Remarques et hypothèses

- Un déplacement implique 2 stations, mais ces stations ne sont pas forcément différentes (un utilisateur peut faire une boucle)
- Les tickets de 7 jours et de 24 heures sont valables à partir de la date du premier déplacement
- Les abonnements annuels sont valables à partir du paiement

## 3 Modèle relationnel

- Station(id, name, payTerminal, capacity, gpsx, gpsy)
- Bike(id, commissionDate, model, state, station)
  - Bike.station référence Station.id
- User(id, password, expiryDate, cardNumber)
- Subscriber(id, RFID, lastName, firstName, addressTown, addressZIP, addressStreet, addressNumber, registrationDate, phone)
  - Subscriber.id référence User.id
- Trip(user, startDate, endDate, startstation, endStation, bike, paid)
  - Trip.startStation et Trip.endStation référencent Station.id
  - Trip.bike référence Bike.id

### 3.1 Contraintes supplémentaires

- Bike.station est unique dans la relation Bike
- Subscriber.id référence User.id de manière unique
- Trip.startStation, Trip.endStation et Trip.bike sont uniques dans la relation Déplacement

## 4 Script SQL DDL

```
CREATE TABLE Stations(
    id          INTEGER      NOT NULL,
    name        TEXT,
    payTerminal INTEGER,
    capacity    INTEGER,
    gpsx        REAL,
    gpsy        REAL,
    PRIMARY KEY (id)
);

CREATE TABLE Bikes(
    id          INTEGER      NOT NULL,
    commissionDate TEXT,
    model        TEXT,
    state        TEXT,
    station     INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY (station) REFERENCES Stations(id)
);

CREATE TABLE Users(
    id          INTEGER      NOT NULL,
    password    INTEGER,
    expiryDate  TEXT,
    cardNumber  INTEGER,
```

```

        PRIMARY KEY (id)
);

CREATE TABLE Subscribers(
    id          INTEGER      NOT NULL,
    rfid        INTEGER      UNIQUE,
    lastName    TEXT,
    firstName   TEXT,
    addressTown TEXT,
    addressZIP  TEXT,
    addressStreet TEXT,
    addressNumber TEXT,
    registrationDate TEXT,
    phone       INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Users(id)
);

CREATE TABLE Trips(
    user        INTEGER      NOT NULL,
    startDate   TEXT,
    endDate     TEXT,
    startStation INTEGER,
    endStation  INTEGER,
    bike        INTEGER,
    paid        INTEGER,
    PRIMARY KEY (user, startDate),
    FOREIGN KEY (user) REFERENCES Users(id),
    FOREIGN KEY (startStation) REFERENCES Stations(id),
    FOREIGN KEY (endStation) REFERENCES Stations(id),
    FOREIGN KEY (bike) REFERENCES Bikes(id)
);

```

## 4.1 Notes sur le script SQL DDL

**Types de données** SQLite3 ne supporte pas beaucoup de types de données différents. Il faut donc les choisir en fonction de l'affinité du type de données à stocker pour un des types de base : INTEGER, REAL, TEXT ou BLOB<sup>1</sup>.

**Typage de Bikes.state** Nous avons choisi d'utiliser un champ de type TEXT qui sera NULL si le vélo fonctionne ; et qui contiendra le problème signalé s'il y en a un.

**Attribut Trips.paid** Il s'agit d'un attribut que nous avons rajouté pour faciliter la facturation.

## 5 Différentes requêtes

### 5.1 Première requête

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

---

1. <https://www.sqlite.org/datatype3.html>

## SQL

```
SELECT DISTINCT Subscribers.id, Subscribers.firstName, Subscribers.lastName
FROM Trips
INNER JOIN Subscribers ON Subscribers.id = Trips.user AND Subscribers.addressZIP = "1050"
INNER JOIN Stations ON Stations.id = Trips.startStation AND Stations.name = "FLAGEY";
```

### Algèbre relationnelle

$$\begin{aligned} Subs &\leftarrow \alpha_{id:subid}(Subscribers) \\ TripsFromSubsInXL &\leftarrow Trips \bowtie_{user=subid \text{ and } addressZIP='1050'} Subs \\ FullResult &\leftarrow TripsFromSubsInXL \bowtie_{startStation=id \text{ and } name='FLAGEY'} Stations \\ Result &\leftarrow \pi_{subid}(FullResult) \end{aligned}$$

### Calcul relationnel tuples

$$\begin{aligned} \{sub.id | Subscribers(sub) \wedge sub.addressZIP = '1050' \wedge \exists t(Trips(t) \\ \wedge \exists s(Stations(s) \wedge t.startStation = s.id \wedge s.name = 'FLAGEY'))\} \end{aligned}$$

## 5.2 Deuxième requête

Les utilisateurs ayant utilisé Villo au moins 2 fois.

## SQL

```
SELECT Trips.user, COUNT(*)
FROM Trips
GROUP BY Trips.user
HAVING COUNT(*) >= 2
ORDER BY Trips.user;
```

### Algèbre relationnelle

$$\begin{aligned} TA &\leftarrow \alpha_{user:ua,startDate:sa}(Trips) \\ TB &\leftarrow \alpha_{user:ub,startDate:sb}(Trips) \\ FullResult &\leftarrow TA \bowtie_{ua=ba \text{ and } sa!=sb} TB \\ Result &\leftarrow \pi_{ua}(FullResult) \end{aligned}$$

### Calcul relationnel tuples

$$\{t_1.user | Trips(t_1) \wedge \exists t_2(Trips(t_2) \wedge t_1.user = t_2.user \wedge t_1.startDate \neq t_2.startDate)\}$$

## 5.3 Troisième requête

Les paires d'utilisateurs ayant fait un trajet identique.

## SQL

```

SELECT DISTINCT TA.user, TB.user
From Trips AS TA
INNER JOIN Trips AS TB
ON TA.user > TB.user
AND TA.startStation = TB.startStation
AND TA.endStation = TB.endStation;

```

Le `ON TA.user < TB.user` permet de ne sélectionner qu'une fois une paire. Si on avait utilisé `ON TA.user != TB.user`, on aurait eu les deux permutations à chaque fois.

## Algèbre relationnelle

$$\begin{aligned}
TA &\leftarrow \alpha_{user:ua,startDate:sa,endDate:ea}(Trips) \\
TB &\leftarrow \alpha_{user:ub,startDate:sb,endDate:eb}(Trips) \\
FullResult &\leftarrow TA \bowtie_{ua>ub \text{ and } sa=sb \text{ and } ea=eb} TB \\
Result &\leftarrow \pi_{ua,ub}(FullResult)
\end{aligned}$$

## Calcul relationnel tuples

$$\begin{aligned}
&\{s_1.id, s_2.id | Subscribers(s_1) \wedge Subscribers(s_2) \wedge s_1.id \neq s_2.id \\
&\wedge \exists t_1, t_2(Trips(t_1) \wedge Trips(t_2) \wedge t_1.user = s_1.id \wedge t_2.user = s_2.id \\
&\wedge t_1.startStation = t_2.startStation \wedge t_1.endStation = t_2.endStation)\}
\end{aligned}$$

## 5.4 Quatrième requête

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

## SQL

```

SELECT DISTINCT TA.bike
FROM Trips AS TA
INNER JOIN Trips AS TB
ON TA.bike = TB.bike
AND TA.endDate < TB.startDate
GROUP BY TB.startDate
HAVING TA.endStation != TB.startStation;

```

## Algèbre relationnelle

$$\begin{aligned}
TA &\leftarrow \alpha_{startDate:sda,endStation:esa,bike:ba}(Trips) \\
TB &\leftarrow \alpha_{startDate:sdb,startStation:ssb}(Trips) \\
Joint &\leftarrow TA \bowtie_{sda < sdb} TB \\
Consec &\leftarrow \gamma_{sda}(Joint) \\
Disjoint &\leftarrow \sigma_{esa \neq ssb}(Consec) \\
Result &\leftarrow \pi_{ba}(Disjoint)
\end{aligned}$$

## Calcul relationnel tuples

$$\{b_1.id | Bikes(b_1) \wedge \exists t_1, t_2 (Trips(t_1) \wedge Trips(t_2) \wedge \\ \wedge t_1.startDate < t_2.startDate \wedge \forall t_3 Trips(t_3) \rightarrow (\neg(t_3.startDate \geq t_1.startDate \wedge t_3.startDate \leq t_2.startDate) \wedge \\ \wedge t_1.endStation \neq t_2.startStation)\}$$

## 5.5 Cinquième requête

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

**Remarque :** SQLite3, le système de bases de données que nous avons choisi ne supporte que très peu de fonctions mathématiques. Il a donc fallu écrire une extension C (voir annexe A) pour calculer la distance entre deux points exprimés en coordonnées géographiques. Il suffit ensuite de la compiler en librairie dynamique et de la charger au run-time (`.load ../distance`).

**Remarque bis :** La requête demande de classer les utilisateurs mais de sélectionner également la date d'inscription qui n'est donnée que pour les abonnés (y compris dans les données de départ). On effectue donc la requête uniquement pour les abonnés.

## SQL

```
.load ../distance
SELECT Trips.user,
       Subscribers.registrationDate,
       COUNT(*),
       SUM(distance(SA.gpsx, SA.gpsy, SB.gpsx, SB.gpsy)) AS sum,
       AVG(distance(SA.gpsx, SA.gpsy, SB.gpsx, SB.gpsy)) AS avg
FROM Trips
INNER JOIN Subscribers
        ON Trips.user = Subscribers.id
INNER JOIN Stations AS SA
        ON Trips.startStation = SA.id
INNER JOIN Stations AS SB
        ON Trips.endStation = SB.id
GROUP BY Trips.user
ORDER BY sum DESC;
```

**Sortie de la requête SQL** Au cas où surviendraient des problèmes avec l'extension C, voici les quelques premières lignes des résultats de l'exécution de la requête (la distance est en kilomètres) :

```
1|2010-01-02T02:44:33|370|1207.94934396373|3.26472795665873
0|2010-01-01T10:00:00|409|1190.54228771068|2.91086133914591
2|2010-01-02T12:15:48|354|1124.53287926094|3.17664655158457
3|2010-01-03T05:53:59|354|1084.18212976889|3.06266138352793
5|2010-01-04T06:27:54|325|1005.28951449224|3.09319850612996
4|2010-01-03T17:51:02|305|980.915784946898|3.21611732769475
6|2010-01-04T20:29:00|303|922.234209777715|3.04367725999246
7|2010-01-05T16:13:48|296|911.168348069189|3.07827144617969
8|2010-01-06T11:14:30|283|890.876347993064|3.14797296110623
12|2010-01-08T13:46:05|282|890.682568600884|3.15844882482583
10|2010-01-07T11:59:56|291|890.305205610363|3.05946806051671
17|2010-01-11T13:28:50|253|837.420477326081|3.30996236097265
16|2010-01-11T04:33:00|265|832.237773879973|3.14051990143386
19|2010-01-13T00:07:57|264|824.574666502193|3.12338888826588
11|2010-01-08T01:19:31|271|822.930182199801|3.03664273874465
```

```
9|2010-01-06T23:52:19|266|818.752562810762|3.07801715342392
18|2010-01-12T10:22:17|250|816.105134849112|3.26442053939645
15|2010-01-10T13:17:47|248|802.891594967705|3.23746610874075
```

## 5.6 Sixième requête

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.

### SQL

```
SELECT stationA, bikecount, usercount FROM
    (SELECT endStation AS stationA, COUNT(*) AS bikecount
     FROM Trips AS TA
     GROUP BY endStation
     HAVING COUNT(*) >= 10)
INNER JOIN
    (SELECT endStation AS stationB, COUNT(DISTINCT user) AS usercount
     FROM Trips AS TB
     GROUP BY endStation)
ON stationA = stationB;
```

## 6 Application

### 6.1 Technologies utilisées et architecture

Nous avons utilisé Flask<sup>2</sup>, un serveur web, générateur de pages dynamiques en Python ; couplé avec SQLite3. Python est un langage très rapide en termes de temps de développement et nous a donc permis de prototyper l'application en quelques lignes.

Le choix de SQLite3 s'est avéré agréable au début puisque toute l'application (BDD comprise) était contenue dans un seul dossier ; il était facile de créer et d'accéder à la base de données. Ce choix a présenté un inconvénient lorsqu'il fallut implémenter la requête 5 demandant de calculer une distance. N'ayant à notre disposition ni de racine carrée, ni de fonction trigonométrique, il a fallu chercher une solution un peu exotique (voir annexe A).

Flask gère donc les requêtes HTTP envoyées par les clients (fichier `main.py`), répond en interrogeant la base de données (toutes les requêtes se situent dans le fichier `requests.py`) et en générant une page html à partir de templates `jinja2` (dossier `templates/`).

### 6.2 Initialisation

Le `README.md` est voulu assez complet quant à l'installation des dépendances nécessaires et l'initialisation des données. Voici cependant un petit rappel des commandes à effectuer dans le dossier `src/` :

```
python buildModel.py      # création de la base de données
python importData.py      # importation des données
./lrebuild.sh              # reconstruction des fichiers de langage
./lcompile.sh               # compilation des traductions
python main.py                # lancement du programme
```

---

2. <http://flask.pocoo.org/docs/0.10/>

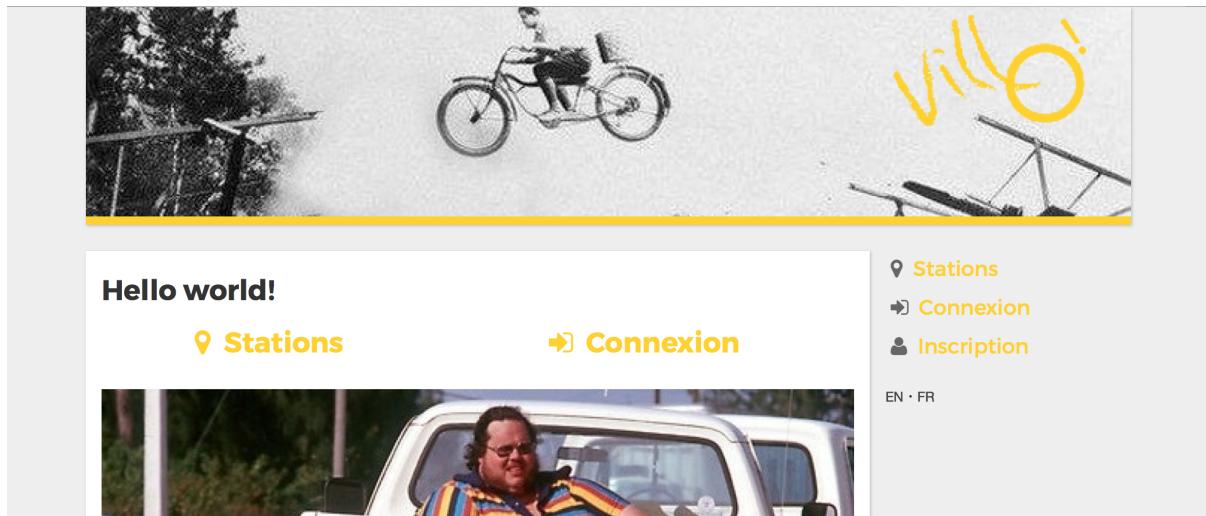


FIGURE 1 – Page d'accueil

### 6.3 Fonctionnement

Toutes les fonctionnalités demandées dans l'énoncé ont été implémentées ; elles ont été accompagnées de différents ajouts facultatifs. La gestion des paiements a bien évidemment été simulée. Certaines fonctionnalités ne seront pas présentées ici, comme la génération aléatoire des RFID pour un souci de clarté.

#### 6.3.1 Utilisateur non connecté

Un utilisateur connecté peut faire plusieurs choses, la première étant d'accéder à la page d'accueil (fig 1). Il peut également consulter la liste des stations (fig 2) sans toutefois avoir la permission de prendre un vélo qui y est déposé (voir boutons grisés en fig 3).

#### 6.3.2 Enregistrement d'un nouvel utilisateur

Enregistrons-nous comme nouvel abonné. Nous sommes face à un formulaire qui est validé autant du côté client (pour produire les messages d'erreur comme en fig 4) que du côté serveur (au cas où un utilisateur malveillant forgerait un POST)<sup>3</sup>. Après avoir rempli le formulaire, nous atterrîsons sur une page qui nous donne notre `userid` (fig 5).

#### 6.3.3 Abonné connecté

Nous nous connectons (fig 6) et nous arrivons sur la page d'accueil qui nous souhaite la bienvenue et qui nous propose plus d'options (fig 7) !

#### 6.3.4 Effectuer un trajet

Prenons le vélo 485 au Cimetière d'Ixelles (fig 8). Nous arrivons sur une page qui nous indique les différentes statistiques de notre trajet courant (fig 9). Un peu plus tard, nous allons voir la page "Mes trajets" et nous pouvons voir le prix que le trajet nous coûtera (fig 10).

#### 6.3.5 Signaler un vélo

Nous avons terminé notre trajet, nous allons déposer mon vélo à Riga. Seulement, il y a eu quelques problèmes avec la selle et il faut les signaler (fig 11). On voit qu'une fois le vélo déposé, il n'est plus accessible (fig 12). Voilà, notre premier trajet est effectué (fig 13) ! Encore heureux : quelques minutes après, la station Riga était pleine (fig 14).

3. Tous les formulaires présents dans l'application sont vérifiés autant du côté client que du côté serveur

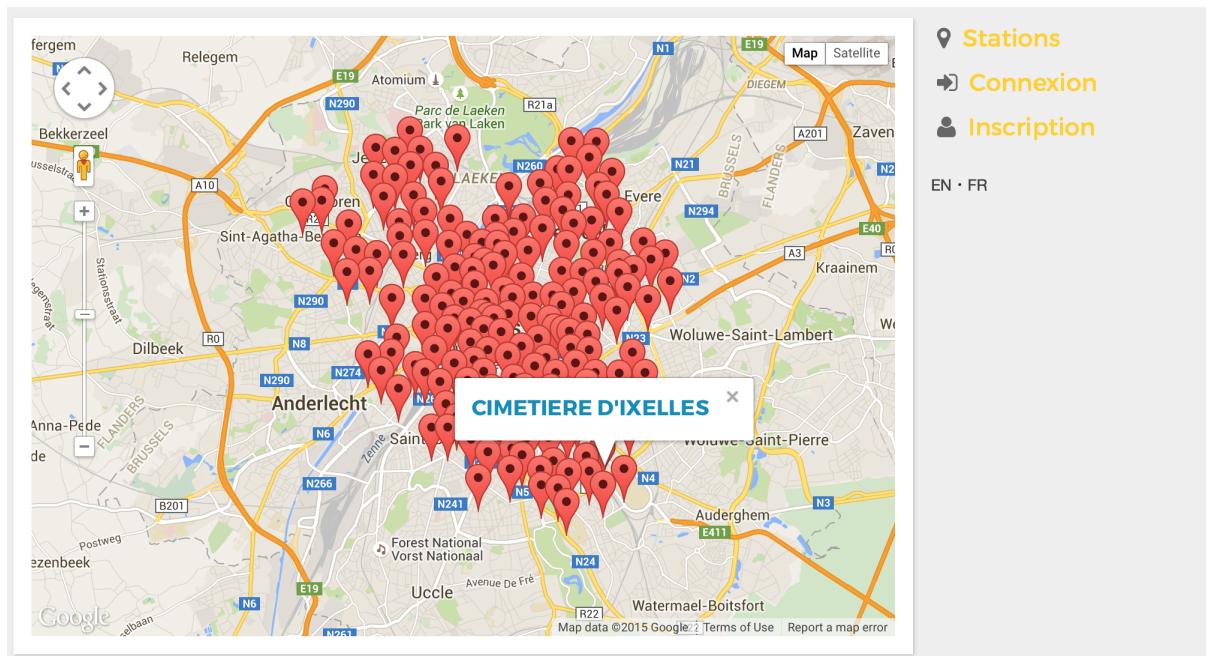


FIGURE 2 – Liste des stations

## Cimetiere d'ixelles

**15 emplacements libres restants**

361	485	498	514
595	709	1074	
1380	1860	1903	

Ticket une journée (1,5€)  Ticket une semaine (7€)

Password (4-digit pin)

Credit card number

**Buy**

EN • FR

FIGURE 3 – Vélos inaccessibles pour un utilisateur non-connecté

Roger

Ladalle

....

azertyuiop|

*Le numéro de téléphone est requis et doit être un nombre à 10 chiffres*

Rue

Numéro

Code postal

Ville

Numéro de carte de crédit

Inscription

FIGURE 4 – Formulaire d'enregistrement

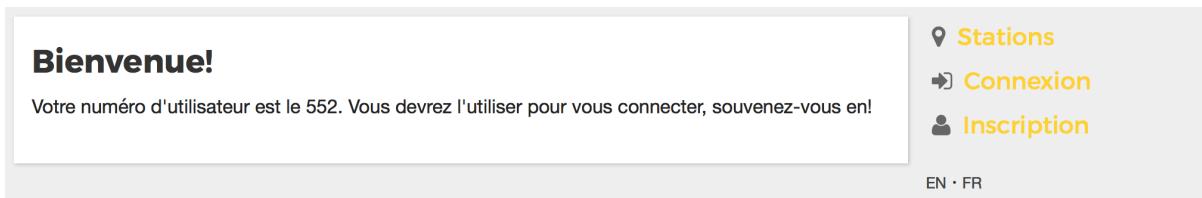


FIGURE 5 – Page de bienvenue

552

....|

Connexion

FIGURE 6 – Formulaire de connexion

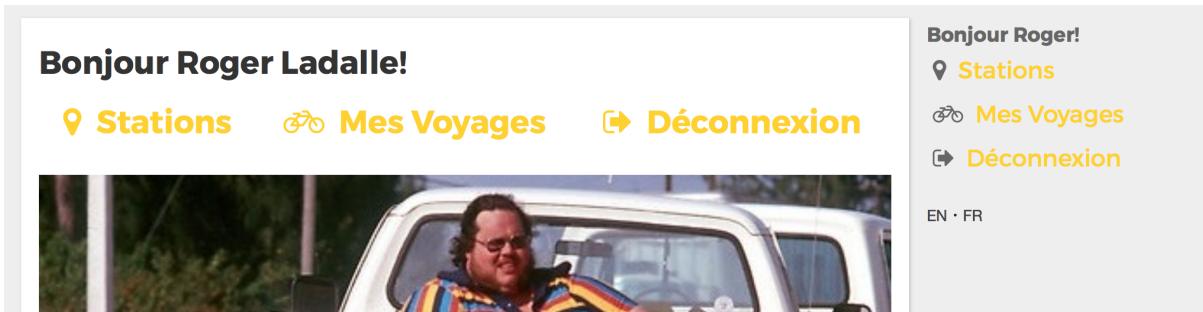


FIGURE 7 – Page d'accueil une fois connecté

This screenshot shows the list of available bike stations at 'Cimetiere d'ixelles'. At the top, it says '15 emplacements libres restants'. Below is a grid of green buttons, each containing a station number followed by a right-pointing arrow: 361 →, 485 →, 498 →, 514 →, 595 →, 709 →, 1074 →, 1380 →, 1860 →, and 1903 →. To the right of the list is a large, stylized yellow 'Ville!' logo. The top right corner of the screen shows the user's name 'Bonjour Roger!', navigation links, and language options.

FIGURE 8 – Nous prenons le vélo 485

This screenshot shows the details of the current trip for bike number 485. It features a large '485' with a bicycle icon above it, and below it, '0 minutes | 0€'. The top right corner shows the user's name 'Bonjour Roger!', navigation links, and language options.

FIGURE 9 – Infos du voyage courant



FIGURE 10 – Mes voyages

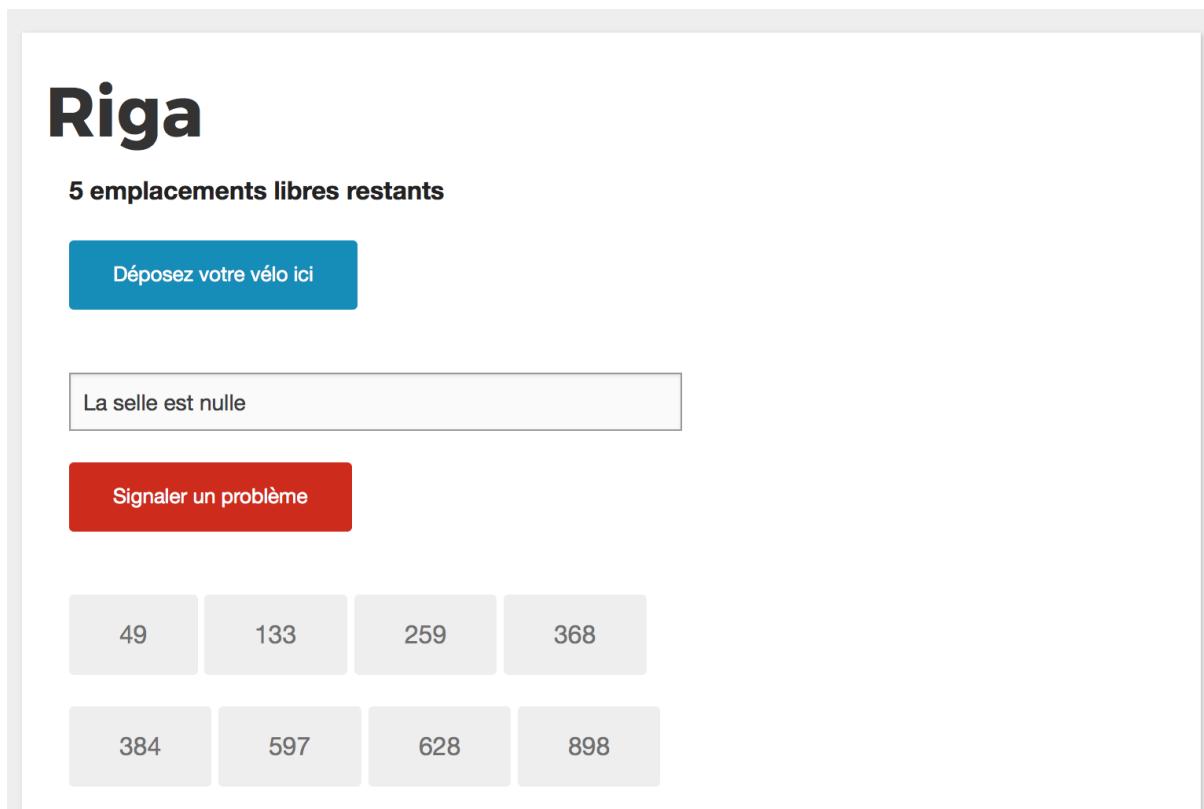


FIGURE 11 – Signaler un vélo

# Riga

4 emplacements libres restants

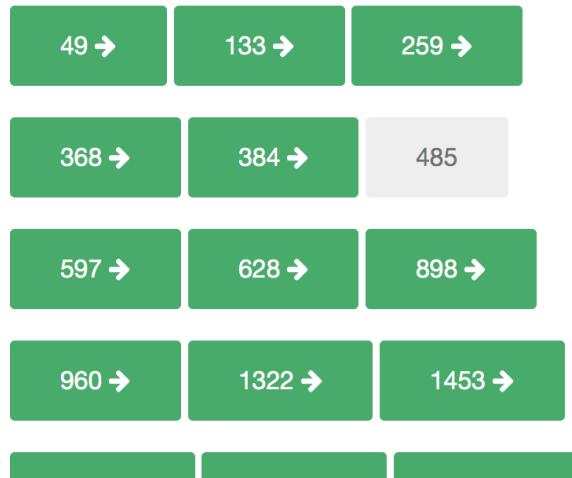


FIGURE 12 – Le vélo signalé n'est plus accessible

## Mes Voyages

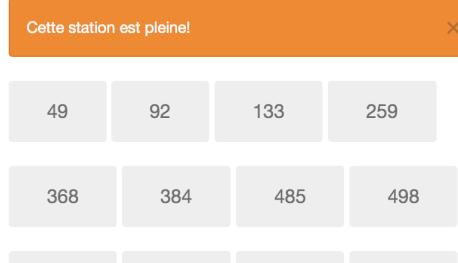
15/05/15 02:12	CIMETIERE D'IXELLES ➔ RIGA	⌚ 159 minutes	€ 7.5€
----------------	----------------------------	---------------	--------

Bonjour Roger!  
📍 Stations  
🚲 Mes Voyages  
➡ Déconnexion  
EN · FR

FIGURE 13 – Premier trajet effectué

# Riga

0 emplacements libres restants



Bonjour Roger!  
📍 Stations  
🚲 Mes Voyages  
➡ Déconnexion  
EN · FR

FIGURE 14 – Une station pleine

FIGURE 15 – Acheter un ticket

FIGURE 16 – Page de bienvenue d'un utilisateur temporaire

## 6.4 Apports personnels

### 6.4.1 Achat d'un ticket - utilisateur temporaire

On a pu le voir plus haut, quand aucun utilisateur n'est connecté et que la station que l'on regarde dispose d'une borne de paiement, on affiche un formulaire d'achat de ticket (fig 15). La procédure est à peu près la même que pour un utilisateur abonné, on va arriver sur une page de bienvenue nous indiquant notre nouveau `ticketid`(fig 16). L'utilisateur a ensuite accès à tous les services Villo !

### 6.4.2 Internationalisation

Nous avons utilisé Flask-Babel<sup>4</sup> pour proposer une version du site en anglais et une autre en français. Quelques scripts sont également fournis pour rajouter d'autres expressions à traduire et des instructions claires sont disponibles autant dans le `README.MD` que sur internet quant à l'ajout de nouveaux langages.

Si on clique sur le bouton "EN" du dessous de la colonne de droite, toute l'application se traduit en anglais (fig 17).

### 6.4.3 Google Maps API

L'affichage des stations se fait sur une carte Google Maps grâce à l'utilisation de l'API javascript publique. Les limitations de cette API gratuite sont suffisamment amples pour un projet de cette échelle.

4. <https://pythonhosted.org/Flask-Babel/>

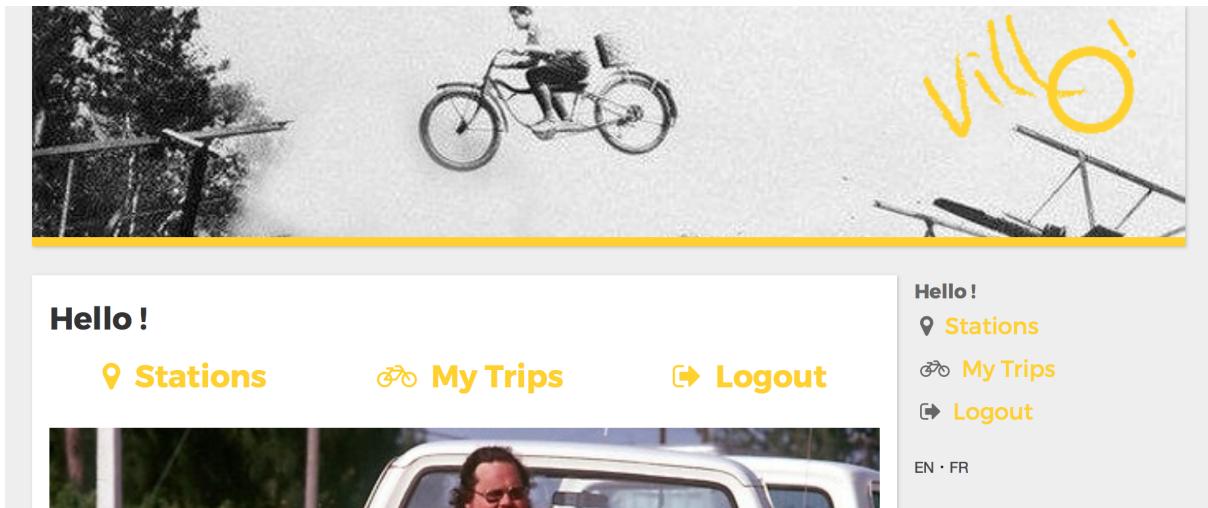


FIGURE 17 – L’application en anglais

#### 6.4.4 Dashboard administrateur

Nous avons décidé de développer un dashboard accessible au chemin `/admin`, de manière non-sécurisée pour des raisons de clarté et d’accessibilité. On peut y voir la liste des vélos cassés et la station où ils sont rangés, les stations remplies et presque remplies ainsi que les stations vides et presque vides. Ce dashboard peut donc donner une vue d’ensemble intéressante sur les problèmes critiques du système à tout administrateur (fig 18). Voici par exemple la requête SQL qui permet de récupérer la liste des stations presque vides :

```
SELECT Stations.*, COUNT(*) AS cnt FROM Bikes
INNER JOIN Stations ON Bikes.station = Stations.id
GROUP BY station
HAVING cnt < 5 and cnt != 0
```

#### 6.4.5 Protection contre l’injection SQL

Toutes les requêtes sont codées de manière à ce que l’entrée d’un utilisateur est assainie. Un utilisateur peut donc se prénommer `"; DROP TABLE Stations` sans aucun souci.

Il suffit en fait, en Python et SQLite3 de passer les paramètres (dans l’exemple ci-dessous, `id`) à une requête de cette manière :

```
c.execute("SELECT * FROM Bikes WHERE id == (?)", (id,))
```

#### 6.4.6 Design

Outre le thème jaune imposé par la ville de Bruxelles, nous avons pensé à rendre le site accessible sur des terminaux mobile. Le design est entièrement *responsive* (fig 19).

## 7 Conclusion

Ce projet fut très enrichissant, notamment grâce à la diversité des techniques utilisées. Il n’était pas difficile de s’y atteler et de rajouter des fonctionnalités par-ci par-là.



**Administration**

**Broken bikes**

Bike id	Station	Issue
485	#68 - RIGA	La selle est nulle

**Full Stations**

#68 - RIGA

**Empty Stations**

No empty stations

**Almost Full Stations**

#68 - RIGA  
#105 - ARTS-LOI  
#169 - PLACE MORICHAIR

**Almost Empty Stations**

#1 - SIMONIS  
#18 - PAGE  
#57 - VAN YSENDUCK/VAN YSENDIJK  
#92 - CLEMENCEAU  
#113 - POELAERT  
#120 - ARCHIMEDE

Bonjour Roger!  
[Stations](#)  
[Mes Voyages](#)  
[Déconnexion](#)

EN · FR

FIGURE 18 – Dashboard administrateur

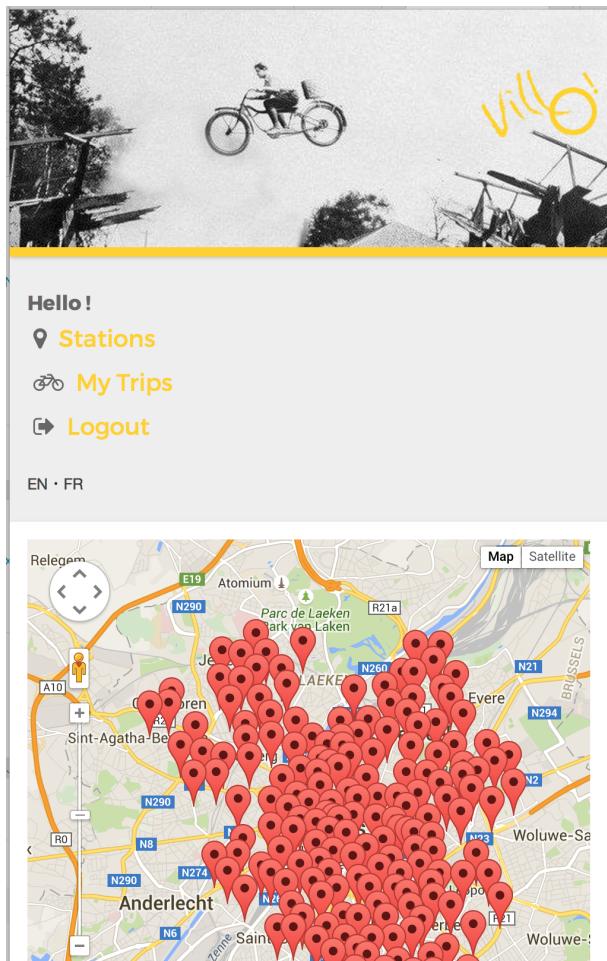


FIGURE 19 – Design responsive

En outre, il nous a permis de découvrir plus en profondeur Flask qui est un framework web en Python extrêmement puissant.

## A Extension C pour la distance

Il est relativement facile de trouver de la documentation sur l'écriture d'extensions pour SQLite3<sup>5</sup>.

```
#include "math.h"
#include <sqlite3ext.h>
#include <stdlib.h>
SQLITE_EXTENSION_INIT1

static void distance(sqlite3_context *context, int argc, sqlite3_value **argv) {
    double gpsy1 = sqlite3_value_double(argv[0]) * M_PI / 180; // lat in radians
    double gpsx1 = sqlite3_value_double(argv[1]) * M_PI / 180; // long in radians
    double gpsy2 = sqlite3_value_double(argv[2]) * M_PI / 180; // lat in radians
    double gpsx2 = sqlite3_value_double(argv[3]) * M_PI / 180; // long in radians
    double r = 6371.0; // earth's radius in kilometers
    double a, c, distance;

    a = pow(sin((gpsx2-gpsx1)/2),2) + pow(sin((gpsy2-gpsy1)/2), 2) \
        * cos(gpsx1) * cos(gpsx2);
    c = 2 * atan2(sqrt(a), sqrt(1-a));
    distance = r*c;
    sqlite3_result_double(context, distance);
}

#endif _WIN32
__declspec(dllexport)
#endif
int sqlite3_distance_init(
    sqlite3 *db,
    char **pzErrMsg,
    const sqlite3_api_routines *pApi
){
    int rc = SQLITE_OK;
    SQLITE_EXTENSION_INIT2(pApi);
    sqlite3_create_function_v2(db, "distance", 4, SQLITE_ANY, NULL, distance, \
        NULL, NULL, NULL);
    return rc;
}
```

---

5. <https://www.sqlite.org/loadext.html>