Rapport Projet S7 - SGBD

Romain PIERRE, Mathys MONELLO, Yannis YOASSI
PATIPE, Abdurahman EL CALIFA KAN IT BENSAIDI

Novembre 2023

"Covoiturage du Campus"

EI7IT204 Projet de SGBD
Département Informatique
S7 - Année 2023/2024

Table des matières

1 Introduction

Le covoiturage sur un campus est un moyen pratique, efficace, convivial et pas cher pour se déplacer d'un campus à une autre ville ou dans le sens contraire. Ce projet porte sur la création d'une base de données et de sa gestion pour permettre aux étudiants de s'inscrire, de proposer des trajets ainsi que de laisser des notes et des commentaires aux autres usagers. Pour cela, nous devons dans un premier temps modéliser nos données, puis créer un schéma relationnel, implémenter la base et permettre son utilisation au travers d'une application.

2 Modélisation des données

Le problème que nous devons modéliser est celui du covoiturage entre étudiants. Pour modéliser ce problème de covoiturage, nous sommes donc partis de l'énoncé. Nous l'avons analysé et nous en avons déduit un modèle entité-association qui modélise la base de données et un ensemble d'opérations qui vont nous permettre d'agir sur cette base.

2.1 Description du contexte de l'application

Avant d'arriver au schéma du modèle entité-association, il est d'abord important de déterminer les entités (ainsi que leurs attributs) et les associations qui vont répondre aux règles de gestion :

2.1.1 Les Entités :

- Étudiants : Cette entité nous permet de représenter les utilisateurs du covoiturage. Elle possède plusieurs attributs comme le nom, le prénom de l'étudiant et un identifiant ID_etudiant.
- Voitures : Cette entité nous permet de représenter la voiture du conducteur. Cette entité contient plusieurs attributs. les attributs type, couleur et état permettent au conducteur de rentrer la description de son véhicule. L'identifiant de cette entité est l'immatriculation du véhicule (ici nous avons supposé que l'immatriculation d'un véhicule ne peut pas changer au cours du temps).
- Trajets : L'entité Trajets est constituée de 4 attributs. L'identifiant de cette entité est ID_Trajet, les attributs ville de départ, date de départ et heure de départ permettent de rentrer les informations du trajet dans la base de données.
- Points d'arrêts : Cette entité nous permet de représenter les points d'arrêt. Un point d'arrêt possède six attributs qui permettent de le décrire précisément dont le prix par passager, la durée du trajet du départ jusqu'à l'arrêt. Un point d'arrêt est identifié par son attribut ID_arret.
- Avis : Cette entité est identifiée par son attribut ID_avis. Les deux autres attributs (note et commentaire) permettent aux passagers d'exprimer leurs avis.

2.2 Les associations et cardinalités :

Pour une modélisation complète du sujet, il faut aussi réussir à associer correctement les entités entre elles et avec les bonnes cardinalités.

1. Posséder : Décrit la relation entre un étudiant et un véhicule. Un étudiant peut posséder 0 ou n voitures, et une voiture appartient à 1 seul étudiant.

2. Utiliser : Modélise la façon dont les véhicules sont liés aux trajets. Une voiture peut être utilisée dans 0 à n trajets, et un trajet ne peut être utilisé que par une seule voiture.
3. S'inscrire : Permet de relier les étudiants aux points d'arrêts et aux trajets. De 0 à n étudiants peuvent s'inscrire à 0 ou à n points d'arrêts et trajets, et un trajet ou un point d'arrêt peuvent avoir de 0 à n passagers.
4. S'arrêter : Relie les trajets aux points d'arrêt, indiquant où le véhicule peut potentiellement s'arrêter. Lors d'un trajet, un véhicule peut s'arrêter de 1 à n fois, mais un point d'arrêt correspond à un seul trajet.
5. Correspondre à : Permet de relier les entités Avis et Trajets. Un trajet peut correspondre à 0 ou n avis.
6. Émettre et Recevoir : Ce sont des associations liées aux avis, indiquant que les étudiants peuvent émettre et recevoir des avis sur les trajets. Un étudiant peut émettre ou recevoir de 0 à n avis. En revanche, un avis est émis par un étudiant et peut concerner de 1 à n étudiants.

2.2.1 Règles de gestion :

En analysant l'énoncé, nous avons déterminé plusieurs règles de gestion que notre modélisation et notre application devront respecter :

- Règles liées aux utilisateurs :
  - Seuls les étudiants inscrits dans le supérieur peuvent s'inscrire au service de covoiturage.
  - Un étudiant peut s'inscrire en tant que conducteur, passager ou les deux.
  - Si un étudiant possède un véhicule , alors il est conducteur.
- Règles liées aux véhicules :
  - Le nombre d'inscrits à un trajet ne peut pas dépasser la capacité du véhicule.
- Règles liées aux trajets :
  - Un conducteur, et uniquement un conducteur, peut proposer un trajet vers ou à partir d'une ville.
- Règles liées aux points d'arrêt :
  - Un étudiant peut proposer ou s'inscrire à un point d'arrêt.
  - C'est le conducteur qui doit valider la proposition ou l'inscription au point d'arrêt.
  - Le conducteur fixe un prix par point d'arrêt.
  - Pour les trajets qui partent du campus, aucun étudiant ne peut monter en cours de trajet (choix du groupe).
- Règles liées aux avis :
  - Seuls les participants au trajet peuvent émettre un avis sur celui-ci.
  - Un avis et une note (optionnelle) peuvent être émis sur une ou plusieurs personnes.

2.2.2 Contraintes

Dans un modèle entité-association, une Contrainte d'Intégrité Fonctionnelle (CIF) est un mécanisme qui assure que les relations entre les entités respectent des règles précises. Dans notre modèle, la CIF entre les entités Étudiants, Trajets et Points d'arrêt permet de modéliser le fait qu'un étudiant inscrit dans un trajet ne peu s'inscrire que dans un unique point d'arrêt.

2.3 Modèle entité-association

Voici le modèle entité-association qui est la combinaison des entités, des associations des cardinalités et des règles de gestion.
Ici, nous pouvons voir deux schémas. La figure 1 représente le MCD de notre premier rendu. La figure 2 représente la version actuelle et finale. La seule modification est que nous avons enlevé l'attribut *ville_ arrivee* dans "Trajets" car il est en fait calculable à partir des points d'arrêt, de la ville de départ et de la distance du trajet. Dans le cas d'un trajet vers campus, la ville d'arrivée est "Bordeaux", sinon c'est le point d'arrêt à la distance la plus grande.

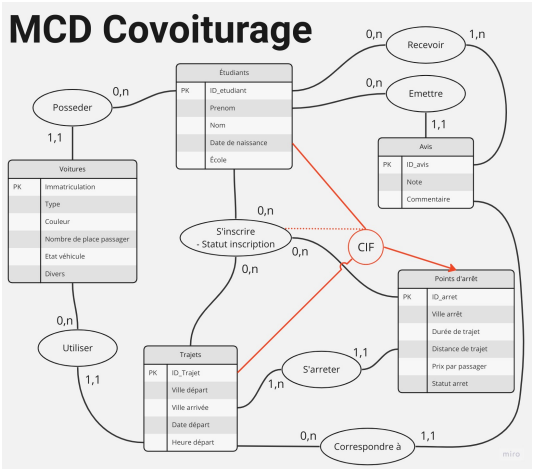

FIGURE 1 – Modèle conceptuelle de données original

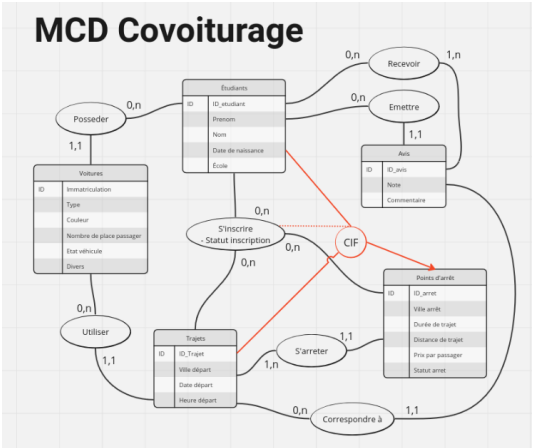

FIGURE 2 – Modèle conceptuelle de données final

3 Schéma relationnel

Après le modèle conceptuel, pour pouvoir réaliser la base de données en SQL, il nous fallait d'abord passer par le modèle relationnel.

3.1 Passage au relationnel

3.1.1 Les relations 1-n

Ces relations disparaissent pour laisser place, à un clé étrangère dans la table possédant un élément au maximum. Dans notre cas cela concernait, la table **Voitures** possédant un unique propriétaire, la table **Trajets** n'ayant qu'une voiture, celle sur les **Avis**, avis étant donné sur un unique trajet.

3.1.2 Le cas des avis

Le sujet nous proposait que les avis concernent une seule personne ou un trajet (toutes les personnes du trajet). Dans ce cas il aurait suffit d'avoir une variable "etudiant_noté" qui serait mise à **NULL** dans le cas ou ce serait une note concernant le trajet.
Cependant nous avons décidé d'avoir une base offrant plus de flexibilité, ainsi nous avons décidé que chaque étudiant pourrait avoir un avis sur un, certains ou tous les étudiants.
Nous avons donc eu à créer en plus de la table **Avis**, une table Réception_avis, composée de 2 clés étrangères ; L' ID de l'étudiant concerné par l'avis et l'ID de l'avis qui le concerne (clés primaires).

3.1.3 L'inscription

Ayant une association non binaire dans notre modèle nous avons du créer une nouvelle table, la table Inscriptions. Une inscription est caractérisée par un ID d'étudiant et un ID de trajet (clés primaires), et possède une clé étrangère vers un point d'arrêt.
Dans notre application les clients peuvent, soit s'inscrire à un point d'arrêt existant, soit proposer de nouveaux **Points d'arrêt**. Un point d'arrêt tout comme une inscription possèdent un état de validation. Par défaut à la création il est mis à **NULL**. Ensuite en fonction de l'action du conducteur du trajet (propriétaire de la voiture associée au trajet), il sera mis à **FALSE** ou **TRUE**.

3.2 Contraintes d'intégrité, dépendances fonctionnelles

La base de données, en pratique est modifiée pa r des clients. Afin de conserver une certaines cohérence des données il est nécessaire de mettre en place des contraintes. Étant donné la complexité du projet elles ne se veulent pas exhaustives.
Elles concernent donc aussi uniquement les données modifiables par les clients.

3.2.1 Unicité

Le fait d'avoir des clés primaires composées de deux éléments, permet d'assurer l'unicité de ces couples. Ainsi, un étudiant ne peut avoir qu'une inscription par trajet, et il ne peut être concerné qu'une fois par la rédaction d'un avis.

3.2.2 Contraintes sur les variables

Chaque variable devra posséder un type et peut être, soit obligatoirement non nulle, soit autorisée à avoir une valeur nulle. Cette dernière caractéristique, nous permet à la fois d'assurer l'existence de valeurs nous paraissant essentielles ou d'ajouter une valeur possible à nos variables. Qu'ici nous exploitons avec l'état des **Inscriptions** (états : non traité (**NULL**), accepté(**TRUE**), refusé(**FALSE**)).

3.2.3 Contraintes sur les requêtes

Après avoir utilisé l'application nous avons remarqué que la base pouvait se retrouver dans des états incohérents suite à des manipulations des utilisateurs.
Ainsi en plus d'avoir pris en compte la contrainte qui évite que les utilisateurs se notent eux mêmes. Nous en avons ajouté pour éviter les inscriptions sur les **Trajets** complets, les inscriptions de la part des conducteurs et les inscriptions sur des **Trajets** passés.
Comme indiqué précédemment nous sommes conscients que cette liste est non exhaustive, et que cette exhaustivité est proportionelle à la connaissance spécifique du sujet (ici un système de co-voiturage).

3.2.4 Points d'arrêt

A première vue un point d'arrêt est caractérisé par la ville et le trajet qui le concerne, ce sont de bons candidats pour devenir des clés primaires. Ce qui fait cette table n'est pas de 3e forme normale, en effet certains éléments (le prix par exemple) de la table dépendent de clés non primaires (la ville et trajet).
Au début du projet, on ne savait pas si un trajet allait avoir plusieurs arrêts dans une même ville, dans ce cas il aurait fallu rajouter des informations, et avoir une clé primaire encore plus longue (impact sur les performances), d'où l'ID pour les **Points d'arrêt**.
Même si cette table n'est pas de 3e forme normale, elle reste de forme normale Boyce-Codd. Etant donné qu'on à une clé primaire avec un seul élément on a un gain de performances.

3.3 Schéma relationnel en 3e forme normale

Etudiants (ID_etudiant, prenom ,nom ,date_de_naissance ,ecole) ;

Avis (ID_avis ,note ,commentaire, #etudiant_redacteur) ;

Reception_avis (#etudiant_note ,#ID_avis) ;

Voitures (immatriculation ,type_voiture ,couleur ,nombre_de_places ,etat, divers, #ID_etudiant) ;

Trajets (ID_trajet ,ville_depart, date_depart ,heure_depart ,#immatriculation) ;

Points_arret (ID_point_arret , ville_arret, duree_trajet ,distance_trajet ,prix_par_passager ,statut_arret ,#ID_trajet) ;

Inscriptions (#ID_etudiant ,#ID_trajet ,#ID_point_arret ,statut_inscription) ;

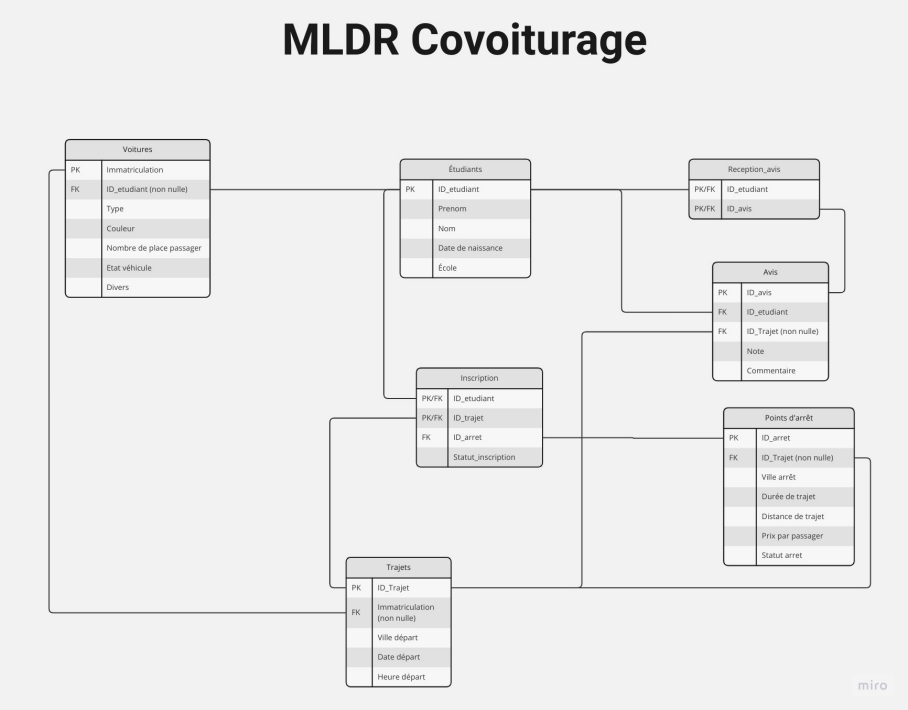

FIGURE 3 – Modèle logique de données relationnel

4 Implémentation

Dans cette section, nous parlerons de la création de la base de données, des contraintes d'intégrité de la base, ainsi que de l'implémentation des commandes SQL réalisant les opérations retenues.

4.1 Création de la base de données, en prenant en compte les contraintes d'intégrité

Pour créer notre base, nous avons utilisé le système de gestion de base de données **PostgreSQL**. Ensuite, nous avons rédigé les scripts de création, de suppression et d'insertion.

4.1.1 Create.sql

Ce script permet de créer les tables de la base de données, ainsi que de définir les attributs, leur type et les contraintes qui les concernent.
Par exemple, on peut voir que lors de la création de la table, on dit que la colonne **ID_etudiant** est de type *serial* et que cette donnée ne peut pas être vide. Les données de type *serial* sont des entiers, mais ce type possède une propriété qui est la suivante : lorsqu'on ajoute une nouvelle ligne, l'attribut *ID_etudiant* incrémente automatiquement.
Pour les données de type *char*, on peut aussi renseigner la taille maximale du champ. Et il y a aussi le type *date* qui nous permet de définir simplement le champ *date_de_naissance* plutôt que de devoir

passer par des conversions.
Nous définissons aussi les clés primaires et les clés étrangères lors de la création de la table. Dans l'exemple suivant, on le remarque grâce à la ligne *CONSTRAINT pk_etudiants PRIMARY KEY (ID_etudiant*, et pour préciser qu'une clé provient d'une autre table, on utilise l'instruction suivante *CONSTRAINT fk_etudiant_redacteur FOREIGN KEY (etudiant_redacteur) REFERENCES Etudiants(ID_etudiant)*.
Cette instruction provient de la création de la table **Avis** et permet de faire le lien entre l'attribut *etudiant_redacteur* de la table **Avis** et l'attribut *ID_etudiant* de la table **Etudiants** puisque l'étudiant rédacteur doit correspondre à un étudiant présent dans la table **Etudiants**.

```
            -- Table Etudiants
CREATE TABLE Etudiants
(
    ID_etudiant SERIAL NOT NULL UNIQUE,
    prenom VARCHAR(20) NOT NULL,
    nom VARCHAR(20) NOT NULL,
    date_de_naissance DATE NOT NULL,
    ecole VARCHAR(20) NOT NULL,
    CONSTRAINT pk_etudiants PRIMARY KEY (ID_etudiant)
);
```

Le script **create.sql** nous permet aussi de définir les contraintes d'intégrité de différentes manières. La première est l'utilisation des mots clés *ALTER TABLE* et *ADD CONSTRAINT* comme le montre l'exemple de code suivant :

```
            ALTER TABLE Avis
    ADD CONSTRAINT note_positive CHECK (note BETWEEN 1 AND 5);
```

Sur cet exemple, on vérifie que l'utilisateur rentre une note comprise entre 1 et 5 inclus.
Nous avons choisi d'ajouter les contraintes en utilisant l'instruction *ALTER TABLE* afin de bien séparer la partie du code où l'on définit les tables et la partie où l'on définit les contraintes d'intégrité, car on peut également les spécifier lors de la création de l'attribut dans la table comme sur cet exemple :

```
    note INT NOT NULL CHECK (note BETWEEN 1 AND 5)
```

Car ces deux manières présentées dans les deux derniers exemples de code sont équivalentes.
Le second moyen c'est d'utiliser des *trigger* qui, lorsqu'on insère de nouvelles valeurs dans la base de données ou que l'on modifie certaines valeurs, les *triggers* vont vérifier que la base reste dans un état cohérent et que certaines contraintes sont aussi respectées. L'exemple de code suivant :

```
-- Eviter que les gens se donnent des avis a eux memes :
CREATE OR REPLACE FUNCTION verif_auto_avis_reception()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.etudiant_note IN (SELECT a.etudiant_redacteur FROM Avis a
    WHERE a.ID_avis = NEW.ID_avis)THEN
        RAISE EXCEPTION 'Un␣r dacteur␣d''avis␣ne␣peut␣pas␣donner
␣␣␣␣␣␣␣␣un␣avis␣  ␣lui-m me.';
```

```
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

nous montre un trigger qui permet d'éviter que les utilisateurs ne se notent pas eux-mêmes.

4.1.2 Drop.sql

Le script de suppression ne contient que très peu de lignes et sert à supprimer toutes les données de la base de données. Le code :

```
DROP TABLE IF EXISTS Inscriptions CASCADE;
```

montre un exemple de suppression des données d'une table.
On utilise l'expression *IF EXISTS* pour ne pas obtenir de message d'erreur si la table n'existe pas lors de l'execution du script.
Ensuite, nous utilisons aussi le mot clé *CASCADE* pour supprimer la table ainsi que les objets qui dépendent également de la table à supprimer. La requête de suppression est la même pour toutes les tables.

4.1.3 Insert.sql

Le script pour insérer les valeurs nous permet d'insérer notre jeu de données que nous avons créé dans la base de données. Notre script commence par la ligne présente sur ce code :

```
        TRUNCATE Etudiants, Avis, Reception_avis, Voitures, Trajets,
        Points_arret, Inscriptions RESTART IDENTITY;
```

Cette ligne permet de supprimer toutes les lignes des tables sans les parcourir si on exécute le script insert.sql sans avoir utilisé drop.sql et create.sql avant.
Le défaut de cette instruction est qu'elle s'effectue avec un verrou exclusif, ce qui veut dire qu'il ne peut pas y avoir d'opérations en parallèle sur les tables visées pendant l'exécution de l'instruction.
Elle permet aussi de réinitialiser les différentes valeurs des tables avec une exécution séquentielle, ce qui permet de ne pas déclencher d'erreurs comme par exemple, une clé étrangère renseignée avant que l'attribut correspondant à la clé étrangère soit défini car on crée les tables dans un ordre défini.
Ensuite, nous avons ajouté les données grâce à des requêtes *INSERT INTO table(xx) VALUES (xx);* comme le montre le code pour la table **Etudiants**.

```
        INSERT INTO Etudiants(ID_etudiant, prenom, nom,
        date_de_naissance, ecole) VALUES
(1, 'Yannis', 'MOUSSO',
TO_DATE('2002-11-01','YYYY-MM-DD'), 'ENSEIRB-MATMECA'),
(2, 'Romain', 'CAILLOU',
TO_DATE('2002-01-08','YYYY-MM-DD') , 'ENSEIRB-MATMECA'),
...
(26, 'Samy', 'TORTUE',
TO_DATE('2003-12-11','YYYY-MM-DD'), 'INSEEC');
```

Nous avons décidé d'ajouter les valeurs de l'attribut *ID_ etudiant* à la main malgré l'utilisation du mot clé *serial*, car c'était plus facile pour nous de rentrer les données dans les autres tables où l'attribut *ID_ etudiant* est défini comme étant une clé étrangère, mais si des étudiants s'inscrivent, l'id de l'étudiant incrémentera automatiquement.
Nous avons aussi défini le format de date que nous avons adopteé dans toute la base de données, ce format est le suivant *YYYY-MM-DD*. Et, nous avons réalisé un affichage pour vérifier que nous obtenons bien le nombre de données attendu par table lors de l'insertion des données dans la base.

4.2 Implémentation des commandes SQL réalisant les opérations retenues

Dans cette partie, nous allons voir des exemples des requêtes que nous avons dû implémenter pour récupérer certaines informations demandées.

4.2.1 Les requêtes de consultation

Pour les requêtes de consultation, nous avons utilisé deux manières de récupérer les informations, la première est l'écriture d'une requête normale de SQL, la deuxième est l'utilisation d'une fonction, ce qui permet de chercher les informations que l'on veut facilement sans devoir réécrire une requête entière.
Les requêtes que nous avons écrit comme des requêtes SQL de base sont les requêtes qui consultent les informations de la table, de manière générale comme la requête suivante :

```
SELECT Etudiants.*
FROM Etudiants INNER JOIN Voitures
ON Etudiants.ID_etudiant = Voitures.ID_etudiant;
```

Cette requête permet d'afficher les informations des conducteurs, elle ne dépend pas de paramètres extérieurs d'où l'utilisation de ce genre de requête.
Pour les fonctions, nous les avons utilisé parce qu'elles permettent d'adapter la recherche à des paramètres et donc de ne pas avoir à réécrire une requête entière chaque fois qu'on veut consulter certaines informations dans la base de données. Par exemple, la fonction suivante :

```
CREATE OR REPLACE FUNCTION trajet_proposes(jour_debut DATE,
jour_fin DATE)
RETURNS SETOF RECORD
LANGUAGE plpgsql AS
$$
BEGIN
    RETURN QUERY SELECT * FROM Trajets WHERE date_depart BETWEEN
    jour_debut AND jour_fin;
END;
$$;
```

permet de consulter les trajets proposés dans un intervalle de jours donnée. On a donc créé une fonction qui prend en paramètre le jour de début et le jour de fin puis retourne le résultat de la requête qui permet d'obtenir les trajets entre les deux jours renseignés.

4.2.2 Les requêtes de statistique

Pour les requêtes de statistique, on a procédé de la même manière que pour les requêtes de consultation. On a des requêtes simples et des fonctions pour récupérer les statistiques spécifiques. Dans ces requêtes, nous utilisons des opérateurs d'agrégation pour calculer les informations désirées. Sur cet exemple de code :

```
        -- Premiere maniere
SELECT AVG(nombre_passager) as moyenne_des_passagers
FROM (
        SELECT COUNT(*) as nombre_passager
        FROM Inscriptions
        WHERE statut_inscription = 'TRUE'
        GROUP BY Id_trajet
) as nombre_passager_par_trajet;

--Deuxieme maniere
CREATE OR REPLACE FUNCTION moy_avis(etudiant_arg INTEGER)
RETURNS NUMERIC
LANGUAGE plpgsql AS $$
DECLARE
    moyenne_note NUMERIC;

BEGIN
    SELECT AVG(a.note) INTO moyenne_note
    FROM Etudiants e
    INNER JOIN Reception_avis r ON e.ID_etudiant = r.etudiant_note
    INNER JOIN Avis a ON r.ID_avis = a.ID_avis
    WHERE r.etudiant_note = etudiant_arg;

    IF moyenne_note IS NULL THEN
        RETURN 0;
    ELSE
        RETURN ROUND(moyenne_note,2);
    END IF;
END;
$$;


SELECT e.nom, e.prenom, v.immatriculation, v.couleur,
moy_avis(e.ID_etudiant) as note_moyenne
FROM Voitures v INNER JOIN Etudiants e
ON v.ID_etudiant = e.ID_etudiant
    INNER JOIN Trajets t
    ON t.immatriculation = v.immatriculation
WHERE t.Id_trajet = 1
GROUP BY v.immatriculation, e.ID_etudiant
```

```
ORDER BY note_moyenne DESC;
```

On voit les deux manières que nous avons utilisé pour calculer les statistiques. Pour la première, on consulte la moyenne des passagers avec une requête SQL basique. Quant à la seconde, on cherche à classer les étudiants conducteurs en fonction de leur note moyenne.
Pour ce faire, nous avons créé une fonction qui calcule la note moyenne d'un étudiant puis nous l'appelons dans notre requête réalisant le classement pour faciliter l'obtention des informations.

4.3 update.sql

Dans ce fichier, nous avons créé des requêtes de mise à jour de la base de données, pour certaines de ces requêtes nous utilisons encore des fonctions pour faciliter les changements et pour d'autres non. Sur les exemples suivants :

```
-- L'ENSEIRB fait faillite, reorientation pour tout le monde...
UPDATE Etudiants
    SET ecole = 'FAC␣DE␣LETTRES'
    WHERE ecole = 'ENSEIRB-MATMECA' ;

-- Inflation, on double les prix...
UPDATE Points_arret
    SET prix_par_passager = prix_par_passager *2;

--Suppression des voitures SUV et de toutes les entrees qui en dependent.
DELETE FROM Inscriptions
WHERE Inscriptions.ID_point_arret
IN (SELECT Points_arret.ID_point_arret
    FROM Points_arret
    WHERE Points_arret.ID_trajet
    IN (SELECT Trajets.ID_trajet
        FROM Trajets WHERE Trajets.immatriculation
        IN (SELECT Voitures.immatriculation  FROM Voitures
            WHERE Voitures.type_voiture = 'SUV')));
DELETE FROM Points_arret
WHERE Points_arret.ID_trajet
IN (SELECT Trajets.ID_trajet
    FROM Trajets
        WHERE Trajets.immatriculation
        IN (SELECT Voitures.immatriculation  FROM Voitures
            WHERE Voitures.type_voiture = 'SUV'));
DELETE FROM Trajets
WHERE Trajets.immatriculation
IN (SELECT Voitures.immatriculation
    FROM Voitures WHERE Voitures.type_voiture = 'SUV');
DELETE FROM Voitures WHERE type_voiture = 'SUV';
```

. Nous pouvons voir les différents mots clés utilisés *UPDATE ... SET, DELETE* qui permettent de modifier la valeur d'un attribut d'une table pour une entité donnée ou de supprimer ses informations.

4.3.1 Des dépendances fonctionnelles qui vont loin

Dans la requête ci dessus on observe, que d'une seule entrée(ici une voiture) peuvent dépendre énormément d'autres entrées.
Une solution pourrait être d'utiliser l'option ON ACTION CASCADE elle permet de spécifier le comportement à adopter pour la ligne "fille" (la ligne faisant référence à la clé étrangère) lorsque la ligne "parente" (la ligne référencée) est supprimée. Dans ce cas l'action serait DELETE.
L'utilisation de cette option dans une contrainte de clé étrangère nous aurait simplifié le code en automatisant la suppression des données dépendantes lorsqu'une ligne parente est supprimée, ainsi maintenir la cohérence des données aurait été bien plus simple. Cependant, cette simplicité aurait nettement augmenté le risque de suppression involontaire de données.

C'est pour cela que nous avons préféré ne pas la rajouter.

5 Utilisation

Cette section s'intéresse à la partie application du projet. L'intérêt est de fournir une interface pour les utilisateurs afin qu'ils puissent facilement manipuler la base de données et utiliser notre service.

5.1 Environnement d'exécution

Pour notre application web, nous avons décidé d'utiliser un serveur http **Apache** et un serveur de S.G.B.D. **PostgreSQL**. La liaison et l'implémentation étant faite en *PHP/html* et en *CSS* pour le visuel. Nous avons fait ce choix par souci de portabilité sur les serveurs de l'école. Ainsi il a été facile de déployer notre application sur l'espace pédagogique de Bordeaux-INP.

5.2 Notice d'utilisation

Le fichier **README.md** contient les instructions pour mettre en place le serveur **PostgreSQL** et le serveur **Apache** de manière locale sur sa machine. Un script **push_server.sh** permet de rapidement envoyer les fichiers sources du dépôt sur le serveur web et donc de fluidifier le développement. L'adresse du site en "production" est renseignée au début du **README.md**.

En bas de la page d'accueil (*index.php*) se trouve un menu (voir Figure 4) qui permet de facilement réinitialiser la base de données ou au contraire de la peupler avec les données de tests. Ce menu de développement est laissé en production pour permettre de facilement charger les scripts afin de tester l'application. Les détails de ces actions sont :

Initialiser la base à zéro :
`drop.sql` ⇒ `create.sql`

Initialiser la base aux valeurs de test :
`drop.sql` ⇒ `create.sql` ⇒ `insert.sql`

Mettre à jour la base de test :
`drop.sql` ⇒ `create.sql` ⇒ `insert.sql` ⇒ `update.sql`

FIGURE 4 – Menu d'administration de la base de donnée

5.3 Descriptions des interfaces

Cette sous-partie détaille les interfaces/fonctionnalités que nous avons implémentées sur notre application.

5.3.1 Page d'accueil

La page d'accueil (voir Figure 5) comporte un menu permettant à l'utilisateur de se connecter ou de s'inscrire, de visualiser ses demandes en tant que conducteur ou passager et de proposer ou rechercher un trajet.
Ensuite une partie statistique est présente, affichant le nombre d'étudiants inscrits, le nombre de trajets proposés et le nombre total de kilomètres déjà parcourus (*i.e.* uniquement les trajets déjà effectués. L'encart du nombre de trajets proposés est cliquable et permet de voir l'intégralité des trajets de la base de données.
Le reste de la page présente les trajets futurs et quelques détails répartis en deux catégories. La première étant les trajets ayant comme ville de départ "Bordeaux" (c'est-à-dire les trajets depuis le campus). La deuxième catégorie comporte elle le reste des trajets (donc ceux qui ont le campus en destination). Les trajets sont cliquables et mènent vers une page qui détaille le trajet et permet à un étudiant de s'y inscrire.

5.3.2 Détails d'un trajet

La page de détails d'un trajet est donc accessible au clic sur un trajet. La page montre le nombre de places restantes sur le trajet, des informations sur le conducteur, la liste des passagers déjà inscrits ou en attente de validation et la liste des points d'arrêt.
Chaque point d'arrêt affiche le nom de la ville d'arrêt, l'heure de départ, la distance du trajet et le prix à payer par passager. Enfin, si le trajet possède encore des places libres, un bouton s'inscrire permet de faire une demande au conducteur pour rejoindre le trajet.
Un exemple de cette page est disponible en Figure 6.

5.3.3 Recherche et proposition

Pour ce qui est de la manipulation des trajets, l'utilisateur peut soit proposer un trajet (si il s'est inscrit en indiquant qu'il a une voiture), soit en rechercher.
Pour la recherche (Figure 7), l'utilisateur doit indiquer s'il recherche un trajet *vers* ou *depuis* le campus, ainsi que la ville d'arrêt et la date. Si la date n'est pas indiquée, cela permet une recherche de tous les trajets disponibles dans le futur pour la ville et la direction indiquées.
Pour la proposition de trajet, c'est le même système sauf que le conducteur doit renseigner plus de détails. La date et l'heure de départ sont obligatoires par exemple. Ensuite, le conducteur doit renseigner la durée du trajet en minutes, la distance du trajet en kilomètres et le prix du trajet. À noter que le trajet pour l'instant ne comprend que

$$Bordeaux \longleftrightarrow Ville\_renseignee$$

ou

$$Ville\_renseignee \longleftrightarrow Bordeaux$$

selon le sens choisi. Le conducteur a dans un deuxième temps la possibilité de renseigner des points d'arrêt intermédiaires autant qu'il le souhaite ou de s'arrêter là. Le premier écran de ce parcours est montré en Figure 8.

5.3.4 Connexion et utilisateurs

Pour la gestion des utilisateurs nous avons fait simple puisque aucune sécurité n'est mise en place, l'intérêt de l'application résidant uniquement dans ses fonctionnalités liées à la base de données. Pour se connecter il suffit donc de choisir un utilisateur de la liste déroulante du menu connexion (Figure 10) pour y être authentifié. Ainsi, on peut s'inscrire ou proposer des trajets, mais aussi regarder ou répondre à ses demandes en cours (voir sous partie suivante).
On peut également rajouter un utilisateur (Figure 9) en y renseignant ses informations à l'aide du bouton "s'inscrire" du menu de l'accueil. À noter que si on renseigne que l'on ne possède pas de voiture, on ne sera pas en mesure de proposer un trajet.

5.3.5 Gestion des demandes

Puisqu'un passager peut demander à s'inscrire à un point d'arrêt, il faut que le conducteur puisse valider ou non son inscription. Pour ce faire, une fois que l'on est connecté, on peut accéder à 2 types de demandes.
Le premier correspond aux demandes en tant que conducteur (Figure 12) et permet de justement accepter ou refuser la demande d'un passager.
Le deuxième correspond aux demandes en tant que passager (Figure 11) et permet de voir l'état de ses demandes en cours, réparties en 3 catégories (en attente, acceptée et refusée).

5.3.6 Fonctionnalités manquantes

Côté application, les fonctionnalités manquantes sont donc l'édition et la lecture des avis, ainsi que la capacité d'un passager à proposer un nouveau point d'arrêt au conducteur.
Ces fonctionnalités n'ont pas été implémentées dans un soucis de temps et car elles n'apporteraient rien de plus pédagogiquement puisque un assez grand nombre de requête est déjà mis en place au niveau du reste des fonctionnalités et que le système de validation est similaire à celui de l'inscription à un trajet/point d'arrêt de la sous partie précédente.

5.3.7 Galeries

Cette sous-partie montre des extraits de notre application web.

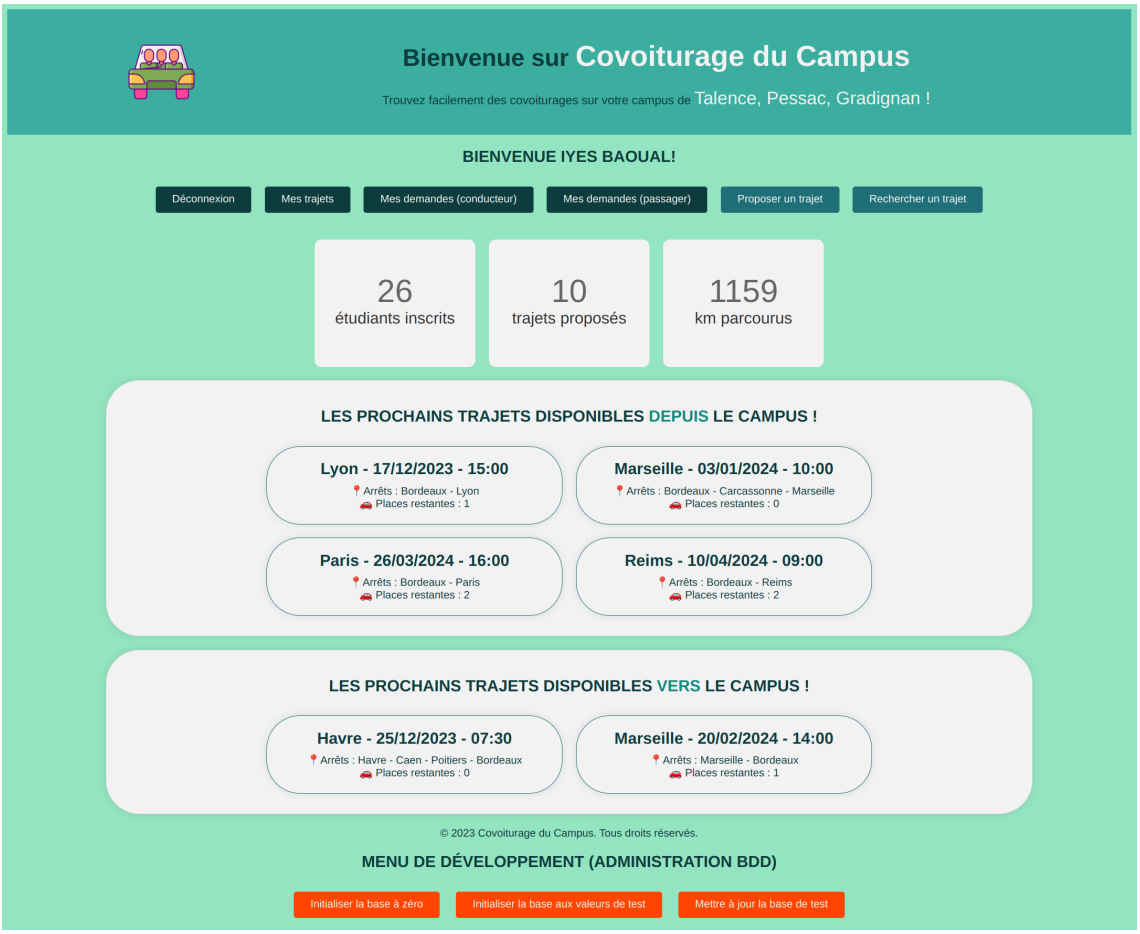

FIGURE 5 – Page d'accueil

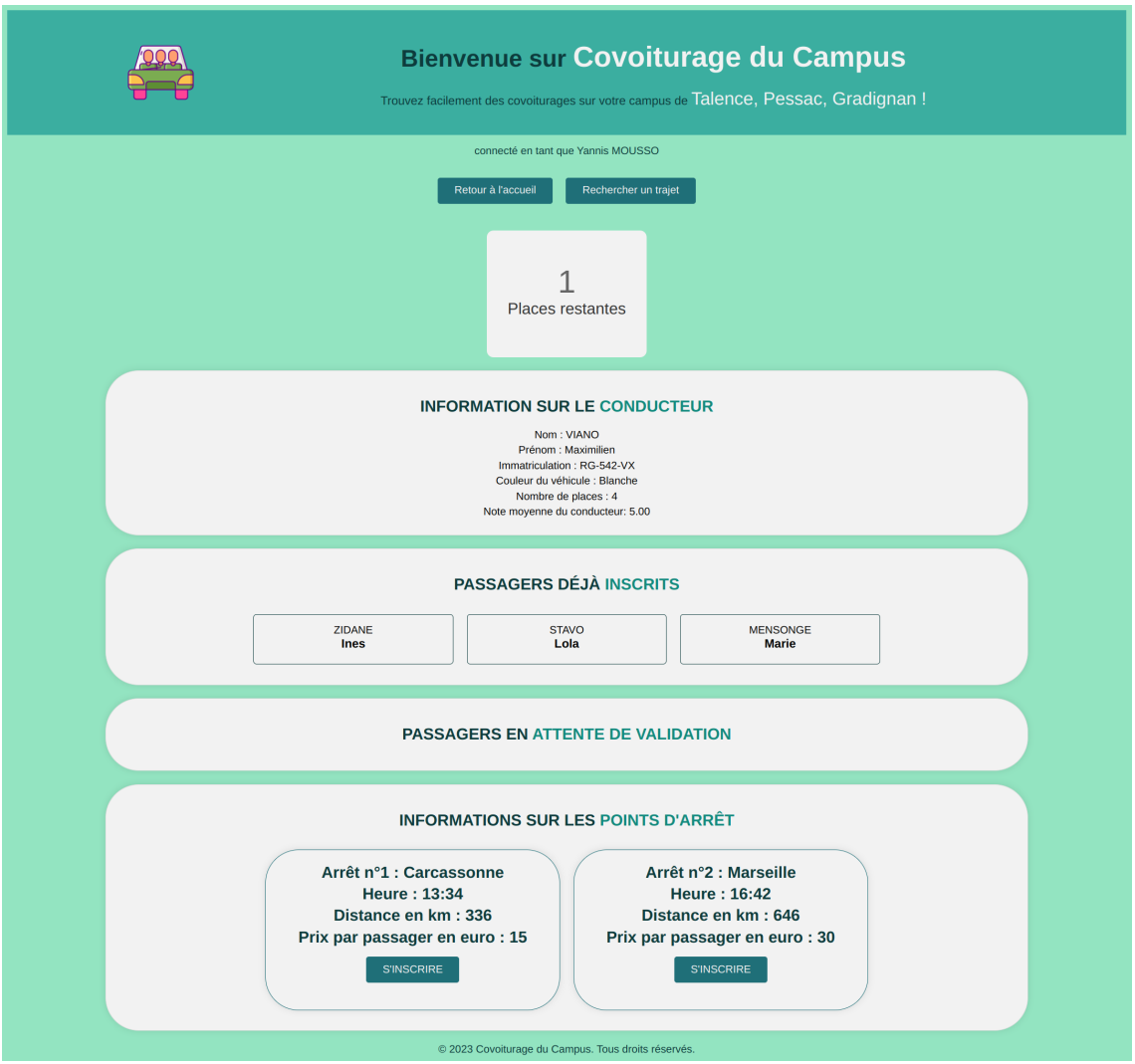

FIGURE 6 – Affichage d'un trajet

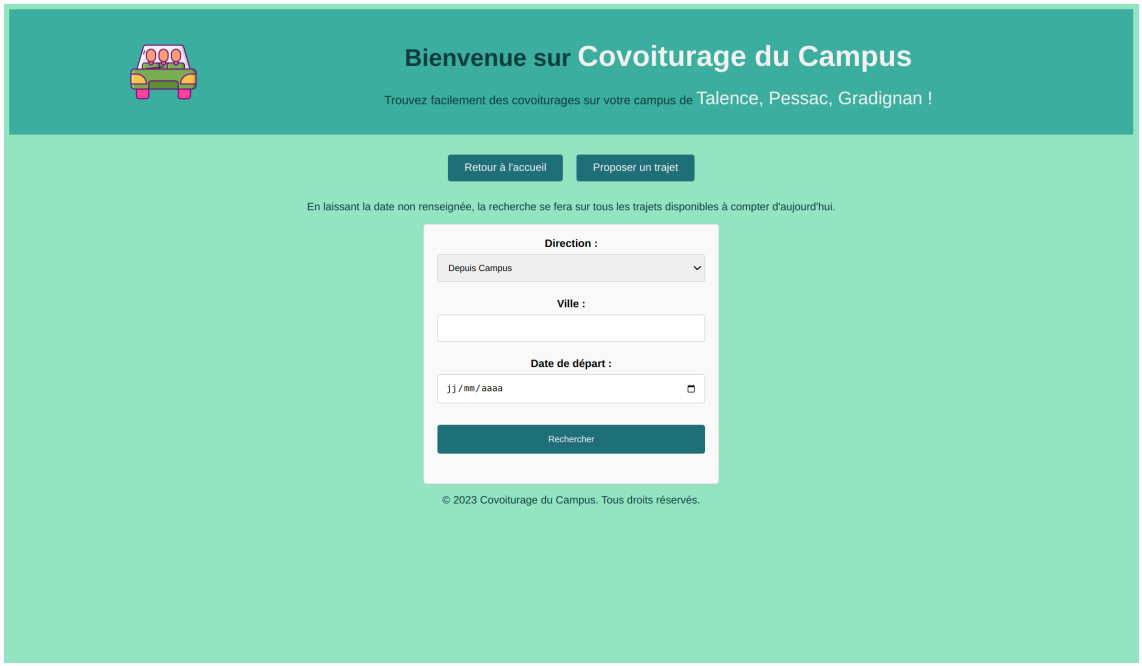

FIGURE 7 – Recherche de trajet

FIGURE 8 – Proposition de trajet

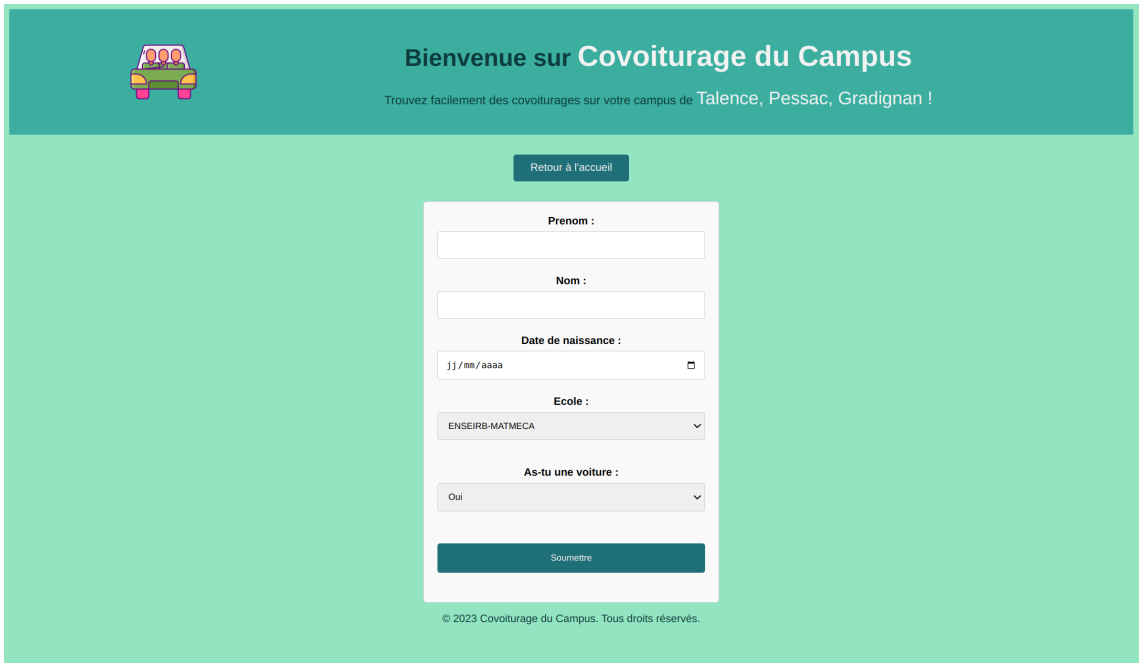

FIGURE 9 – Inscription d'étudiant

FIGURE 10 – Connexion

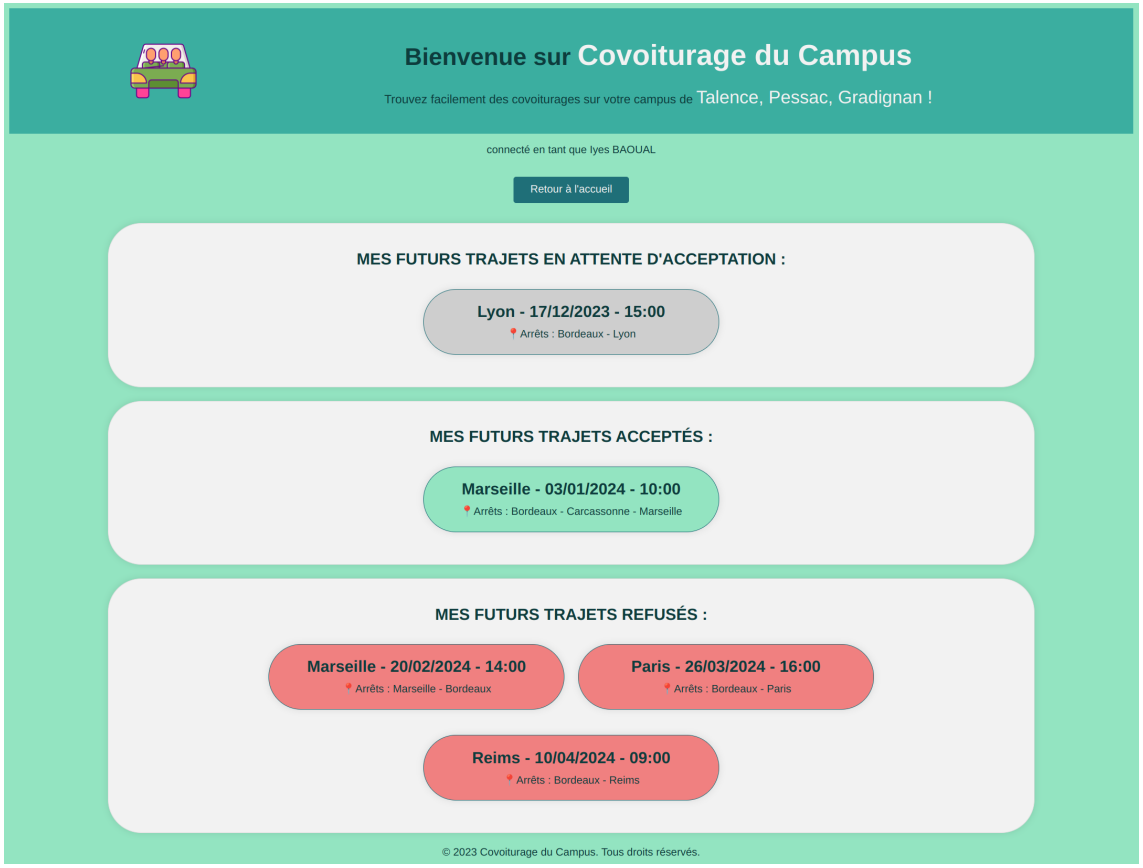

FIGURE 11 – Demandes en tant que passager

FIGURE 12 – Demandes en tant que conducteur