



Groupe 2 : Qualité

Florence GOURMELON

Cédric BEZY

DEMARCHE DES TESTS EFFECTUES

Janvier 2016



Table des matières

Introduction	3
1. Normes de codage	4
1.1. Général	4
1.2. SQL	5
1.3. Python	6
1.4. JavaScript	6
2. Description des tests	7
2.1. Tests d'efficacité.....	7
2.2. Tests unitaires	7
3. Groupe 3 : Formulaire et fonctionnement	8
3.1. Contenu des formulaires	8
3.2. Validité des attributs	12
3.3. Fonctionnalité de l'application	14
4. Groupe 4 : Facturation.....	18
4.1. Tests pour la tarification	18
5. Groupe 5 : Application Centrale	19
5.1. Tests pour l'application	19
6. Groupe 6 : Attribution	22
6.1. Tests pour l'attribution des courses	22
7. Groupe 7 : Conducteur	23
8. Groupe 8 : Statistique	24
8.1. Vérification des cartes.....	24
9. Groupe 9 : Android.....	29

Introduction

Dans le cadre du projet Taxi SID, une application a été créée pour permettre à l'entreprise Taxi-Capitole de Toulouse de pouvoir gérer leur base de données. Ainsi, les étudiants de SID ont participé à la mise en place de cette application, répartis en 10 groupes.

La création d'une application nécessite qu'elle puisse être utilisée par le client sans problème. Le groupe Qualité est chargé d'effectuer les tests nécessaires afin de garantir que les fonctionnalités de l'application soient opérationnelles. Plus précisément, ce groupe a pour objectif :

- Définir une charte contenant les normes de codage
- Définir une démarche de test pour l'ensemble des groupes
- Mettre en place des tests unitaires ainsi que des tests pour vérifier le bon fonctionnement de l'application
- Prévention des problèmes non anticipés

Ce document relate la démarche des tests appliqués groupe par groupe. Dans un premier temps, un paragraphe relate les normes de codage définies pendant le projet. Ensuite, dans un deuxième temps, un paragraphe expliquera l'intérêt et les objectifs des méthodes de tests utilisés dans le projet, globalement des tests unitaires et des tests de projets.

Enfin, les parties 3 à 9 décrivent les tests effectués respectivement pour chaque groupe effectuant du code, du groupe n°3 au groupe n°9.

Le groupe 3 est chargé de concevoir les formulaires de l'application web. En d'autres termes, il s'agit du corps de l'application. Les groupes 4 à 7 sont rattachés à une partie précise de la base de données concernant l'application. Ces parties sont respectivement la facturation, la base de données générale, l'attribution des courses, et la géolocalisation spatiale. Tous ces groupes utilisent les langages Python, accompagné de langage SQL ou de langage HTML. Le groupe 8 est chargé de réaliser une interface statistique à l'aide de R Shiny. Et finalement, le groupe 9 s'occupe de concevoir une version de l'application destinées aux androïdes à l'aide du langage Java.

1. Normes de codage

Les normes de codage définissent une éthique commune concernant le code contenu dans un programme. En d'autres termes, elles sont là pour éviter que les programmeurs réalisent des codes ayant des structures complètement différentes les uns des autres, ce qui serait ingérable pour tout le monde. En outre, les normes permettent de faire un code lisible, compréhensible, bien structuré.

1.1. Général

- Le code doit être lisible et aéré (utilisation d'espaces et de lignes vides).
- Espaces à côté des opérateurs. Ne faites pas `a=1` mais plutôt `a = 1`.
- Noms de variables **cohérents** et **compréhensibles**.
- Utilisez le trait d'union ("_") pour séparer les mots dans un nom de variable, quand il y en a plusieurs.
- Indentation des structures de contrôles (IF, FOR, WHILE, etc.) avec des tabulations.
- **Commentaires de code :**
 - faire apparaître clairement l'objectif du code
 - Détails : commentaires pour chaque ligne ou groupe de ligne du code. On doit pouvoir comprendre ce qu'il se passe dans chaque "paragraphe".
 - Commentaires situés au-dessus du code (de préférence)

⇒ Ainsi le code pourra être compris rapidement.

EXEMPLE :

```
def mean(liste):
    """
    Fonction renvoyant la moyenne des valeurs contenu dans 'liste'
    """
    ... (instruction 1)
    ... (instruction 2)

# variable note
notes = [10, 15, 14, 8]

# moyenne des notes
a = mean(notes)
moyenne = mean(notes)

✗ ' "a" n'est pas un nom de variable comprehensible '
✗ ' "moyenne" est un nom trop vague. On sait que c'est une moyenne, mais
de quoi ? Bonne question '

# moyenne des notes
var = mean(notes)

✗ ' "var" est incohérent. On pourrait penser a "variable" ou "variance",
rien a voir avec une moyenne '

# moyenne des notes
moyenne_notes = mean(notes)

✓ ' "moyenne_notes" est compréhensible et coherent avec le resultat ' ☺
```

1.2. SQL

- Utilisez des alias pour les tables et les vues dans les requêtes.
- **Ecrivez les *keywords* en MAJUSCULES** (SELECT, FROM, WHERE).
- Ecrivez le nom des paramètres, variables, tables, vues, curseur, trigger, etc. en minuscules.
- **Pas de ponctuation ni d'accentuation**
- Utilisez le trait d'union ("_") pour séparer les mots dans un nom de variable, quand il y en a plusieurs.
- Privilégiez les jointures aux requêtes imbriquées. Par exemple (ci-dessous), la première requête est préférable à la seconde.

Exemple :

```
-- table_alpha (a1, a2, ..., an)
-- table_beta (b1, b2, ..., bm)

-- a1 et b1 : attributs de meme origine

-- Selectionner les valeurs de a2 tel que b2 = 5.
```

1ère Requete (avec jointure) :

```
SELECT table_alpha.a2
FROM table_alpha, table_beta
WHERE table_alpha.a1 = table_beta.b1 AND table_beta.b2 = 5
```

2ème Requete (requêtes imbriquées) :

```
SELECT a2
FROM table_alpha
WHERE a1 IN (
    SELECT b1
    FROM table_beta
    WHERE b2 = 5
)
```

1.3. Python

- Préférez l'utilisation de la **programmation fonctionnelle**. En effet, l'encapsulation de fonctions a des avantages considérables :
 - Le code est réutilisable et on évite les copier/coller (redondances).
 - Le déroulement du script est plus lisible.
- Ajoutez un commentaire pour expliquer une fonction après l'avoir définie. Soyez clair et écrivez des phrases courtes mais complètes.
- Privilégiez l'utilisation de noms de variables parlants qui rendent le code lisible. Une fonction lisible ne devrait même pas avoir besoin de commentaires supplémentaires.

1.4. JavaScript

- Pour JavaScript, se baser sur tout ce qui a été cité précédemment.
- Convention : nommer les constructeurs (fonctions) avec une majuscule : ça facilite la lecture du code.

2. Description des tests

2.1. Tests de fonctionnalité

Les tests de fonctionnalités nécessitent de contrôler si l'application est bien conforme à ce qui apparaît dans le code, et si ce qui apparaît dans le code apparaît bien sur les pages de l'application. Cela permet de garantir une application qui fonctionne correctement.

On définit une liste d'items qui doivent impérativement être vérifiés. Les items correspondent à des problèmes qui ne doivent pas apparaître. Imaginons un utilisateur qui chercherait à se connecter à un autre compte que le sien, il ne faut pas qu'il puisse se connecter avec n'importe quel mot de passe. Plus concrètement, il s'agit d'une anticipation des problèmes qui gêneraient l'application.

Les groupes 3 et 9 sont pleinement concernés par ces tests, vu que leurs missions sont de configurer respectivement les formulaires de l'application et l'application pour Android. Le groupe 8, qui configure une interface statistique, est également concerné par ce test.

2.2. Tests unitaires

Les tests unitaires consistent à vérifier que les fonctions programmées fonctionnent correctement. Dans le cas contraire, ils permettent de repérer rapidement des erreurs.

Le code conçu doit pouvoir évoluer avec le temps. Il doit donc marcher même si le programme évolue. Les tests unitaires sont utilisés dans ces cas.

Sous Python, la vérification des fonctions est utilisée avec la fonction `assert`. Voici grosso modo un exemple de tests sur une fonction "addition" :

```
def addition(a, b):  
    return a + b  
  
assert addition(1, 1) == 2      # test de l'addition  
assert addition(1, -1) == 0    # test avec chiffre négatif  
assert addition(4, 2) == 6     # test avec autre chose que des 1  
assert addition(4.5, 2) == 6.5 # test avec des floats
```

Les tests unitaires concernent les groupes 4 à 7 pour les fonctions qui serviront pour l'application, ainsi que pour l'interface statistique du groupe 8.

3. Groupe 3 : Formulaires et fonctionnement

Le test des formulaires consiste à vérifier que le formulaire est bien conforme à ce qui est attendu. Pour cela :

- Chaque page du formulaire doit correspondre au contenu du module correspondant.
- Les pages sont cohérentes entre elles et s'enchainent comme convenu (bouton validation, utilisateur connecté ou non, etc.)

3.1. Contenu des formulaires

L'objectif de cette sous-section est de vérifier que l'affichage des pages correspond au code contenu dans chaque module. Ce qui signifie que chaque objet (zone de texte, bouton) apparaissant sur la page doit être relié à une partie du code. Dans le détail, il est nécessaire de vérifier que l'objet associé à chaque attribut est correct. Parallèlement, on vérifiera que ce qui est contenu dans le code apparaît bien sur la page web (plus précisément, il s'agit de vérifier s'il n'y a pas de code inutilisé). On ne se préoccupe pas de l'aspect dynamique, mais uniquement du contenu.

3.1.1. reservation.py

Le module *reservation.py* contient la classe regroupant les attributs nécessaires pour la réservation (date, lieu de départ et d'arrivée, etc.). Dans l'application, il est divisé en deux pages web :

- la première page est une page contenant les informations principales, concernant l'identité de l'utilisateur et les caractéristiques de la réservation. Il faut prendre en compte le cas où l'utilisateur est connecté (et donc l'identité déjà définie, voir Figure 1) de celui où il n'a pas créé de compte (nécessitant des attributs supplémentaires, voir Figure 2).
- La deuxième page contient les informations complémentaires de la réservation. (Figure 3).

Réserver un taxi

Bonjour Florence!

Utilisateur connecté.

Itinéraire

Adresse de départ

Date et horaire de départ

àHeure

Adresse d'arrivée


CapitoleTAXI
☎ 05 34 250 250
170 taxis, le plus grand groupe de Toulouse et Midi-Pyrénées
 170 taxis

Figure 1 – Page de réservation pour un utilisateur connecté

Réserver un taxi

Utilisateur non-connecté.

Informations utilisateur

Itinéraire

Adresse de départ

Date et horaire de départ

àHeure

Adresse d'arrivée

Figure 2 – Page de réservation pour un utilisateur non connecté

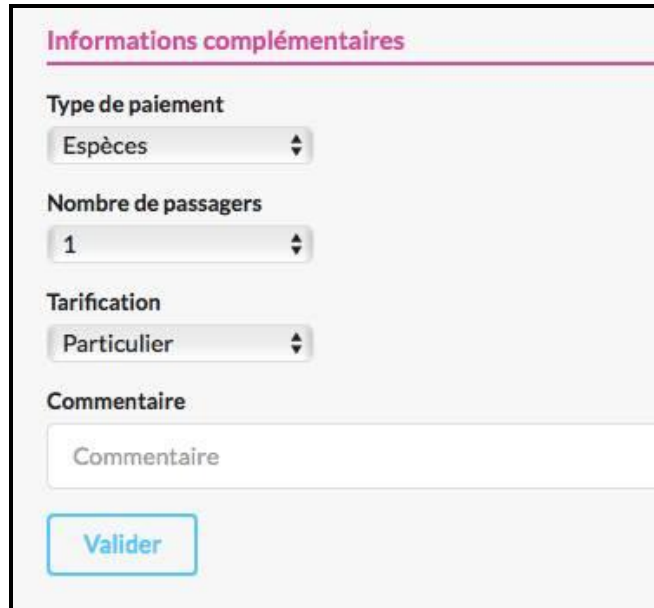


Figure 3 – Page d'informations complémentaires pour un trajet

Le test de vérification est valide :

- ✓ le contenu des pages web de l'application est vérifié, le type d'objet de chaque attribut correspond bien à ce qui est indiqué dans le code.
- ✓ le contenu du code apparaît bien sur les pages de l'application (il n'y a pas de code inutile).

3.1.2. utilisateur.py

Le module *utilisateur.py* est composé de plusieurs classes, liés aux comptes d'utilisateurs. Tout comme pour le module *reservation.py*, on vérifiera que les pages web correspondent bien au code, et que le code correspond également à la page.

- enregistrement d'un utilisateur (Figure 4)
- connexion de l'utilisateur si celui-ci est déjà enregistré (voir Figure 5),
- mot de passe oublié (Figure 6)
- réinitialisation du mot de passe.

Tout comme pour le module *reservation.py*, le test effectué ici consiste surtout à vérifier que l'affichage des pages correspond bien au code contenu dans les modules.

S'enregistrer

Prénom

Nom

Numéro de téléphone

Adresse email

Ville

Code postal

Adresse

Numéro

Mot de passe

Confirmer le mot de passe

Valider

Figure 4 – Page web d'enregistrement pour un utilisateur non inscrit

Se connecter

Adresse email

Mot de passe

Valider [Mot de passe oublié](#)

Figure 5 – Page web de connexion pour les utilisateurs inscrits

Entrez votre adresse email pour que nous puissions vous envoyer un lien de réinitialisation de votre mot de passe

Adresse email

Valider

Figure 6 – Page web pour l'oubli des mots de passe

*** réinitialisation mot de passe ***

✉ Vérifiez vos mails pour confirmer votre adresse email.

*** modification des données d'utilisateur ***

Le test de vérification est valide :

- ✓ le contenu des pages web de l'application est vérifié, le type d'objet de chaque attribut correspond bien à ce qui est indiqué dans le code.
- ✓ le contenu du code apparaît bien sur les pages de l'application (il n'y a pas de code inutile).

3.2. Validité des attributs

Le module *validateur.py* contient plusieurs classes de fonctions, qui permettent de contrôler la cohérence des données rentrées par un utilisateur pas toujours rationnel :

- **Unique** : pour contrôler qu'un attribut est unique (identifiants : téléphone / adresse mail)
- **AdresseValide** : pour contrôler qu'une adresse postale est valide.
- **DateValide** : pour contrôler qu'une date est cohérente.

3.2.1. Unicité des identifiants

La classe "*Unique*" contient des fonctions pour vérifier l'unicité de certains paramètres, notamment des identifiants, tels que le numéro de téléphone ou l'adresse mail.

Les tests suivants ont été effectués :

- ✓ Enregistrement avec le numéro de téléphone d'un compte déjà existant :
→ apparition du message d'erreur adéquat : "Numéro de Téléphone déjà utilisé"
- ✓ Enregistrement avec une adresse e-mail déjà existante.
→ apparition du message d'erreur adéquat : "Adresse Mail déjà utilisée"

✗ **ATTENTION** : les problèmes ci-dessus génèrent les bons messages d'erreurs, mais leur apparition joue sur la lisibilité : les zones de texte en question sont toutes petites, et c'est pénible pour l'utilisateur.

3.2.2. Authenticité de l'adresse

La classe "*AdresseValide*" vérifie qu'une adresse postale rentrée par l'utilisateur existe bien. Une adresse est valide si elle est géolocalisée. Les adresses postales sont utilisées lors de l'enregistrement d'un utilisateur, ainsi que lors de la réservation d'un taxi.

Les tests suivants ont été effectués :

- ✓ Enregistrement avec une adresse postale valide :
→ OK
- ✓ Enregistrement avec une adresse postale invalide, qui n'existe pas :
→ apparition du message d'erreur adéquat : "Cette adresse n'est pas reconnue".
- ✓ Demande de trajet avec une adresse postale valide :
→ OK
- ✓ Demande de trajet avec une adresse postale invalide :
→ apparition du message d'erreur adéquat : "Cette adresse n'est pas reconnue".
- ? Utilisation d'un code postal incohérent.
→ L'adresse est quand même trouvée, erreur ou pas ?

3.2.3. Authenticité de la date

La classe "*DateValide*" vérifie qu'une date est cohérente. Les réservations sont effectuées pour le futur ; ça n'a aucun sens de réserver pour la veille.

Les tests suivants ont été effectués :

- ✓ Réservation avec une date antérieure au moment de la réservation :
 - apparition du message d'erreur adéquat : "La date de réservation ne peut pas être antérieure à la date actuelle."
- ✓ Réservation avec une date postérieure au moment de la réservation :
 - OK

3.3. Fonctionnalité de l'application

Cette sous-section gère les contrôles nécessaires pour le bon fonctionnement de l'application. Ces contrôles sont primordiaux, car ça concerne la base d'une application digne de ce nom.

3.3.1. Inscriptions

Contrôle et anticipation des problèmes liés à l'inscription

- ✓ **Inscription : les champs sont bien renseignés (nom, prénom, ville...)**
 - Lorsqu'un champ n'est pas renseigné, le message d'erreur adéquat apparaît : *"The field is required"*.
- ✗ **Inscription avec un e-mail valide**
 - Une inscription peut être effectuée avec n'importe quel e-mail, valide ou non, du moment qu'il n'est pas déjà utilisé.
 - Est-ce du ressort des fournisseurs ?
- ✓ **Inscription : un mail de confirmation est envoyé à l'adresse indiquée.**
 - OK, le mail est bien reçu. La confirmation est opérationnelle.

3.3.2. Connexion

Contrôles et anticipation des problèmes liés à la connexion des utilisateurs.

- ✓ **Connexion avec une adresse mail non renseignée**
 - Message d'erreur adéquat : "Cette adresse mail n'est associée à aucun compte d'utilisateur"
- ✗ **Connexion : l'utilisateur doit avoir activé son compte (mail de confirmation)**
 - L'utilisateur peut se connecter sans avoir activé son compte
- ✗ **Connexion avec un mauvais mot de passe.**
 - On peut se connecter avec n'importe quel mot de passe !!!

3.3.3. Mots de Passe : Initialisation et Oubli

Contrôles et anticipation des problèmes liés aux mots de passe (autres que l'unicité d'un mot de passe liés à un compte : voir sous-section précédente 3.3.2).

- ✓ **Initialisation du mot de passe : la confirmation est identique**
 - ➔ Apparition du message d'erreur adéquat : "*Les mots de passe doivent être identiques*"
 - ✗ **ATTENTION** : avec l'apparition du message d'erreur, le cadre pour le mot de passe devient très petit, masqué par l'erreur.
- ✓ **Mot de passe oublié : un mail est bien envoyé à l'adresse**
 - ➔ Le mail est reçu, et permet de réinitialiser le mot de passe
- ? **Réinitialisation du mot de passe opérationnelle**
 - ➔ Non testé, puisqu'il faut d'abord régler le problème du mot de passe unique.

3.3.4. Informations de l'utilisateur

Contrôles et anticipation des problèmes liés aux informations de l'utilisateur : peut-il y accéder ? Les modifier ?

- ✓ **Informations : l'utilisateur peut accéder à ses informations**
 - ➔ OK
- ✓ **Informations : l'utilisateur peut modifier ses informations**
 - ➔ OK pour les informations générales
- ✗ **Informations : l'utilisateur peut modifier son mot de passe**
 - ➔ Pas de modification du mot de passe quand l'utilisateur est connecté.

3.3.5. Réservations

Tests liés au formulaire de réservation.

- ✓ **Réservations : les champs sont renseignés**
 - ➔ Lorsqu'un champ n'est pas renseigné, le message d'erreur adéquat apparaît : "*The field is required*"

Les autres tests liés à la réservation sont effectués en détail par les groupes 4 à 6.

3.3.6. Déconnexion

✓ **Déconnexion**
→ OK, l'utilisateur est bien déconnecté

3.3.7. Autre cas à prendre en compte

Problème : Supposons que je suis un troll, je peux m'inscrire plusieurs fois, avec une fausse adresse. (Malheureusement, y en a qui n'ont que ça à faire de leurs journées). Par exemple :

'06.01.02.03.04' / 'trollage1@fausse.adresse'

'06.06.06.06.06' / 'trollage2@fausse.adresse'

En effet, la confirmation s'effectue par mail et non par numéro de téléphone. Idem pour la récupération du mot de passe !

Ainsi, si le vrai possesseur d'un de ces numéros de téléphone veut effectuer une réservation, il se heurte à une impossibilité de le faire. Il doit se connecter, mais il ne connaît pas son mot de passe. Il ne peut pas le réinitialiser, puisque l'adresse mail est inexistante. Ainsi, Taxi Capitole perd un client pour chaque troll de ce genre.

4. Groupe 4 : Facturation

4.1. Tests pour la tarification

Taxi-Capitole admet plusieurs tarifications qui dépendent de nombreux choix.

4.1.1. Fonction *type_tarif*

- Type de tarification :
 - ✓ De jour ou de nuit.
 - ✓ Dimanche / jour fériés

La fonction *type_tarif*, ayant pour paramètres les caractéristiques de la réservation, renvoie le type de tarification associée à cette réservation : A, B, C ou D. Dans l'application, l'aller-retour n'est pas pris en compte, on ne tiendra donc que des catégories C et D.

Le test consiste à définir une grille pour l'ensemble des cas possibles en prenant soin de prédire le résultat prévu pour chaque cas, d'appliquer la fonction pour chacun de ces cas, puis de comparer les résultats des fonctions aux prédictions.

Ce travail a l'inconvénient d'être long car il y a – **nbCas** – cas différents à prendre en compte, mais garantit la robustesse de la fonction.

4.1.2. Fonction *calcul_tarif*

- Type de tarification (liée avec le coût au km)
- Le déplacement est-il d'ordre professionnel / particulier ?
- Kilométrage
- Tarif minimum
- Heures d'attente
- Suppléments Gare / Aéroport
- Suppléments animaux
- Supplément bagages

A partir des paramètres ci-dessus, la fonction *calcul_tarif* calcule le montant du tarif pour une certaine course.

Tout comme le test précédent, le test pour cette fonction consiste à définir une grille pour l'ensemble des cas possibles en prenant soin de prédire le résultat prévu pour chaque cas, d'appliquer la fonction pour chacun de ces cas, puis de comparer les résultats des fonctions aux prédictions.

5. Groupe 5 : Application Centrale

5.1. Tests pour l'application

5.1.1. Bannissement d'un individu

Petit rappel : un bannissement se fait à partir du numéro de téléphone et contient la date de début, la date de fin, ainsi que le motif.

On bannit d'abord un utilisateur (insertion dans la liste des bannis). Puis on essaye de lui ajouter une course.

Test du trigger Insert_Update_Courses

```
INSERT INTO banissements VALUES( '33628251338', '05/01/2016', '06/01/2016', 'Impolitesse')
```

```
INSERT INTO courses(utilisateur, conducteur, places, priorite, debut, fin, retour, commentaire, depart, arrivee)
VALUES ('33628251338', '33699428430', 4, 'high', '2016-01-05 02:03:04.3256', '2016-01-05 04:03:04.3256', FALSE, 'Haha', 1,2)
```

```
ERREUR: l'utilisateur est actuellement banni
***** Erreur *****
```

```
ERREUR: l'utilisateur est actuellement banni
État SQL :P0001
```

- ✓ Le test est réussi : lorsqu'un utilisateur est banni, il ne pourra pas réserver de taxi, qu'il soit inscrit ou non... (à moins de changer de numéro de téléphone / d'adresse mail).

5.1.2. Habitudes

On ajoute un utilisateur. On lui ajoute une course (qui sera une habitude)

S'il fait une autre course avec la même arrivée, il a déjà cette habitude.

Test du trigger Insert_Courses

```
INSERT INTO utilisateurs(telephone) VALUES ('33221144556');

INSERT INTO courses(utilisateur, conducteur, places, priorite, debut, fin,
retour, commentaire, depart, arrivee) VALUES ('33221144556','33699428430',
4, 'high', '2016-01-07 02:03:04.3256', '2016-01-07 04:03:04.3256', FALSE,
'Haha', 1, 2);
-- Que se passe-t-il ?

INSERT INTO courses(utilisateur, conducteur, places, priorite, debut, fin,
retour, commentaire, depart, arrivee) VALUES ('33221144556','33699428430',
4, 'high', '2016-01-06 02:03:04.3256', '2016-01-06 04:03:04.3256', FALSE,
'Yeah', 10, 2);
```

- ✓ Le test est réussi : l'habitude a déjà été insérée une fois, et n'est pas insérée une deuxième fois.

5.1.3. Trigger SupprCourse

Un utilisateur réserve deux courses différentes (ex : aller-retour). On souhaite supprimer la première course une fois qu'elle est effectuée (ainsi que sa facture), mais la course suivante ne doit pas être modifiée ou supprimée.

Test Trigger SupprCourse

```
-- Suppression de la date de fin de la première course

UPDATE courses
SET fin = NULL
WHERE numero = 1;

-- Résultat : La première course est supprimée et la facture correspondante est
supprimée aussi

-- Suppression de la date de fin de la seconde course

UPDATE courses
SET fin = NULL
WHERE numero = 2;

-- Résultat : La seconde course n'est pas supprimée et la facture correspondante
n'est pas supprimée

-- Test ok (réalisé le 2016-01-06 à 11:25)
```

- ✓ Le test est réussi : la course effectuée par le client est effacée de la table, mais les courses à venir par le même client sont conservées.

5.1.4. Trigger verifheuredepart

On souhaite qu'il y ait cohérence entre l'heure de réservation et l'heure de début de la course. C'est à dire que l'heure de début de la course soit plus récente que l'heure de réservation.

Test trigger verifheuredepart

-- Insertion d'une course avec l'heure de début inférieure à l'heure de l'insertion

```
INSERT INTO courses(utilisateur, conducteur, places, priorite, debut, fin,
retour, commentaire, depart, arrivee) VALUES ('33221144556','33699428430',
4, 'high', '2016-01-04 02:03:04.3256', '2016-01-04 04:03:04.3256', FALSE,
'Haha', 1,10)
```

ERREUR: L'heure de depart de la course doit être postérieure à l'heure de la commande

***** Erreur *****

ERREUR: L'heure de depart de la course doit être postérieure à l'heure de la commande

État SQL :P0001

- ✓ On voit qu'effectivement si l'heure de début de course est plus ancienne ou égale à l'heure de réservation, il y a erreur. Le test est donc valide.

6. Groupe 6 : Attribution

6.1. Tests pour l'attribution des courses

6.1.1. Fonction conducteur_dispo_ordonnee

Cette fonction permet d'ordonner les conducteurs libres en fonction des paramètres suivants :

- Une liste de stations
- Le nombre de places du véhicule

Test type :

```
def test_conducteur_dispo_ordonnee():  
    liste = ['Balma', 'Capitole']  
    a = conducteur_dispo_ordonnee.ordonnancement(liste, 3)  
    assert a == [('33659854658',), ('33656892345',,)]
```

6.1.2. Fonction convert_date

Cette fonction convertit la date rentrée en paramètre en timestamp (temps en secondes depuis le 1^{er} janvier 1790)

Test de la fonction :

```
def test_convert_date(self):  
    date="2016-07-01 19:30:00"  
    assert utile.convert_date(date, date_format = "%Y-%m-%d %H:%M:%S") ==  
    1467394200.0
```

7. Groupe 7 : Conducteur

8. Groupe 8 : Statistique

Le groupe Statistique travaille sous R-studio, pour faire une interface statistique avec Shiny, (extension de R-studio). L'interface est divisée en 4 parties :

- Représentation spatiale des conducteurs (utilisation de cartes)
- Représentation spatiale des clients (utilisation de cartes)
- Représentation spatiale des stations, avec coloration en fonction du nombre de taxi.
- Analyse statistique

Compte tenu du fait que les données ne soient pas mises à disposition, des données ont été simulées afin d'effectuer les tests.

✖ Il faudra simuler le programme sur des données différentes en taille et en contenu, afin d'améliorer la qualité des tests, et garantir que le programme fonctionne bien !

8.1. Vérification des cartes

Le test consiste à vérifier que les cartes soient opérationnelles. On en profitera pour vérifier également que le programme s'adapte à la base de données : par exemple si on la diminue ou si on l'augmente. Ce test consiste surtout à montrer que le programme n'est pas figé à une certaine base de données.

Plus concrètement, voici les points à vérifier :

- Le contenu de la base de données est représenté correctement.
- La mise en forme : couleurs et formes des points.
- Que se passe-t-il en modifiant les tuples représentés ? Il ne faut pas que le programme soit figé à une base de données précise.

8.1.1. Représentation des conducteurs

La Figure 7 est une carte qui géolocalise l'emplacement de tous les conducteurs de la base de données, à partir de la latitude et de la longitude.

La Figure 8 est une carte qui géolocalise seulement 3 tuples pris au hasard dans la base de données, afin de vérifier que le programme s'adapte en fonction de la base de données (test d'adaptation). En effet, le programme ne doit pas être figé à une base particulière.

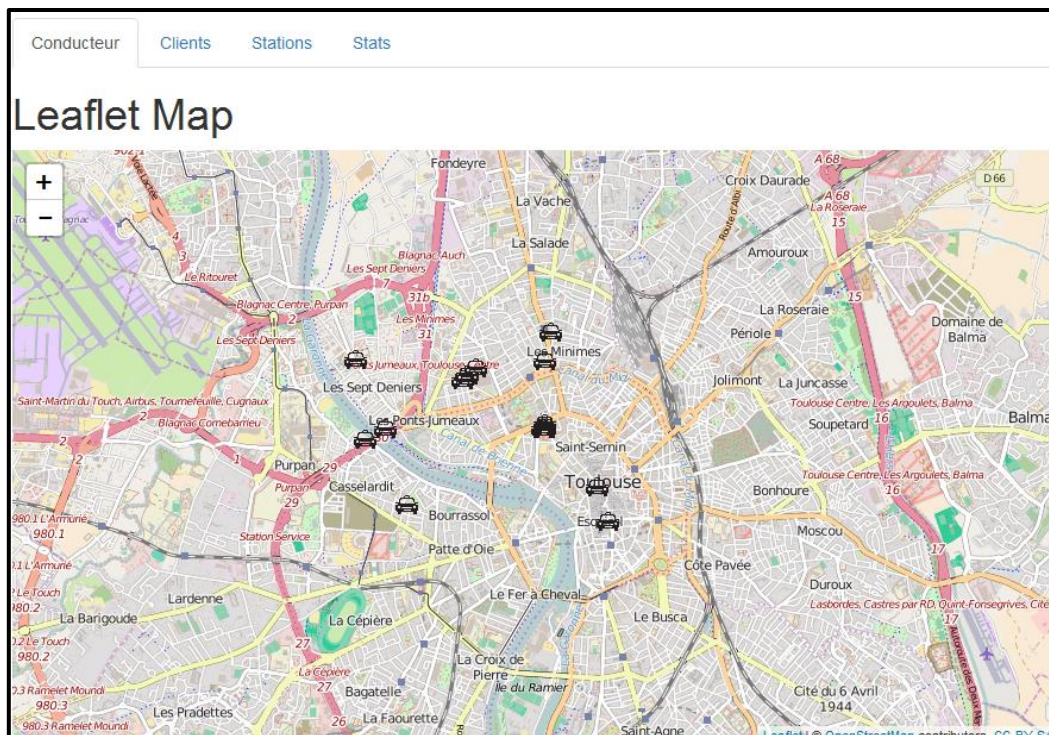


Figure 7 – Carte montrant la localisation de tous les taxis contenus dans la BD

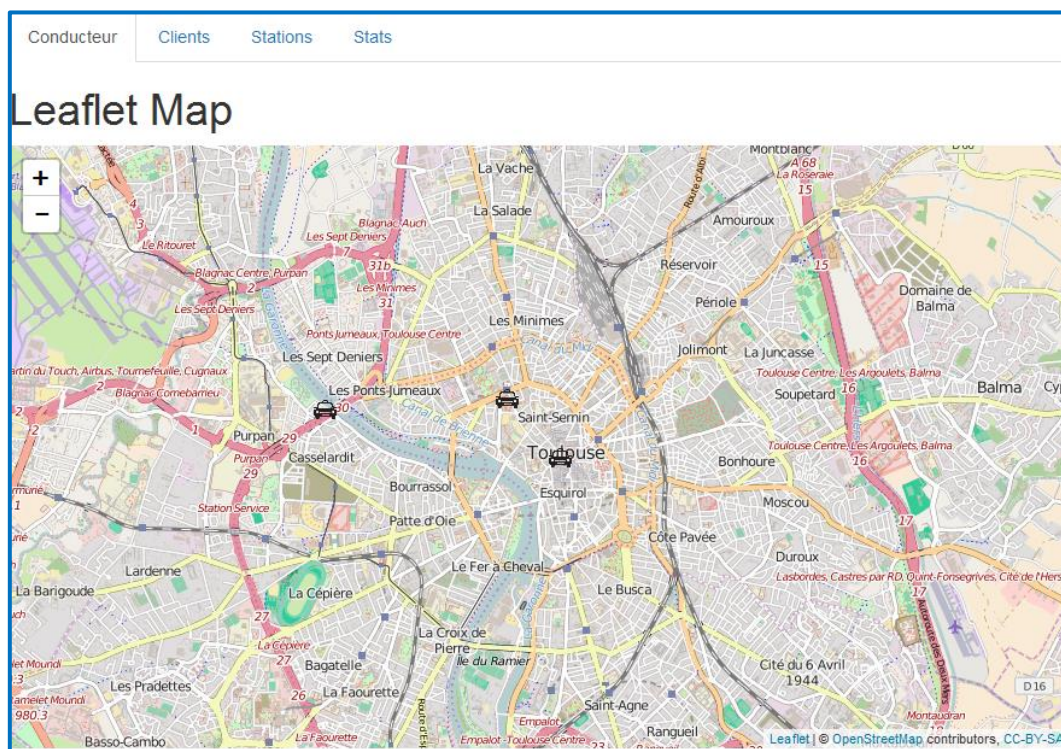


Figure 8 – Carte montrant la localisation d'un nombre limité de taxis contenus dans la base.

- ✓ Les taxis sont bien localisés sur la carte de la Figure 7.
- ✓ Les longitudes et latitudes ont été cherchées sur le Web, et sont cohérentes avec le plan.
- ✓ La forme des points correspond bien à celle souhaitée (forme de taxi)

- ✓ Le test d'adaptation est valide : sur la carte de la Figure 8, il n'y a bien que les 3 points pris au hasard qui sont affichés. Donc en ajoutant ou supprimant des tuples, le programme fonctionne bien.

8.1.2. Cartes

La Figure 9 est une carte qui géolocalise les clients contenus dans la base de données (1 client), tandis que la Figure 10 est une carte qui géolocalise ce même client ajouté d'un nouveau tuple simulé, afin de vérifier si le programme s'adapte à la taille des données.

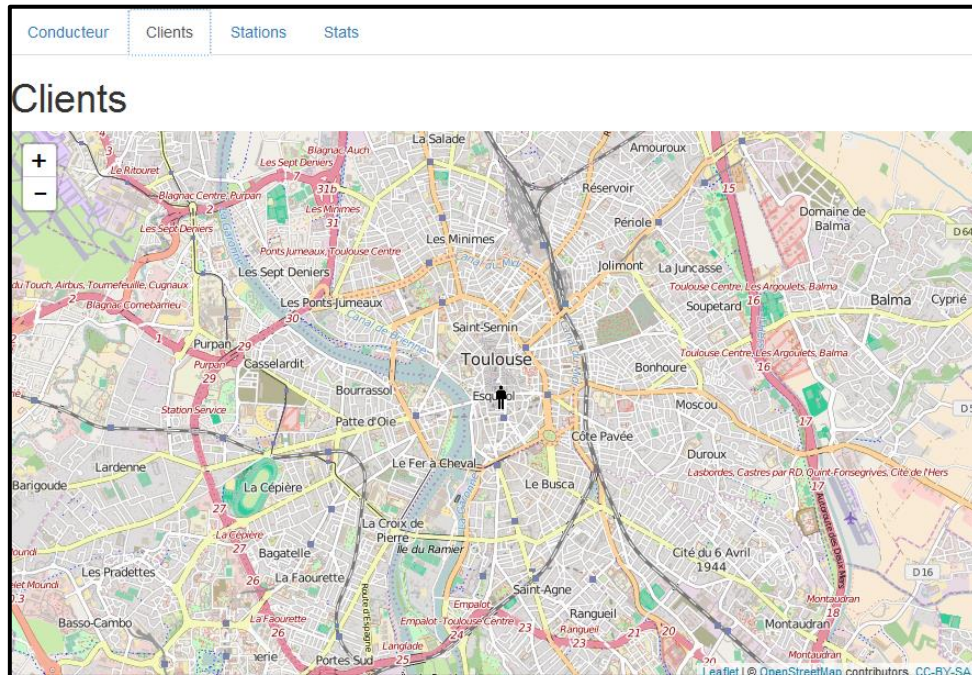


Figure 9 – Carte montrant la position des clients présents dans la base de données

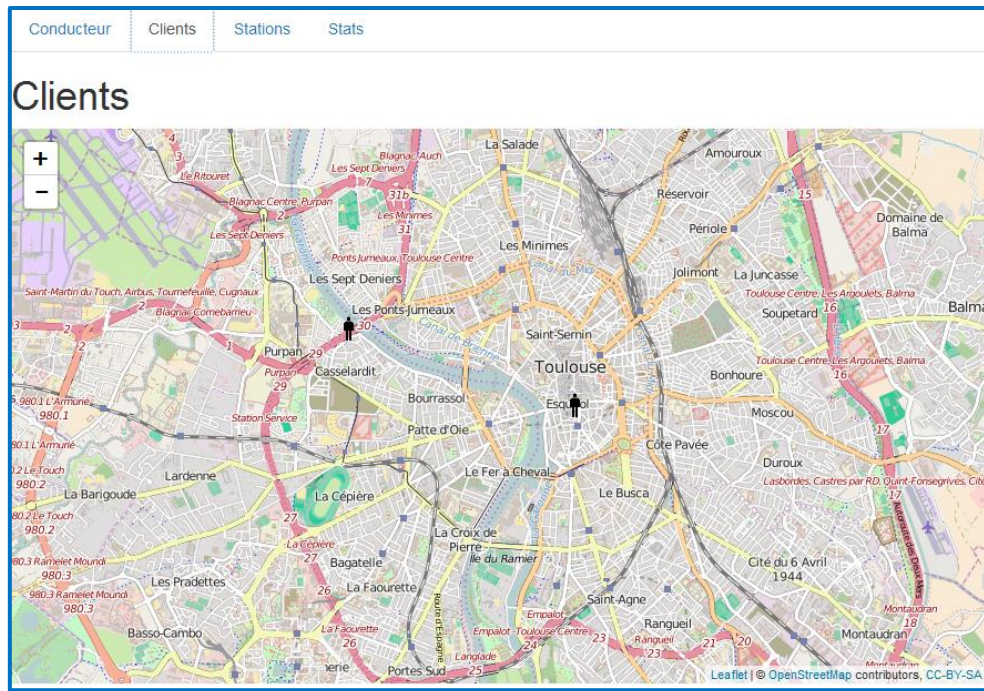


Figure 10 – Carte

- ✓ Le client contenu dans la base est correctement géolocalisé sur la carte de la Figure 9.
- ✓ La forme des points correspond bien à celle souhaitée (forme de piéton).
- ✓ Le test d'adaptation est valide : l'individu rajouté est bien représenté. Donc en ajoutant ou supprimant des tuples, le programme fonctionne bien.

8.1.3. Stations

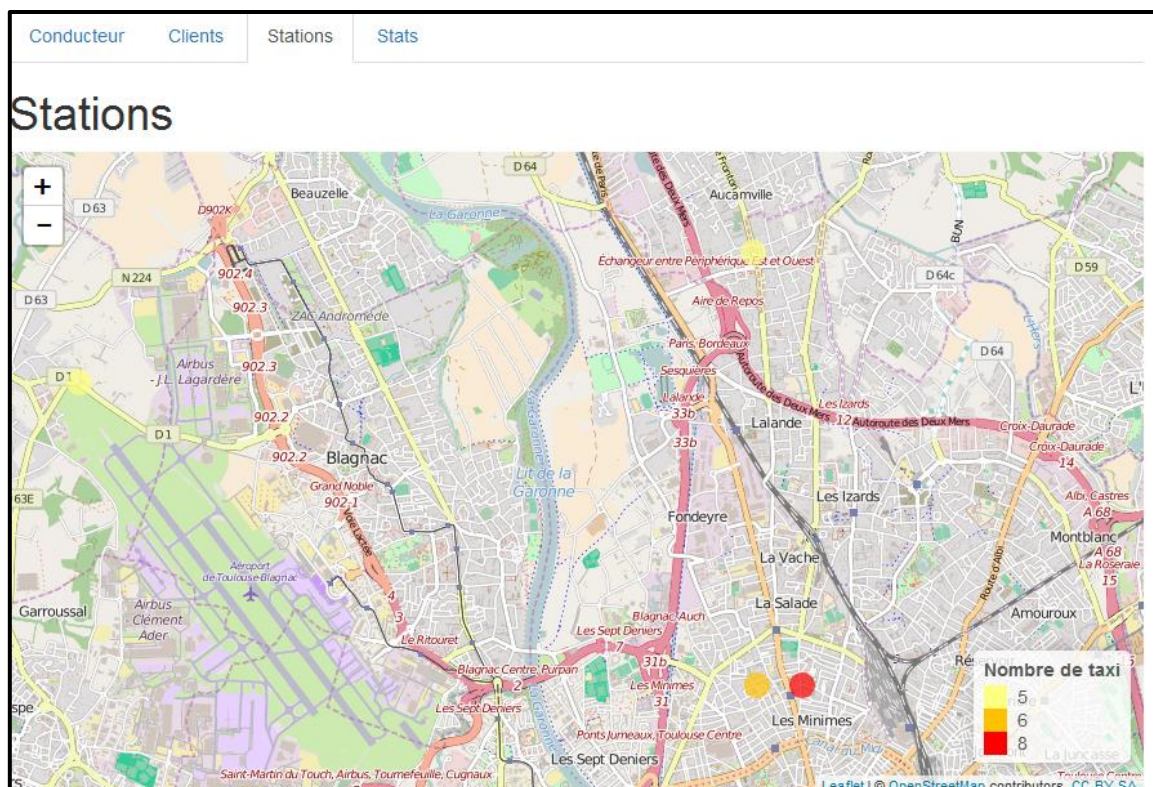


Figure 11 – Carte montrant la localisation des stations

9. Groupe 9 : Android