Tristan Leduc
Victor Perez
Hugo Robert
Roxanne Xu

Rapport de projet
BDCO

Table des matières

I. Introduction

Voici le rapport de notre projet de BDCO. Son but fut de mettre au point une application se basant sur le principe similaire que l'application blablacar : des conducteurs proposent à des utilisateurs de les emmener à leur itinéraire à bord de leur véhicule, si ceux-ci doivent aller dans la même direction.

II. Mode d'emploi

a) Comment lancer l'application

Pour compiler le projet, il faudra lancer Maven avec les commandes suivantes :

- mvn install pour installer les packages additionnels renseignés dans maven
- mvn compile pour compiler les fichiers du projet
- mvn exec:exec pour exécuter le projet

On peut par la suite naviguer simplement dans l'application :
1ère étape : l'identification par connexion ou création de compte
2ème étape : accès à l'application (gestion de véhicule, trajets, tronçons, soldes…)
3ème étape : fermeture de l'application en cliquant sur le bouton rouge close d'une fenêtre.

b) Limites de l'application VerbiageVoiture

Bien que les bases fondamentales du projet VerbiageVoiture aient été posées au cours du développement de l'application, il subsiste quelques manques. Nous allons les énumérer :
Manques par rapport au cahier des charges :

- La recherche de tronçon n'est disponible que sans correspondance (la recherche avec correspondance n'est pas encore supporté dans cette version)
- Il n'y a pas de limite de réservation pour un trajet en fonction du nombre de place
- Les boutons Montée/Descente des tronçons ou encore indiquer le début et la fin d'un trajet depuis l'utilisateur qui a créé le trajet sont implémentés mais ne sont pas pris en compte pour le paiement du tronçon.

Bugs :

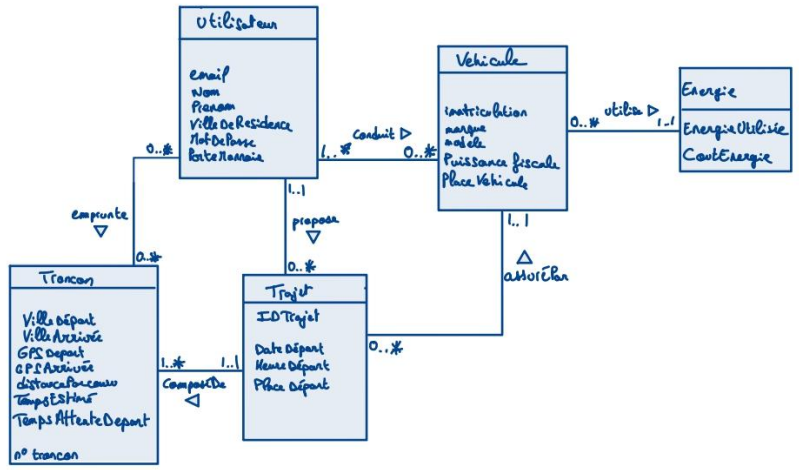
- Un utilisateur X peut réserver un trajet qu'il à lui même crée (cependant dans un cas réel, on imagine mal un utilisateur vouloir payer pour "rien")
- Un utilisateur peut réserver autant de fois qu'il veut un même trajet (en pratique cela peut être utile si il veut réserver pour plusieurs personnes depuis un même compte)
- certaines vérification pour les paramètres entrée par l'utilisateur ne sont pas effectués

III. Description du projet

III.1)Analyse de la base de donnée

a) Diagramme entité-association

Diagramme entité association :



Contraintes de valeur

energie utilisée = {essence, diesel, electrique, hybride}

placeDépart > 0

placeDépart < PlacesVehicule

PlaceVéhicule > 0

Puissance fiscale > 0

distanceTroncon > 0

dureeTroncon > 0

TempsAttenteDépart ≥ 0

Temps Estimé > 0

Distance parcourue > 0

PorteMonnaie ≥ 0

n° tronçon ∈ N*

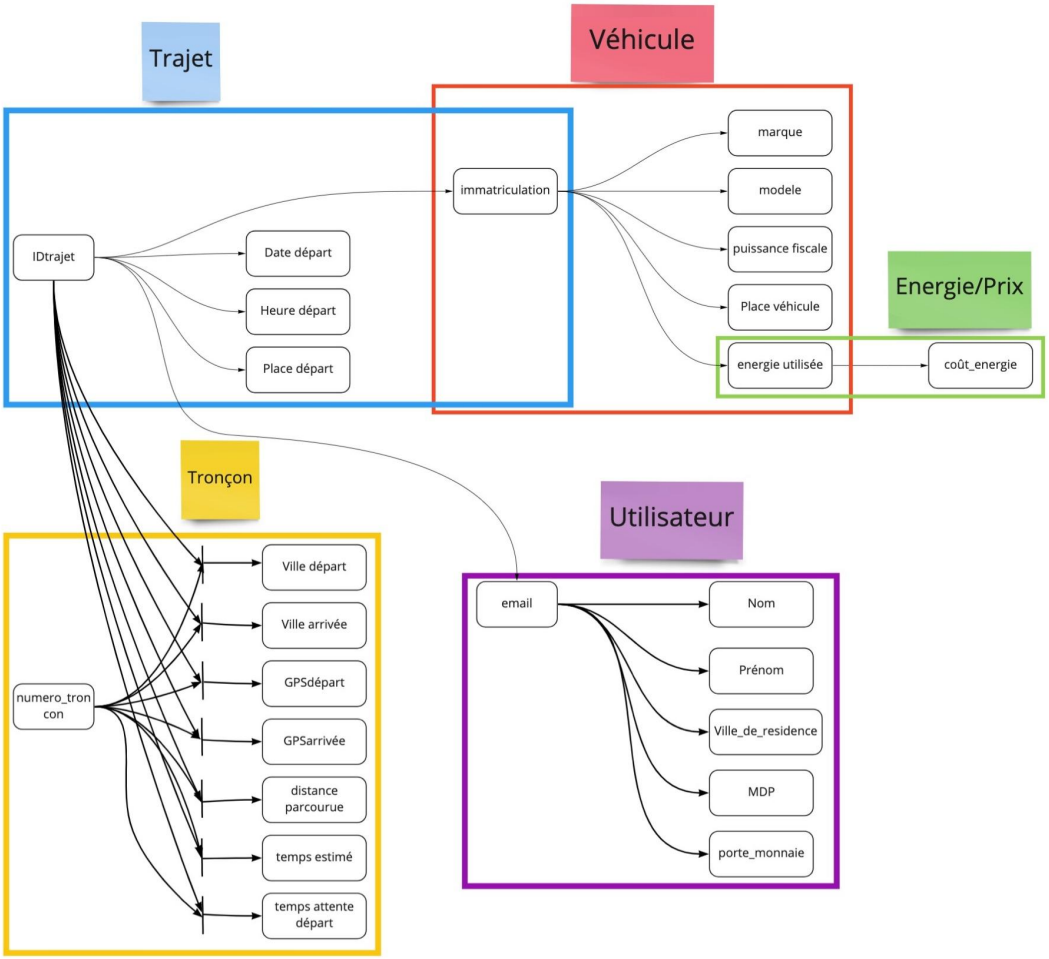
CoutEnergie > 0

Contraintes textuelles :

- Des passagers peuvent monter dans le véhicule à chaque étape
- 1 changement de véhicule maximum (si l'attente prévue à l'étape n'excède pas 1 heure,que la distance euclidienne entre les points GPS d'arrivée et de départ de l'étape ne dépasse pas 0,01)
- Pour proposer un trajet, un conducteur doit être associé à au moins une voiture

Le schéma entité association retranscrit les relations entre les entités du modèle. Il nous a fallu définir des contraintes de valeur pour certaines valeurs des tables. Nous avons aussi pu prendre conscience des attributs utiles. Ce schéma permettra, par la suite, de trouver un modèle relationnel en prenant en compte la redondance dans les tables d'un côté, et la facilité d'utilisation de celles-ci par des requêtes de l'autre.

b) Du schéma E/S vers un modèle relationnel



L'objectif de cette étape est de traduire le schéma entité-association en un modèle relationnel implémentable en SQL. Nous avons donc fait plusieurs choix techniques que nous allons commenter :

- Ce modèle permet d'avoir des tables sous la forme normale pour chacune d'elle
- Table Tronçon : son identification de fait par (IDTrajet, numero_troncon) qui permettent ainsi d'obtenir tous les paramètres d'un tronçon.
- Nous avons supprimé les associations "composé de"," propose", "assuré par", "utilise" caril il y avait une cardinalité 1..1 d'un côté de l'association (donc ces associations peuvent être supprimé car une jointure entre 2 clé primaire de 2 table permettra de faire le même rôle).
- Ce modèle relationnel permet, à notre sens, d'éviter au maximum la redondance (ce qui est pratique lorsqu'on veut ajouter/modifier des lignes à nos tables avec

des requêtes UPDATE/DELETE), toutes en interconnectant chaque table le plus possible pour faciliter les requête de type SELECT.

c) La formalisation des différentes tables

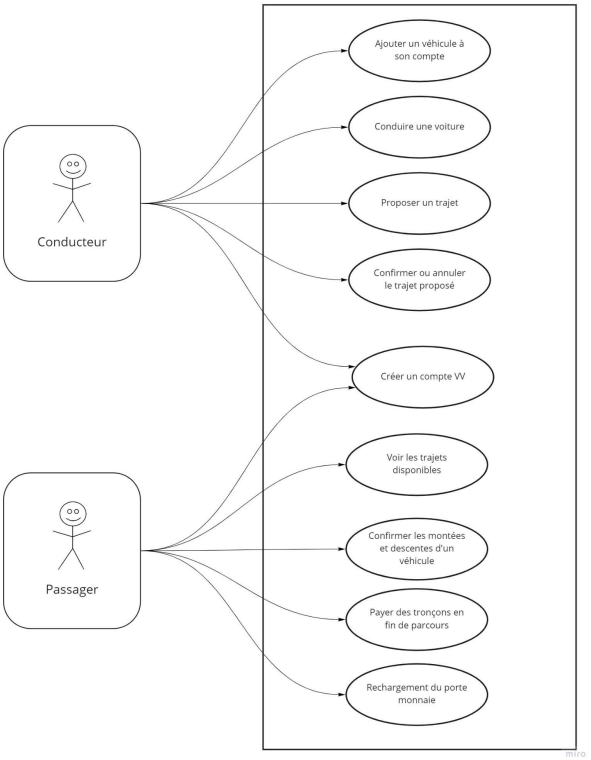
La base de données de notre projet se divise en sept tables :

- la table **UTILISATEUR** : recense les utilisateurs de l'application comme son nom l'indique. Un utilisateur est identifié de manière unique par son email. Il est également identifié par son nom, prénom, sa ville de résidence, son mot de passe, et son porte monnaie.
- la table **TRAJET** : représente les trajets planifiés par les conducteurs. Un trajet est identifié par
    - sa clé primaire IDTRAJET
    - son lieu de départ
    - l'immatriculation du véhicule assigné au parcours (la clé étrangère qui identifie un véhicule de manière unique)
    - la date d'arrivée et de départ
    - et les indices de validation effectives du début et de la fin du trajet par les utilisateurs
- La table **TRONCON** : étroitement liée à la table trajet, elle répertorie les différentes étapes d'un trajet. Chaque tronçon est identifié par son numéro de séquence dans un trajet, et le couple (numéro de troncon, idTrajet) forme la clé primaire de cette table. Les coordonnées GPS de départ et d'arrivée sont également dans cette table, ainsi que la distance parcourue, le temps estimé, le temps d'attente au départ du tronçon. Pour compléter le tout, des indices sont présents afin que l'utilisateur puisse confirmer sa montée dans le véhicule au début du tronçon (MONTEE_VALIDEE) et sa descente une fois à destination (DESCENTE_VALIDEE).
- La table **VEHICULE** : répertorie le parc de voitures enregistrées sur l'application. Les véhicules sont identifiés par leur immatriculation, leur modèle, leur puissance fiscale, la place dans le véhicule ainsi que l'énergie utilisée.
- La table **ENERGIE_PRIX** : associe le coût des énergies à leur type.

- La table **EMPRUNTE** : comporte les utilisateurs prenant part à un trajet. Il est ainsi nécessaire d'associer un email à un idTrajet et à un tronçon particulier.

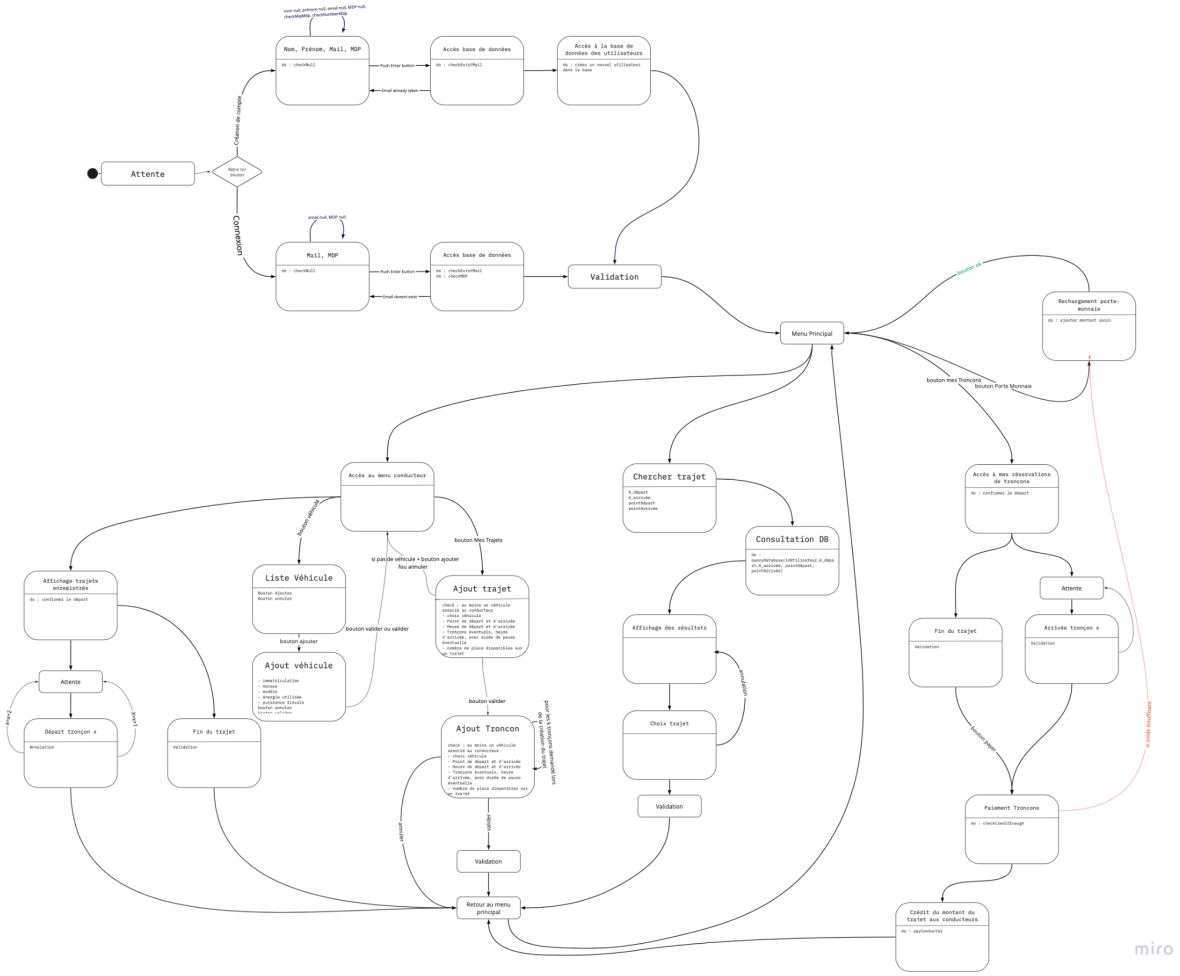
III.2) Analyse fonctionnelle et architecture de l'application

a) Les cas d'utilisation



Notre conception se base sur deux types d'utilisateurs, le Conducteur et le Passager. Ces utilisateurs auront des rôles différents. Cependant, un utilisateur peut être à la fois Conducteur et Passager. Sur l'application ses rôles seront donc transparent pour l'utilisateur, mais les actions d'un passagers (recherche trajet, ajout/suppression/paiement tronçons) ne sont pas soumis au même conditions que les actions en tant que conducteur (création/suppression trajet) puisqu'un conducteur, pour pouvoir proposer un trajet, doit avoir renseigné au moins une voiture.

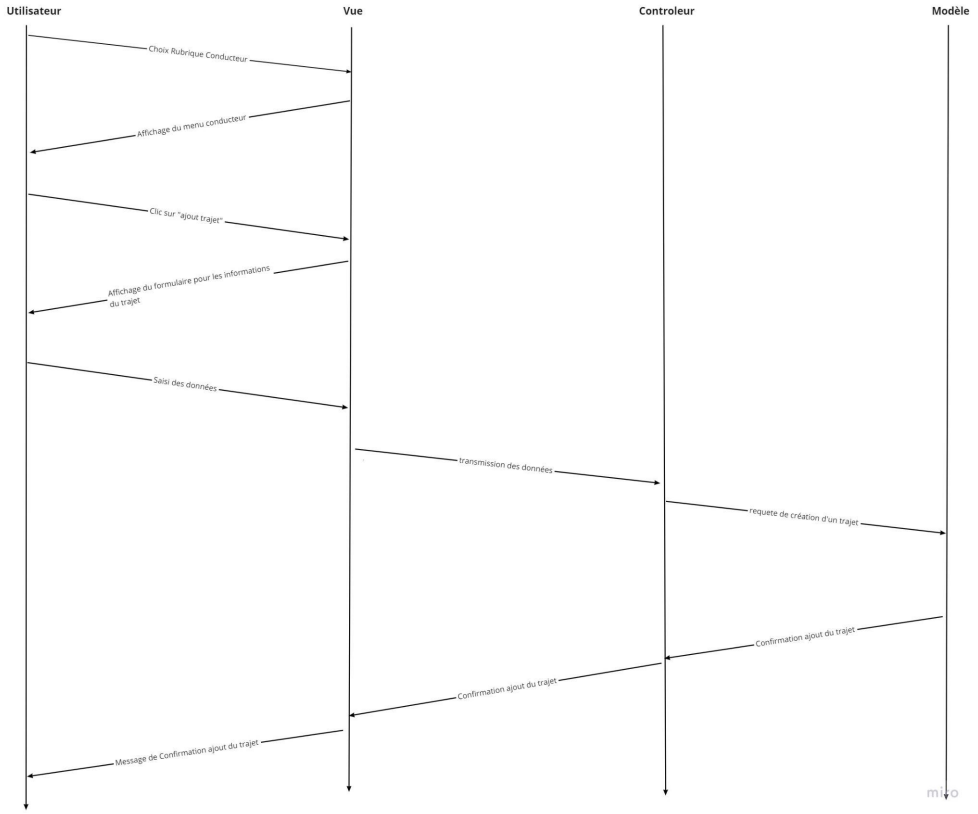
b) Diagramme états-transitions



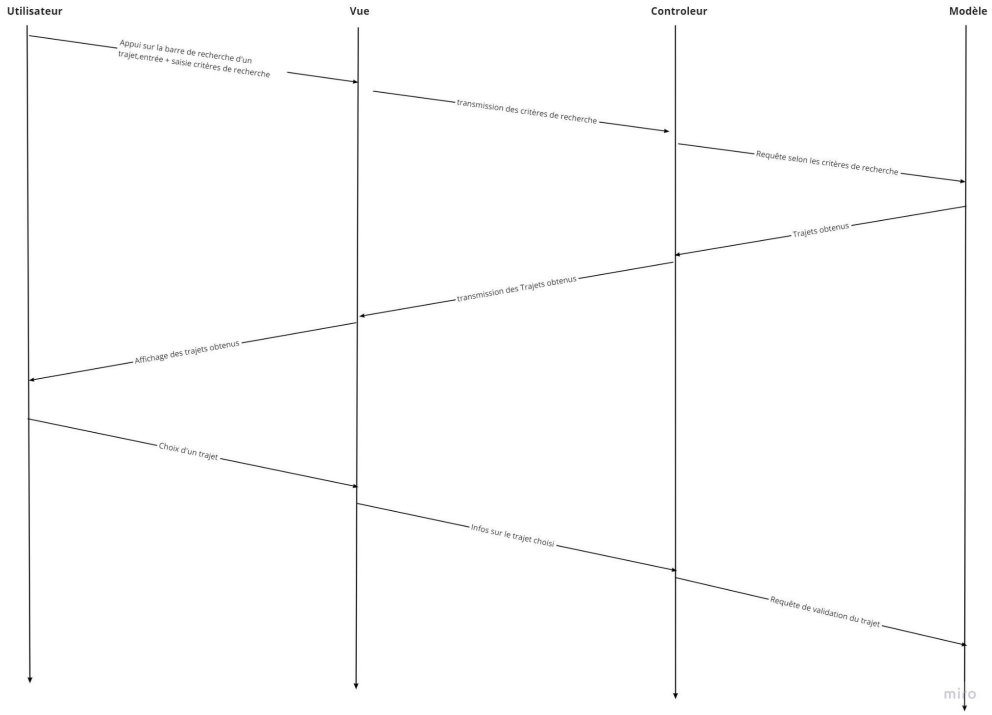
Ce diagramme présente les différentes étapes menant à la connexion d'un utilisateur, à l'ajout d'un véhicule et d'un trajet, à la recherche d'un trajet et aux tronçons déjà réservés par un passager. On peut observer la première phase d'identification (inscription ou connexion) qui partent toutes deux de la page initiale et qui vont vers le menu principal. Une fois au menu on a plusieurs choix : gérer ses véhicules, son solde, ses trajets créés, ses tronçons réservés et rechercher de nouveaux tronçons suivant votre lieu de départ et d'arrivée. Toutes ces branches reviendront finalement au Menu Principal. Cependant, si on souhaite payer des tronçons mais que l'utilisateur n'a pas les fonds, il va passer directement de l'étape "paiement tronçon" puis vers solde sans passer par le menu principal (flèche rouge). Certaines fonctionnalités, non demandées dans le cahier des charges, sont volontairement absentes comme la suppression de véhicule pour un utilisateur par exemple.

c) Quelques diagrammes de séquences

Ces diagrammes présentent le cheminement de l'ajout d'un trajet ainsi que celui de la recherche d'un trajet à travers l'architecture MVC.



Recherche et sélection d'un trajet par l'utilisateur



IV) Conception de Verbiage Voiture

a) Interface graphique

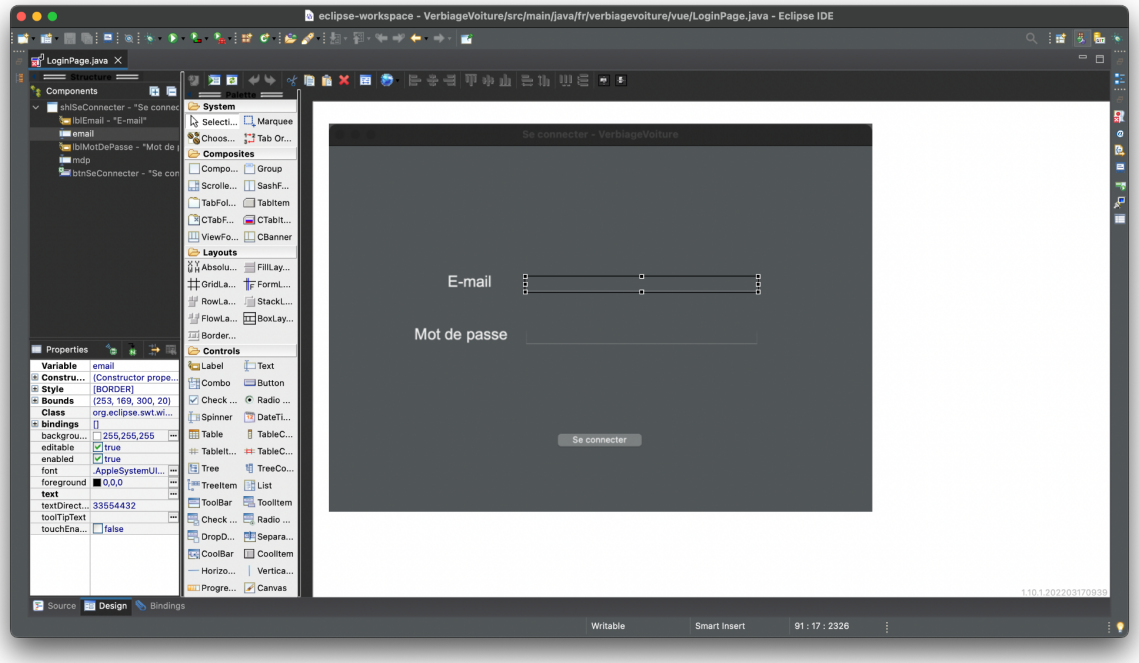
Afin de réaliser l'interface graphique de l'application, nous avons utilisé l'outil WindowBuilder intégré à Eclipse. L'intérêt d'un tel outil est qu'il offre une interface drag&drop permettant de très rapidement créer une telle interface sans connaissances préalables dans cette facette du développement Java.

Nous avons choisi cet outil car, outre sa bonne réputation, il a l'avantage d'être gratuit, open source, et multiplateforme, ce qui permettra à notre projet d'être facilement exécutable sur différentes configurations. De plus, son bon suivi assure que les technologies employées aujourd'hui ne seront pas obsolètes trop rapidement.

Dans le détail, WindowBuilder génère automatiquement le code Java (un objet provenant du package SWT par élément de l'interface) correspondant à l'interface graphique construite au sein de son interface. Les objets générés (champs de texte, etc.) possèdent des méthodes respectant la typologie employée en Java (Object.getText() par exemple), facilitant leur utilisation dans le programme. Enfin, il est possible de définir différents Listener permettant de réagir aux interactions de l'utilisateur avec l'interface graphique (clic sur un bouton, etc.).

Une des volontés de notre projet est de pouvoir s'exécuter facilement sur différentes configurations, et de ne pas dépendre d'un IDE en particulier. Ainsi, nous avons écrit un script Maven permettant de le compiler et de l'exécuter facilement à travers le terminal. Une de ses particularités est de récupérer sur Maven Central la version du package d'interface graphique SWT correspondant au système d'exploitation utilisé.

Nous pouvons voir à la suite une capture d'écran du WindowBuilder d'Eclipse, ainsi qu'un extrait du code correspondant généré :




```
    Label lblMotDePasse = new Label(shlSeConnecter, SWT.NONE);
    lblMotDePasse.setText("Mot de passe");
    lblMotDePasse.setFont(SWTResourceManager.getFont("Arial", 20, SWT.NORMAL));
    lblMotDePasse.setAlignment(SWT.CENTER);
    lblMotDePasse.setBounds(106, 232, 129, 29);

    mdp = new Text(shlSeConnecter, SWT.BORDER | SWT.PASSWORD);
    mdp.setBounds(253, 237, 300, 20);

    Button btnSeConnecter = new Button(shlSeConnecter, SWT.NONE);
    btnSeConnecter.addMouseListener(new MouseAdapter() {
        @Override
        public void mouseUp(MouseEvent e) {
            if (Login()) {//if connection success
                ChangeWindow();
                MenuPrincipal window = new MenuPrincipal(myco);
                window.open();
            }
            else {

                Message window = new Message("connexion impossible : adresse mail ou mot de passe incorrect");
                window.open();
            }
        }
    });
    btnSeConnecter.setBounds(289, 368, 120, 27);
    btnSeConnecter.setText("Se connecter");

}

protected boolean Login() {
    return myco.CheckEmailAndMDP(email.getText(), mdp.getText());
}
```

b) Implémentation des accès à la base de donnée

Une fois la partie Vue implémentée et l'architecture de l'application mise en place, l'enjeu est d'ajouter l'interaction entre l'application et la Base de Donnée.

Pour ce faire, nous avons implémenté toute une partie contrôleur : il existe un contrôleur pour chaque table de la base de donnée (utilisateur, voiture, trajet, tronçon) qui permet ainsi, à

travers de méthodes ayant en entrée les paramètres nécessaires aux requêtes SQL et en sortie le résultat souhaité. Ce type de méthode a donc pour but de générer le code SQL, d'envoyer la requête SQL vers la base de donnée, d'analyser la réponse et de renvoyer une interprétation de cette réponse.

Ainsi, toutes les requêtes identifiées dans "analyse des accès à la base de données" ont été implémentées dans cette partie de l'application.

Tous ces contrôleurs (dont leurs méthodes) ont alors été regroupés sous une classe unique "MyConnection" qui, au démarrage de l'application crée une connexion à la base de donnée, permet aussi d'accéder à toutes les méthodes des contrôleurs, garde en mémoire certains champs (notamment l'email qui identifie de manière unique un utilisateur) et à la fermeture de l'application ferme cette connexion.

Voici un extrait de la classe myConnection qui permet d'ouvrir/fermer une connexion avec la Base De Donnée :

```java
// Méthode qui crée une connexion à la BD
public MyConnection() {
    try{
        // Chargement du driver Oracle
        System.out.print("Loading Oracle driver... ");
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        System.out.println("loaded");

        // Connection à la BD
        System.out.print("Connecting to the database... ");
        this.conn = (Connection) DriverManager.getConnection(URL, USERNAME, PASSWD);
        System.out.println("connected");

        // Demarrage de la transaction (implicite)
        this.conn.setAutoCommit(false); // Pas de autocommit
        this.conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
    } catch (SQLException e) {
        System.err.println("failed");
        e.printStackTrace(System.err);
        this.conn = null;
    }
    user = new UtilisateurController(conn);
    vehicule = new VehiculeController(conn);
    troncon = new TronconController(conn);
    trajet = new TrajetController(conn);
    energie = new EnergieController(conn);
}

//fermeture de la connexion
public void closeConnection() throws SQLException {
    try {
        System.out.print("closing connection to the database... ");
        conn.close();
        System.out.println("closed ");
    } catch (SQLException e) {
        System.err.println("failed");
        e.printStackTrace(System.err);
        this.conn = null;
    }
}
```

Ainsi, du point de vue de la partie Vue de l'application, l'accès à la base de donnée est très simple : il suffit d'instancier une objet MyConnection à l'ouverture de l'application, utiliser ces méthodes relativement explicites en donnant en paramètres les données rentrés par l'utilisateur et en affichant les valeurs retournés.

Enfin, dans chaque page de l'application, un listener sur le bouton de fermeture de la fenêtre s'active lorsqu'on clique dessus et permet, avant d'arrêter le système, de fermer la connexion à la base de données.

Ce choix d'implémentation permet ainsi de séparer au mieux la partie graphique de l'application de la partie interaction avec la base de données. On augmente ainsi la lisibilité du code (donc plus facile et rapide à comprendre, corriger, améliorer) et permet, par une organisation plus simple, de rendre le code plus fiable.

IV. Gestion d'équipe

a) Séparation des tâches

La stratégie mis en place lors de ce projet de base de donnée a été de partager les tâches par groupe de 2 et d'adopter un travail de groupe au sein des binômes d'une part parce que ce choix permet d'être plus agile et de pouvoir travailler sur plusieurs choses en mêmes temps sans souffrir d'une perte de temps à cause de coordination lente entre des membres trop nombreux, et, d'autre part, d'être plusieurs sur une même tâche pour confronter nos idées et garder du recul sur la conception de l'application, surtout au début du projet où il est très important d'être attentif sur les besoins attendu par le cahier des charges et sur la nécessité de cloisonner chaque tâche pour se laisser une marge de liberté le plus large possible sans se fermer des portes tout seul.

L'objectif était également de tester notre adaptation aux difficultés, pour ce faire nous avons décidé de changer les groupes chaques semaines pour s'assurer de ne pas créer 2 binômes indépendant et de garder une synergie au sein de l'équipe avec chaques participant qui aurait un point de vue différent des autres et ainsi pourrait y apporter une forme "d'expertise" dans un certain domaine. Cette expertise permet ainsi de savoir à qui s'adresser suivant le problème rencontré et d'adapter les tâches qui ont été nombreuses lors de la dernière étape au savoir-faire de chaque membre. Durant cette dernière étape on a notamment pu constater une expertise pour victor autour de la partie graphique de l'application et des compatibilité du projet avec les différents système, ou encore hugo autour de la gestion de Base de données et de l'interaction application-Base de donnée.

Bien Sûr, une liberté été laissé à chaque participant pour partager son point de vue, proposer des systèmes différents de ceux qui étaient choisi au départ pour pouvoir rester agile face aux problèmes et y répondre de la manière la plus pertinente en confrontant les idée de chacun avoir de faire un choix de groupe et non un choix personnel.

b) Rétrospection

Le projet a pu être réalisé en grande partie, avec un respect presque total du cahier des charge du sujet. Cependant, nous n'avons pas eu beaucoup de temps pour se lancer dans la partie développement pure du code. En effet, nous avions commencé à l'implémenter qu'à partir de la 3ème semaine. Le manque de temps, surtout avec la période de partiel au même moment que la réalisation de différents projets font que nous ne sommes pas entièrement satisfait des fonctionnalités de l'application VerbiageVoiture.

V. Bilan

Grâce à sa grande pluralité, ce projet nous a permis de développer une application en full stack. Nous avons rencontré quelques difficultés pour trouver comment mettre en place efficacement l'interface graphique.