École Centrale Lyon

MOD 4.6
Rapport

Agence de voyage SNCF

Élèves :
Erwan Benkara-Mahammed

Enseignant :
Liming Chen
Thomas Duboudin

6 décembre 2021

Table des matières

Table des figures

Introduction

Ce rapport explique la conception et l'implémentation d'un système de gestion de base de données pour une agence de voyage SNCF. Dans un premier temps, on décrira les attentes autour du projet et son cahier des charges. Puis, on s'intéressera à une possibilité d'infrastructure pour supporter la solution ainsi que la charge qu'il implique. Ensuite, on détaillera l'implémentation en propre de la solution avec les choix de conception et de réalisation. Un guide d'installation est présenté dans une dernière partie.

1 Cahier des charges

L'agence SNCF a en charge la vente et la gestion des billets pour des trajets en TGV. Il s'agit donc de mettre en place une solution qui permette aux clients de réserver des billets et à l'agence de gérer les ressources, à savoir les trains, les trajets et les billets.

1.1 Fonctionnalités côté client

La liste des fonctionnalités principales quant à l'interface client est la suivante :

— Rechercher un billet de train pour un trajet donné
— Créer un compte client avec ses informations personnelles
— Réserver un billet de train
— Consulter ses réservations
— Souscrire à une carte de réduction

L'ensemble du parcours client doit être fluide et intuitif sans quoi un client risque d'abandonner sa recherche. Ainsi, la création de compte doit être rapide mais est cependant obligatoire pour réserver un billet afin de relier un client identifié à un billet. Cela permet aussi au client de pouvoir consulter ses trajets passés et à venir ainsi que de souscrire à une carte de réduction qui lui permet de tarifs préférentiels.

1.2 Fonctionnalités côté agence

Il faut garder en mémoire que la solution est développée pour une agence de voyage de la SNCF et non pas pour le service national de gestion du trafic ferroviaire français. Ainsi, il convient de limiter les possibilités de modification/suppression/ajout pour les administrateurs de l'agence. Ceux-ci ne pourront conséquemment pas ajouter une gare TGV.

La liste des fonctionnalités principales quant à l'interface agence (administration) est la suivante :

— Consulter tout ou partie des tables de la base de données
— Ajouter/modifier/supprimer des trajets
— Ajouter/modifier/supprimer des cartes de réduction

— Ajouter/supprimer des trains

Là encore, la fluidité et la simplicité doivent être de mise pour que l'administration de la base de données se fasse facilement. On attend également que l'interface renvoie des messages de confirmation ou d'erreur après des opérations effectuées sur la base de données.

2 Infrastructure

D'après les données récentes, les TGV accueillent plus d'une centaine de millions de passagers par an. La solution doit donc supporter cette charge sans interruption de service afin de garantir d'une part, la qualité de celui-ci et un maximum de réservation et d'autre part, protéger l'image de marque de la SNCF. L'ensemble de l'infrastructure matérielle doit être pensé pour supporter les milliers de connexions journalières alors même que la charge n'est pas constante mais variable en fonction des périodes de l'année (vacances scolaires, événements importants, etc.), des jours de la semaine (plus de temps le week-end pour organiser un voyage) mais aussi en fonction des heures de la journée puisque les réservations se font généralement aux heures creuses (entre 12h et 14h ou à partir de 18h/19h).

La suite de cette partie explique en partant de la couche applicative [1] jusqu'à la couche matérielle, une possibilité d'infrastructure en omettant cependant la couche de liaison et une partie de la couche réseau qui sont du ressort des fournisseurs d'accès à internet. La SNCF pourra cependant souscrire un contrat spécifique auprès d'un des FAI français.

NOTE : Il ne s'agit pas de la présentation de la solution réellement implémentée dans le cadre de ce projet mais d'une possibilité concrète et argumentée d'implémentation réelle.

2.1 Couche applicative et logicielle

La couche applicative est destinée à l'interaction utilisateur (administrateur ou client). Il s'agit ici de la couche la plus importante qui comporte un serveur web et un serveur de base de données. Le domaine du développement Web étant bien balisé, une pile LAMP (Linux Apache MySQL PHP) semble tout à fait envisageable.

Pour autant, cette pile logicielle est quelque peu vieillissante et d'autres solutions logicielles peuvent ici être envisagées.

2.1.1 Choix de l'OS

L'utilisation de Linux n'est pas impérative mais est souvent préférée des administrateurs systèmes et réseaux en raison de sa facilité d'utilisation et de sa très large utilisation. On ne proposera pas d'autres alternatives, celle offerte par Windows Server étant jugée inappropriée et coûteuse. Pour la distribution, on choisira CentOS ou RedHat qui sont mieux adaptées aux environnement de production, assurant une compatibilité entre les éléments logiciels.

1. On retient ici le modèle en couche TCP/IP et non le modèle OSI

2.1.2 Choix de la technologie serveur

La technologie serveur est ici cruciale car elle est responsable du traitement et de la distribution des requêtes. Dans la pile LAMP, c'est le logiciel HTTPD d'Apache qui est utilisée mais il existe aujourd'hui des alternatives bien plus performantes à commencer par la version 2 nommée HTTPD2 (couramment appelée Apache2) ou bien Nginx qui depuis plusieurs années occupe une place croissante. On retiendra donc cette dernière solution logicielle côté serveur en raison des fonctionnalités qu'elle propose : reverse proxy, gestion simplifiée du protocole HTTP sur *tuyau* SSL/TLS (HTTPS), architecture *event-based*.

2.1.3 Choix de la technologie de base de données

MySQL a longtemps été très populaire et reste une technologie de base de données très utilisée à travers le monde entier. Pour autant, le volume de donnée immense que doit traiter la SNCF rend ici son utilisation discutable en raison de ses performances vis-à-vis de la montée en charge. Dans la partie suivante, quelques alternatives sont présentées.

2.2 Choix du langage de programmation et du framework de développement

Enfin, il s'agit de choisir le langage de programmation et le framework Web associée. PHP a longtemps a occupé une position de domination dans le domaine du Web mais d'autres technologies ont émergé comme Python ou NodeJS. L'utilisation de Python est très fréquente lorsqu'il s'agit d'écrire des API (avec des frameworks comme Django ou Flask par exemple) car le langage jouit d'une grande popularité en raison de sa souplesse et de sa simplicité. Cependant, pour des raisons propres à Python, les performances constatées pour le rendu de contenu statiques (pages HTML, images ou autres documents) et la vitesse d'exécution ne sont pas satisfaisantes. On peut se référer aux liens suivants pour des explications plus détaillées : l'impossibilité de parallélisation en Python [GIL] et les recommandations officielles de Django relatives aux documents statiques [SF]. NodeJS s'appuie le moteur Javascript V8 Engine et jouit de beaucoup d'avantages notamment le fait que toutes les fonctions soient non-bloquantes grâce à un système de *callback*, sa flexibilité et sa *scalabilité* ou encore son système de cache. On retiendra donc cette solution avec le framewoork web Express et son écosystème.

En résume, les choix applicatifs et logiciels sont les suivants :
— OS : CentOS
— Serveur : Nginx
— SGBD : PostrgeSQL
— Langage de programmation/Framework web : NodeJS/Express

Dans le cadre du projet, la solution a été implémentée la couple NodeJS/Express et MySQL comme système de base de données. Les parties relatives au déploiement n'ayant que peu de pertinence dans le cadre du cours.

2.3 Couche réseau

Comme on l'a vu, le site Web de la SNCF reçoit des milliers de visiteurs par jours et les serveurs doivent donc traiter un nombre important de requête. Afin d'éviter une congestion du réseau, il faut assurer une duplication de l'ensemble de la pile applicative : serveur web et serveur de base de données.

Ainsi, selon la technologie de base de données choisies, il est possible d'utiliser des bases de données réparties. Dans le cas de MySQL, la solution équivalente distribuée s'appelle MySQL Cluster bien que dans les cas de gros volumes de données, cette technologie atteint vite ses limites. Pour remédier à cela, PostgreSQL (open-source), OracleDB (commercial) ou MSSQL (commercial) semblent plus adapter du point de vue performances.

La même logique s'applique au serveur Web. Il faut s'assurer que la charge est correctement répartie entre les différentes instances de l'application qui sont déployées sur les serveurs. Une stratégie de *load-balancing* (répartion de charge en français) est donc à prévoir. Le répartiteur de charge est un logiciel chargé d'assurer la bonne distribution des requêtes aux serveurs en fonction de leur charge de travail actuelle et passée. Avec la solution retenue, soit une instance Nginx peut être dédiée pour opérer la répartition de charge soit le module `pm2` peut être mis en place, celui-ci ayant l'avantage d'être spécialisé pour les applications NodeJS et offre une large palette de fonction.

2.4 Couche matérielle

Pour finir, regardons de plus près la couche matérielle qui est le support de tout le reste de l'application. Il est indispensable de prévoir des ressources suffisantes pour supporter la pile applicative que l'on vient de présenter. Il paraît donc impensable de ne pas tabler sur des serveurs dédiées, c'est-à-dire avec des ressources (RAM et CPU) exploitées par l'application de l'agence de voyage uniquement. L'hébergement qu'il soit sur site ou chez une entreprise tierce doit garantir trois aspects :

- la bande passante qui doit être suffisamment élevée
- la résilience des serveurs (tolérance aux panne, support rapide)
- la disponibilité des ressources

Si le choix est fait de faire appel à une entreprise tierce, en France, OVH a une offre complète de solution d'hébergement. Sinon les principaux acteurs du Cloud (Amazon AWS, Google GCP, Microsoft Azure) offrent des solutions qui peuvent largement convenir aux besoins.

Afin de clarifier et illustrer les explications précédents, on donne un diagramme de la solution complète qui permet de faire le lien entre les différents éléments évoqués.

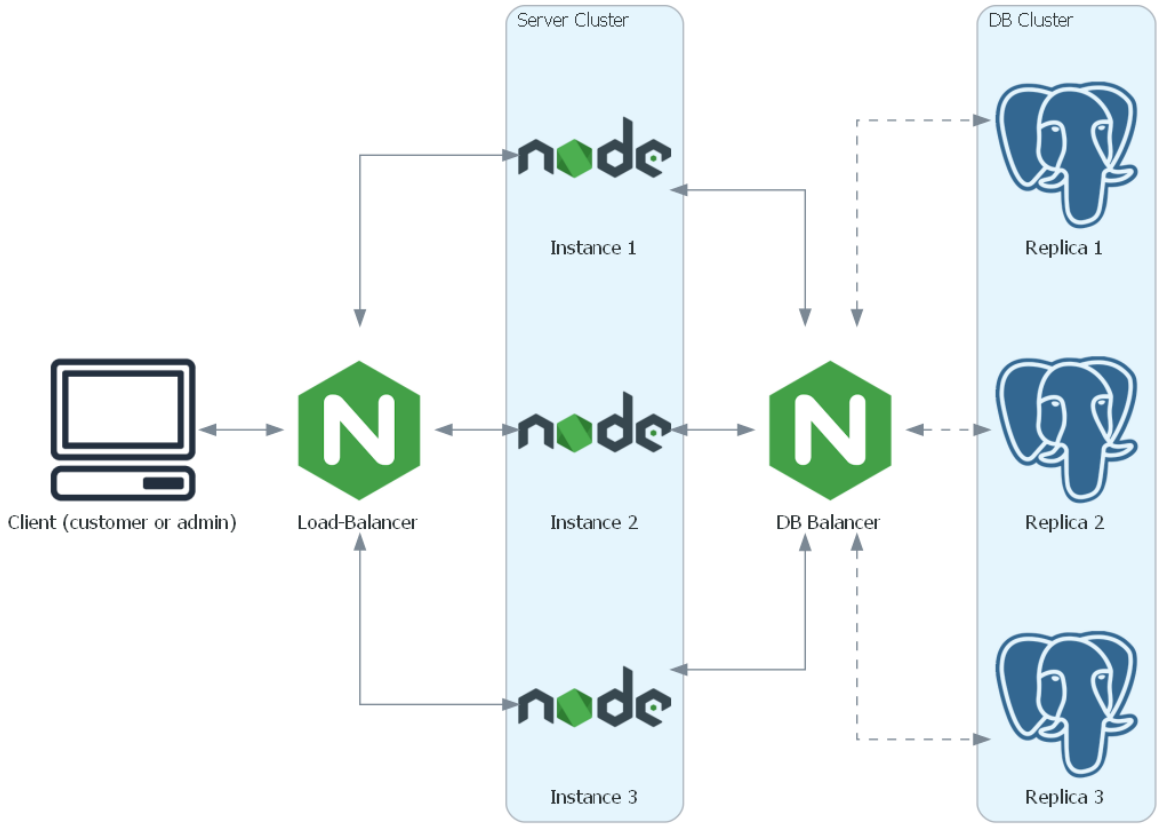

FIGURE 1 – Diagramme de la solution établie avec les différents composants. Les support physiques et les OS sont omis

3 Architecture de la base de données

3.1 Mise en place des modèles de données

Le sujet décrit les différents éléments attendus pour la gestion de l'agence de voyage. On retrouve par exemple les trajets, les clients ou encore les trains qui sont caractérisés par des attributs clairement identifiés. Le sujet disponible en Annexe A, semble distinguer les entités et attributs suivant :

- Billet(identifiant, ville de départ/arrivée, gare de départ/arrivée, date et heure de départ/arrivée, heure de départ/arrivée, prix)
- Train(identifiant, voitures)
- Voiture(identifiant, nombre de place)
- Place(identifiant, côté couloir/fenêtre)
- Client(nom, majeur ou mineur, réduction)
- Réduction(type, pourcentage)

Si en l'état, la modélisation de la base de données semble prête à l'emploi en traduisant chaque entité évoqué par une table, plusieurs aspects pratiques imposent de modifier les relations et les tables précédentes.

L'entité Place qui référence un siège dans un train et une voiture donnée est contrai-

gnante en pratique même s'il semble intuitif de modéliser une place de la sorte. Par exemple, en se projetant, la création d'un train va imposer la création de 8 voitures qui vont elles-mêmes engendrer la création d'une centaine de siège par voiture. Puis, lors de la création d'un Billet, il faudra relier celui-ci à un train puis à une voiture puis à un siège. Dans un soucis de simplicité et puisque le client ne peut jamais choisir ni sa place ni sa voiture (même sur le site officiel de la SNCF), on va omettre la table Place et remplir le train en commençant par la première place de la première voiture jusqu'à la dernière place de la dernière voiture.

Pour autant l'information de la place du client qui vient d'effectuer sa réservation est importante, il convient donc de conserver cette information. On va séparer la notion de Billet et de Voyage pour conserver à la fois les informations propres aux clients (numéro de place, numérode voiture et situation couloir ou fenêtre du siège) et les informations propres au trajet. La nouvelle entité Voyage contient donc : ville de départ/arrivée, date et heure de départ/arrivée, prix et nombre de siège restant côté couloir et fenêtre. L'entité Billet quant à elle va référencer l'identifiant du trajet, l'identifiant du client, le numéro de voiture et de siège ainsi que sa situation couloir ou fenêtre.

3.2 Détermination du schéma de base de données

On peut alors dresser le diagramme Entité-Relation puis le schéma de la base de données en y indiquant les relations entres les tables ainsi que les cardinalités des relations.

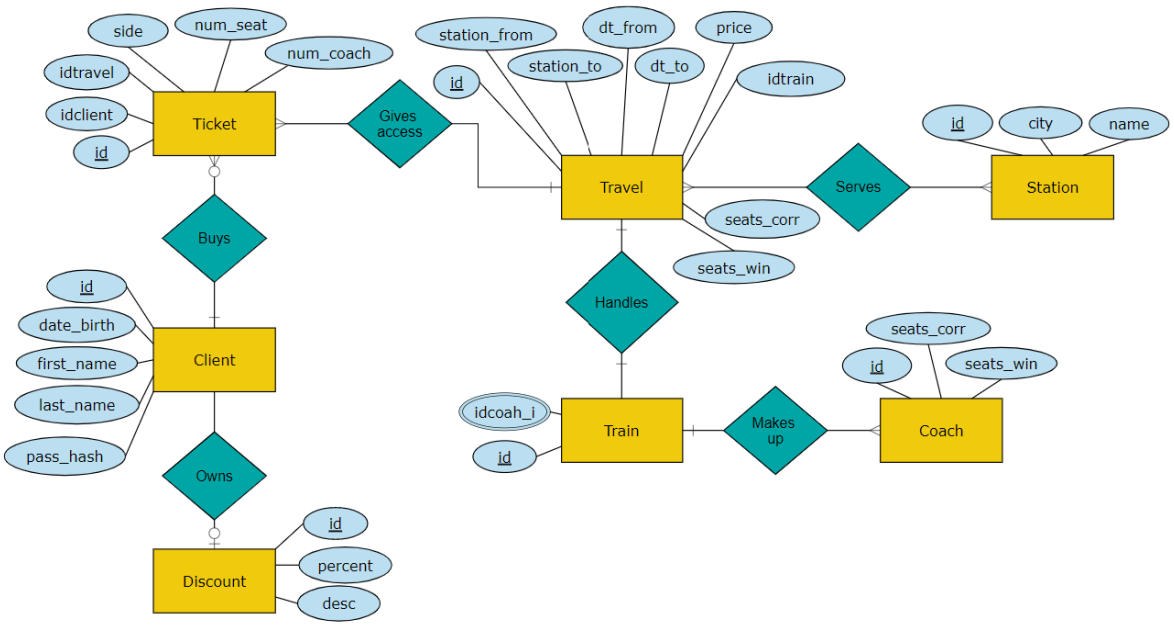

FIGURE 2 – Diagramme Entité-Relation pour l'agence de voyage SNCF

La figure 2 permet de comprendre un peu mieux les relations entre les entités et donne déjà une bonne idée des clefs primaires et étrangères qu'il va falloir utiliser pour relier les tables de la base de données.

Enfin, on donne la diagramme du schéma de la base de données. On trouve les tables avec leurs colonnes, les clefs primaires sont en gras et les clefs étrangères sont surlignées en violet. Les cardinalités des relations sont également indiquées.

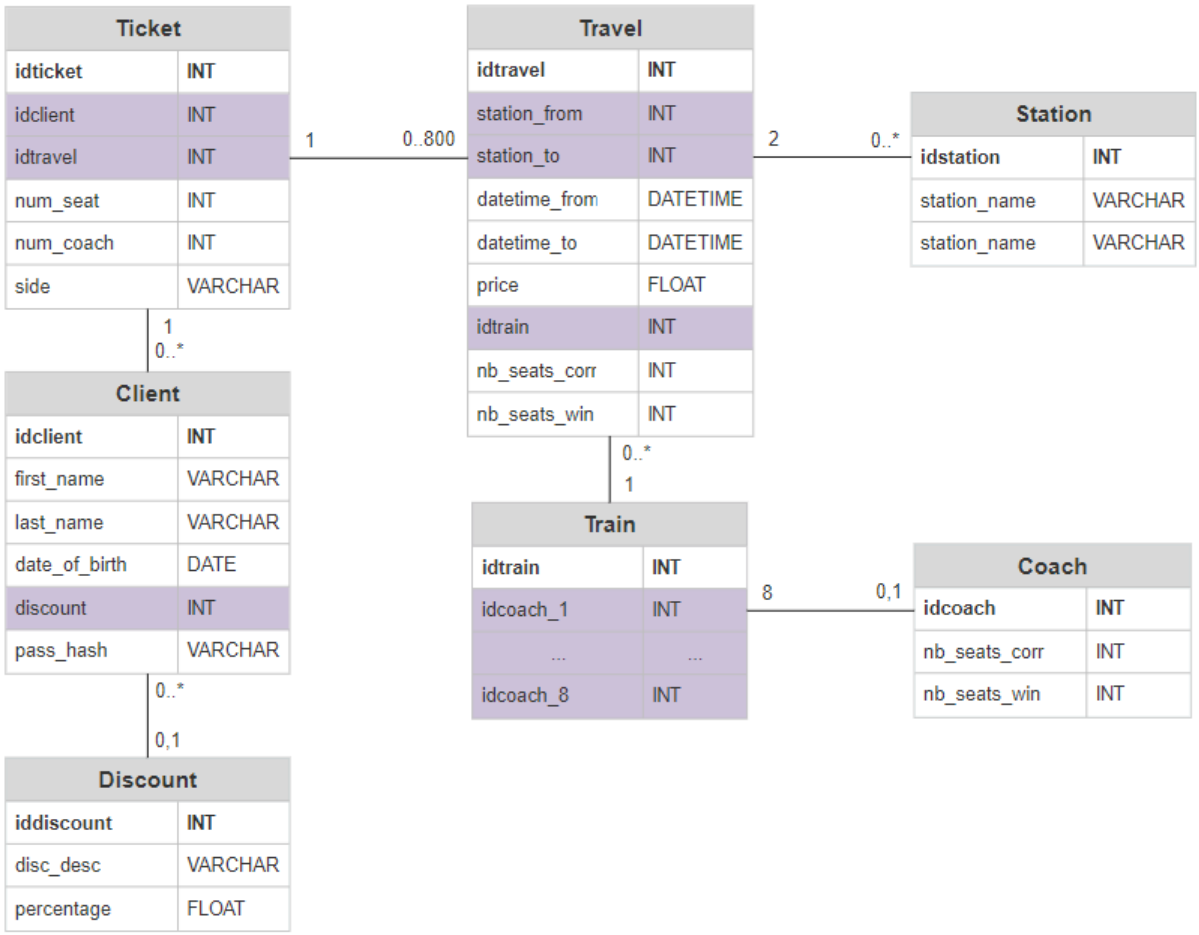

FIGURE 3 – Diagramme du schéma de la base de données

3.3 Relations et clefs étrangères

Donnons quelques détails sur les relations données par le schéma de la figure 3.

Tables autonomes : Coach, Station et Discount

On peut remarquer que 3 tables sont tout à fait autonomes en ce sens qu'elles peuvent être remplies sans la contrainte d'une clef étrangère. Ces tables sont celles engagées dans des relations de type `0..* 0.1` à savoir **Coach**, **Station** et **Discount**. Intuitivement, on peut se dire qu'une voiture est l'élément atomique et insécable d'un train, une voiture a donc une existence en elle-même. De même, une gare peut être vue comme un élément indépendant et peut être engagé ou non dans une relation sans pour autant dépendre d'une autre entité. Le même raisonnement peut être appliqué pour la table Discount qui référence les différentes cartes de réduction.

Table Train

La table **Train** est composée de 9 colonnes et est rigide. Un numéro *idtrain* identifie de manière unique un train et des identifiants *idcoach_i* renseignent les voitures qui lui sont rattachées (numérotés de 1 à 8). Ici, on fixe donc le nombre de voiture à 8 alors que les trains TGV à forte affluence en ont 18 car ils sont composés de deux trains de 9 voitures.
En vérité, on pourrait facilement adapter la table. Il suffit de rajouter les colonnes *idcoach_i* pour `i` allant de 9 à 18. Puis dans le cas d'un train qui n'aurait pas 18 voiture, les identifiants seraient passer à 0 pour matérialiser l'inexistence de la voiture pour le train en question. On comprend donc la cardinalité `8` qui symbolise le fait qu'un train possède exactement 8 voiture alors qu'une voiture est attachée à, au plus, un seul train.

Table client

La table **Client** stocke l'ensemble des clients de l'agence de voyage. Les clients sont identifiés par un numéro unique et la table référence également le nom et prénom du client, sa date de naissance, sa carte de fidélité ainsi que le hash de son mot de passe. On ne discutera pas de la dernière colonne, la justification de son existence étant propre à l'implémentation logicielle, relevant plus d'un aspect pratique que d'un véritable choix de modélisation, elle sera discutée dans la partie appropriée. Cette colonne dans une implémentation réelle n'apparaîtrait pas.
Un client est souscripteur ou non d'une carte de fidélité mais générateur de billet. Un client a donc au plus une carte de fidélité d'où la relation de type `0.1`. Mais un client peut réserver plusieurs voyages ou aucun, il est donc naturel de trouver une relation de type `0..*`.

Table Ticket

La table **Ticket** stocke les billets qui sont réservés par les clients. Un billet est identifié par un numéro unique, l'identifiant du client, le numéro du trajet auquel il est rattaché ainsi que la place matérialisée par son numéro de siège et de voiture et sa situation couloir ou fenêtre. Un billet est propre à un seul et unique client et à un seul et unique voyage d'où les relations de type `1` qui sont indiquées sur le schéma de la table.

Table Travel

La table **Travel** représente les voyages qui sont opérés par la SNCF et qui sont donc disponible à la réservation. Un voyage se traduit par un numéro unique, la gare de départ et d'arrivée, la date et l'heure de départ et d'arrivée, un prix, le train qui assure le trajet ainsi que le nombre de places restantes côté couloir et côté fenêtre.

3.4 Déclenchement d'opérations sur écoute d'événement

Si les tables peuvent être dans un premier temps, rempli par l'administrateur de la base de données, c'est, à terme, les administrateurs de l'agence de voyage SNCF qui vont compléter ces tables. Il faut donc assurer un minimum de garantie quant au bon remplissage de la base de données. Cela passe par exemple par une limitation du nombre d'opération que les administrateurs de l'agence doivent réaliser. Par exemple, si les administrateurs

doivent ajouter un train, il faut pouvoir garantir que les voitures lui soient bien rattachées. Cette opération d'affectation des voitures étant périlleuse, on peut imaginer une solution qui compléterait automatiquement le train en sélectionnant les voitures disponibles ou, a minima, en en créant de nouvelles.
C'est la deuxième option qui a été retenue dans un premier en raison de sa facilité d'implémentation. On a joute donc un *trigger* à la table **Train** qui se déclenche avant l'insertion d'une ou plusieurs lignes, autrement dit sur l'événement `BEFORE INSERT` en jargon SQL. Avant l'insertion, 8 nouvelles voitures sont créées et les identifiants de ces voitures sont passés comme valeurs des attributs `idcoach_i`.

De la même manière, lorsqu'un billet est réservé ou annulé, il faut décrémenter ou incrémenter le nombre de place disponible pour le trajet correspondant et pour le côté (couloir ou fenêtre) adéquat soit donc sur les événements `AFTER INSERT AFTER DELETE`.

La définition des différents *triggers* sont consultables en Annexe B ou directement dans le code source de l'application.

4 Architecture logicielle

Dans cette partie, on présente dans un premier les bases logicielles utilisées pour l'implémentation de la solution puis on détaille quelques aspects de celle-ci dans un second temps.

4.1 Choix technologiques et spécifications

Comme expliquée dans la deuxième partie de ce rapport, la technologie côté serveur choisie est NodeJS. Il s'agit d'une technologie serveur écrite en Javascript et qui tire partie du moteur V8 Engine (plus d'informations [ici]). NodeJS jouit d'un écosystème très riche et orienté vers les technologies du Web. NodeJS supporte par exemple les frameworks frontend Vue ou React. Dans ce projet, on utilisera le framework *Express*. *Express* permet de créer un squelête d'application Web avec des fonctionnalités de gestion des requêtes, un interfaçage très simple avec des moteurs de templates répandus (EJS, PUG, etc.) ou encore des fonctions de rendering de documents.

Squelête d'aplication Web

La figure 4 ci-contre montre l'arborescence créée par *Express*. Détaillons les différents éléments qui s'y trouve.

Commençons par le fichier `app.js` qui est le fichier centrale de l'application. Il permet de définir toute les dépendances de celle-ci ainsi que de paramétrer les modules utilisés, les gestionnaire de requêtes, les routeurs d'URL, le moteur de templates etc. Dans ce fichier, on crée donc une application *Express* avec la ligne `var app = express()`. Puis toutes les lignes suivantes utilisant la fonction `use` permet d'ajouter des dépendances à l'application. Par exemple, la ligne `app.use(express.urlencoded( extended: false ))` permet d'intégrer le parsing des arguments passés dans l'URL.
Ce fichier est un module et l'instance d'application `app` ainsi créée doit donc être exportée

avec la commande `module.exports = app`.

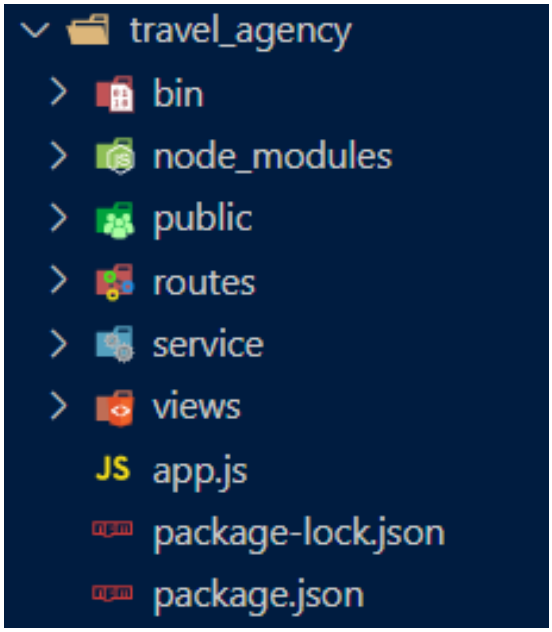

FIGURE 4 – Squelête de l'application Web

En regardant attentivement, on peut remarquer que l'application utilise des fichiers provenant du dossier `routes`. Les fichiers de ce dossier exportent des routeurs d'URL. On peut voir ces fichiers comme des sous applications qui ont en charge des tâches précises vis-à-vis de l'application globale. Dans le cas de notre application, on trouve 4 fichiers : `admin.js`, `booking.js`, `index.js` et `users.js`. Le premier fichier `admin.js` a en charge la plateforme d'administration qui est détaillée après. Le fichier `booking.js` définit le parcours client depuis la recherche d'un trajet à la génération d'un billet. Le fichier `index.js` se charge simplement de servir la page d'accueil du site. Enfin, `users.js` est le routeur qui traite toutes les requêtes relatives à la gestion des comptes (création de compte, connexion, modification d'informations, etc.). De cette manière, on peut développer quasi indépendemment les différentes fonctionnalités du site web tout en gardant organiser l'arborescence.
Notons également la présence du dossier `service` qui définit la connexion avec la base de données avec notamment un fichier `conf.json` qui renseigne les paramètres de connexion.

Les autres dossiers sont plus classiques, le dossier `views` contient les templates de pages HTML qui sont générées à la volée par le serveur en fonction des paramètres qui sont passés au moteur de template EJS qui est utilisé dans le cadre de ce projet. Le fichier `public` contient des fichiers statiques servis à la demande : fichiers javascript, feuille de style CSS ou encore des images.

Enfin le dossier `bin` est, ce qui est communément appelé, l'*application endpoint*. Il contient les fichiers qui seront directement appelés lors du lancement du serveur. Dans notre cas, il n'y a qu'un seul fichier qui démarre l'application Web mais des fichiers de test d'application pourrait également être intégrés.

Modules complémentaires utilisés

Autour d'*Express* gravitent des dizaines de librairies qui permettent notamment d'interfacer l'application avec une base de données au moyen d'un connecteur. Comme l'utilisation d'ORM est interdite dans le cadre de ce projet, on utilisera le module `mysql2` qui permet d'ouvrir une connexion avec la base de données et d'exécuter des requêtes brutes en fournissant tout de même une méthode de parsing d'argument. L'extrait de code suivant donne un exemple de formulation de requête avec un objet `Connection` référencé par la variable `con`.

```
/* Récupération des paramètres passés dans la requête */
var firstName = req.body.first_name,
    lastName = req.body.last_name;

/* Préparation de la requête */
var stmt = "SELECT * FROM client WHERE last_name = ? AND first_name = ?";

/* Exécution de la requête avec les parmètres */
con.query(stmt, [firstName, lastName], (err, data) => {
    /* Affichage de l'erreur si besoin sinon affichage des données
    ↪  réupérées */
    if (err) throw err;
    else {
        console.log(data);
    }
});
```

Les `?` sont des caractères de substitution pour les arguments passés à la requête. Lors de l'appel à la méthode `query`, la chaîne de caractère est interpolée pour y insérer les paramètres passés sous forme de tableau. Ce mécanisme permet aussi d'échapper toutes les chaînes de caractère ce qui garantit ainsi un aspect de sécurité au regard des injections SQL les plus courantes.

`Moment.js` peut également être cité dans liste des modules utilisé. Il permet une manipulation très simple des objets datetime. Ainsi, la manipulation et l'affichage de ces objets peut être faite en choisissant des *locales* (région du moment avec un format de date/heure particulier) ainsi que des format plus commode pour des êtres humains.

4.2 Structure du site et navigation

Le site comporte 2 grandes parties qui sont celles évoquées précédemment dans le cahier des charges de l'application : une partie réservée au client, l'autre aux administrateurs de l'agence de voyage.

La navigation démarre à la page d'accueil et permet à l'administrateur de démarrer immédiatement sa navigation. L'administrateur peut se rendre sur la plateforme dédiée à la gestion de la base de données. cette plateforme est protégée par une authentification *Basic Auth* avec les identifiants suivant :

— Utilisateur : `admin`
— Mot de passe : `AdminPass`

Le client peut démarrer en se connectant ou en créant un compte ou même en lançant immédiatement une recherche. La création du compte requiert un nom, un prénom, une date de naissance, la sélection d'une carte de réduction (possibilité de choisir aucune carte évidemment) ainsi qu'un mot de passe avec vérification. Les figures suivantes donnent un aperçu de quelques pages accessibles mais le mieux est de se faire son idée en installant directement l'application en suivant le guide de la partie suivante. L'Annexe C fournit tous les schémas d'URL et les méthode HTTP associées au besoin.

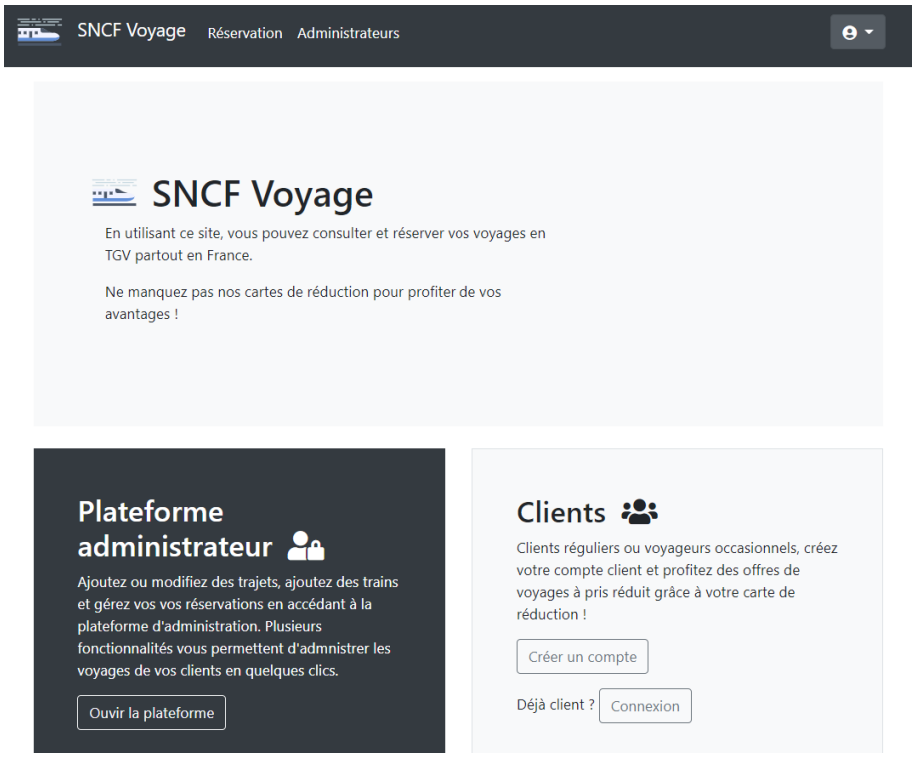

FIGURE 5 – Page d'accueil du site Web

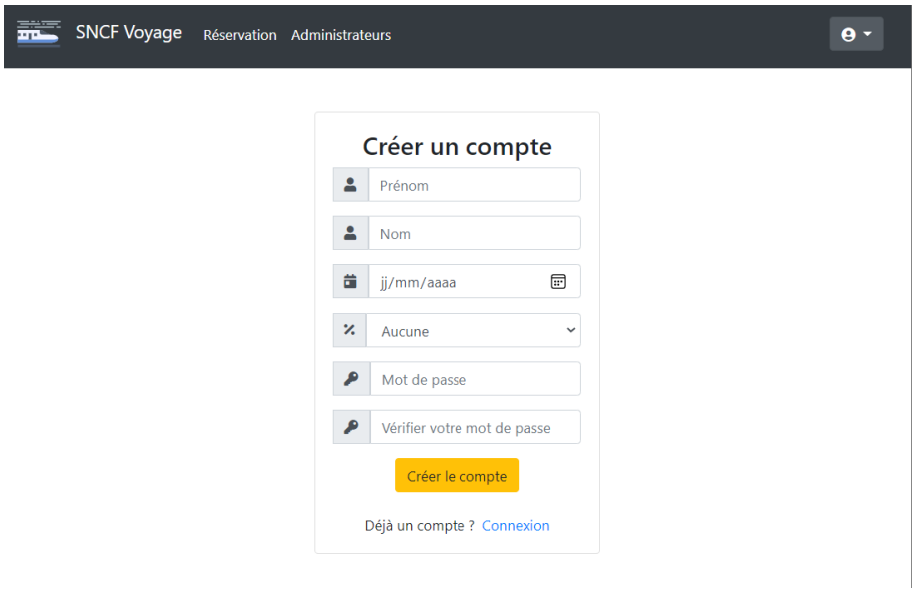

FIGURE 6 – Page de création de compte

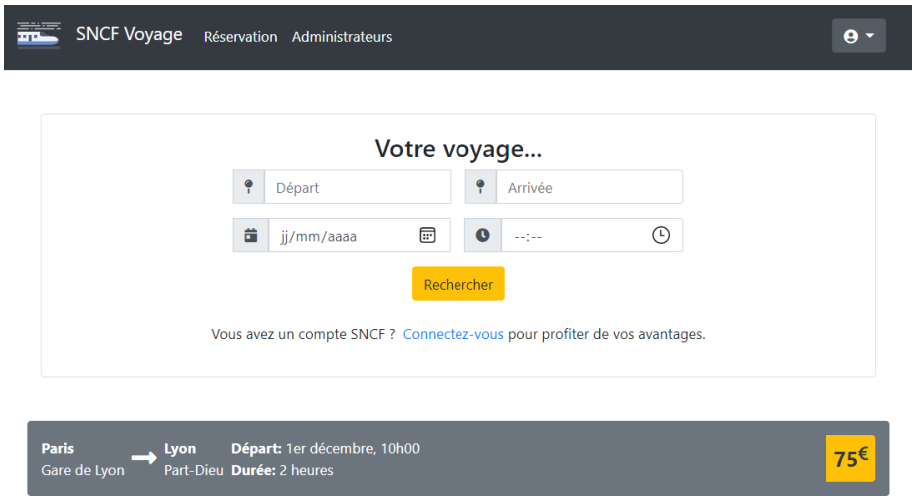

FIGURE 7 – Page de recherche de trajet

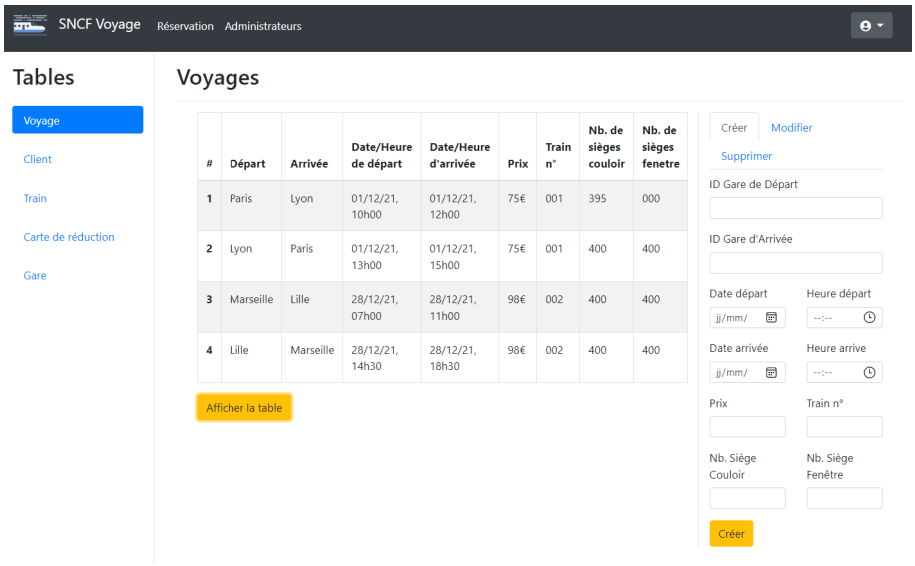

FIGURE 8 – Interface d'administration de la base de données

L'intégralité du site a essayé de respecter l'engagement du cahier des charges relatif à la facilité de navigation et une attention particulière a été portée sur la forme. On pourra remarquer que la feuille de style CSS du dossier `public/stylesheets` est quasi et ne reflète donc pas la mise en forme des pages HTML. L'ensemble du site utilise intensivement la collection d'outils Bootstrap qui définit des classes et pseudo-éléments CSS. Il suffit pour l'utiliser d'ajouter le lien CDN de Bootstrap ainsi que les dépendances (bundle javascript et jQuery). On peut voir l'ajout des ces classes dans les templates (dossier `views`).

4.3 Quelques pistes d'amélioration

Dans la solution actuelle, il n'y a pas de modèle d'utilisateur à proprement parler. Les clients qui passent commandes sont confondus avec les utilisateurs du site. Cela est pratique puisque on n'a pas à dupliquer les données clients et utilisateurs. Mais, ce mécanisme ne permet pas de gérer des sessions et d'avoir une bonne robstesse pour sécuriser les connexion. A l'heure actuelle, la session est persisté par le moyen d'un cookie appelé `user` et qui est récupéré dans les requêtes pour pouvoir identifier le client qui réserve un billet.

D'autre part, les valeurs des arguments ne sont pas toutes validées avant d'exécuter les requêtes. Un minimum est assuré grâce aux formulaires HTML qui permettent de définir certaines bornes pour l'utilisateur mais qui ne sont pas suffisantes. Des vérifications au niveau du serveur sont, dans quelques fonctions faites, mais doivent être implémenté aussi au niveau de la base de données.

Enfin, on peut davantage tirer parti les mécanismes qui sont offerts par la base de données. L'utilisation de fonction ou l'implémentation de nouveaux *trigger* devraient pouvoir de gagner et en robustesse mais aussi en efficacité. Notamment, il faudrait absolument implémenter les mécaniques de transaction lors des réservation ou d'autres modifications pour s'assurer que les opérations réalisées soient licites.

Mode d'emploi

Dans cette dernière partie, on détaille les pré-requis et l'installation de l'application de gestion de la base de données.

4.4 Pré-requis

Tout d'abord, il faut installer les dépendances de l'application. L'application étant écrite avec la technologie NodeJS, il faut bien sûr l'avoir installée. Le lien est disponible [ici]. Un script Python est fourni pour peupler quelques lignes de la base de données, il faut donc avoir une version de Python 3 installée si vous voulez l'utiliser.

4.5 Installation locale

Récupération de la codebase

Commencez par télécharger l'archive `travel_agency.rar` et la décompressser.
Cette archive est également disponible sur un repo Github dont l'adresse est [ici]. Vous pouvez donc cloner le repo :

```
git clone https://github.com/ebenkara15/travel_agency.git
```

Installation de la base de donnés

L'ensemble des fichiers relatifs à la base de données est contenu dans le dossier nommé `./travel_agency_db`. Le sous-dossier `./travel_agency_db/db_dump` continent tous les scripts SQL utiles pour créer la base de données. Ouvrez MySQL Workbench, connectez vous au serveur. Puis ouvrez le fichier `travel_agency_install.sql` (*File* > *Open SQL Script*) et exécutez le. La base de données devrait se créer (rafraîchir l'onglet *Schéma* à gauche) avec toutes les tables et les *triggers*.

Pour peupler la base de données, un script Python est fourni dans le dossier relatif à la base de données, il est nommé `populate.py`.
Si vous souhaitez l'utiliser, installez les packages nécessaires qui sont spécifiés dans le fichiers `requirements.txt`.

```
pip install -r requirements.txt
```

Enfin, lancez le programme qui va demander les informations de connexion à la base de données.

```
python populate.py
```

Installation de l'application Web

Ouvrez un terminal et verifiez que l'installation de NodeJS et `npm` sont effectives :

```
node --version
npm --version
```

Vous devriez voir les numéros de version de 2 exécutables. Si vous avez bien installé les programmes mais que les commandes précédentes ne marchent pas, les chemins des installations ne soient pas reporter dans votre `PATH`. Ajoutez les si ce n'est pas le cas.

Avant de passer à l'installation, il faut modifier les paramètres de connections à la base de données. Modifiez le fichier `conf.js` du dossier `service` dans le répertoire de l'application, c'est-à-dire le dossier `./travel_agency`. Si tout fonctionne, exécutez les commandes suivantes :

```
npm install

node start
```

L'application Web est désormais disponible à l'adresse `localhost` (ou `127.0.0.1`) sur le port 3000 i.e. `localhost:3000`. Vous pouvez également cliquer [ici] pour l'ouvrir une fois celle-ci lancée.

NOTE : pour se connecter à la plateforme utilisateur, il faut utiliser les identifiants suivants :

— Utilisateur : `admin`
— Mot de passe : `AdminPass`

A Enoncé du sujet

Nous voulons faire la gestion d'une agence de voyage SNCF. Un billet de voyage est identifié par cette agence, par son numéro et se caractérise également par :

- la ville de départ/arrivée
- la gare de départ/arrivée
- la date de départ/arrivée
- l'heure de départ/arrivée
- la prix du billet

Le billet va être utilisé dans un train identifié par son numéro et comporte plusieurs voitures. Une voiture est identifiée par son numéro et le nombre de places disponibles. Chaque place est identifiée par un numéro et sa situation fenêtre/couloir.

La réservation concerne le client identifié par son nom, il peut être adulte ou enfant et il a droit ou non à une réduction. Un réduction est identifiée par son type (J-8, J-30, Découverte, etc.) et se caractérise par un pourcentage. La réservation s'achève par une confirmation ou une annulation. Il faut calculer le prix total à payer déduit de la réduction.

B Déclencheurs de la base de données

Table Train

```
CREATE DEFINER=`root`@`localhost` TRIGGER `train_BEFORE_INSERT` BEFORE
↪  INSERT ON `train` FOR EACH ROW BEGIN

-- Sélection de la valeur maximum de la colonne idcoach
SET @idmax = (SELECT MAX(idcoach) FROM coach);

-- Création des 8 voitures à affecter au train en cours de création
INSERT INTO coach (idcoach)
        VALUES (@idmax+1),(@idmax+2),(@idmax+3),
               (@idmax+4), (@idmax+5), (@idmax+6),
               (@idmax+7), (@idmax+8);

-- Affectation des voitures au nouveau train
SET NEW.idcoach_1 = @idmax + 1, NEW.idcoach_2 = @idmax + 2,
    NEW.idcoach_3 = @idmax + 3, NEW.idcoach_4 = @idmax + 4,
    NEW.idcoach_5 = @idmax + 5, NEW.idcoach_6 = @idmax + 6,
    NEW.idcoach_7 = @idmax + 7, NEW.idcoach_8 = @idmax + 8;
END
```

Table Ticket

```
CREATE DEFINER=`root`@`localhost` TRIGGER `ticket_AFTER_INSERT` AFTER
↪  INSERT ON `ticket` FOR EACH ROW BEGIN

-- Modification de la table `travel`
UPDATE travel t

-- Décrémentation du nombre de place
SET    t.nb_seats_corr = IF(NEW.side='corr', t.nb_seats_corr - 1,
↪  t.nb_seats_corr),
           t.nb_seats_win = IF(NEW.side='win', t.nb_seats_win - 1,
           ↪  t.nb_seats_win)

-- Sélection du voyage
WHERE  t.idtravel = NEW.idtravel;
```

```
CREATE DEFINER=`root`@`localhost` TRIGGER `ticket_AFTER_DLEETE` AFTER
↪  DELETE ON `ticket` FOR EACH ROW BEGIN

-- Modification de la table `travel`
UPDATE travel t

-- Décrémentation du nombre de place
SET    t.nb_seats_corr = IF(OLD.side='corr', t.nb_seats_corr - 1,
↪  t.nb_seats_corr),
           t.nb_seats_win = IF(OLD.side='win', t.nb_seats_win - 1,
           ↪  t.nb_seats_win)

-- Sélection du voyage
WHERE  t.idtravel = OLD.idtravel;
```

C Schéma d'URL

Page d'accueil

GET `/` : page d'accueil du site

Plateforme administrateur (protégée par Basic Auth)

GET `/admin` : accueil de la plateforme
GET `/admin/api/get/:table` (AJAX only) : renvoie les données contenue dans la table `:table`
POST `/admin/api/create/:table` (AJAX only) : créer une nouvelle ligne dans la table `:table` avec les paramètres encodés dans l'URL
POST `/admin/api/update/:table` (AJAX only) : met à jour une ligne dans la table `:table` avec les paramètres encodés dans l'URL
POST `/admin/api/delete/:table` (AJAX only) : supprime une ligne dans la table `:table` avec les paramètres encodés dans l'URL

Clients

GET `/users` (Redirection) : redirige l'utilisateur vers la page d'identification ou la page d'accueil si un utilisateur est déjà connecté
GET `/users/login` : page d'identification ou redirection vers la page d'accueil si un utilisateur est déjà connecté
POST `/users/login` : soumet un formulaire de connexion. Renvoie vers la page de recherche en cas de succès
GET `/users/register` : affiche la page de création de compte ou redirige vers l'accueil si un utilisateur est déjà connecté
POST `/users/register` : soumet un formulaire de création de compte. Renvoie vers la page de connexion en cas de succès
GET `/users/logout` : déconnecte un utilisateur
GET `/users/account` : affiche le compte de l'utilisateur connecté
POST `/users/edit/account/:discount` (AJAX only) : modifie la carte de réduction de l'utilisateur avec une vérification d'éligibilité de l'identifiant `:discount`

Réservations

GET `/booking` : page d'accueil de recherche de trajet
POST `/booking/search` : lance une recherche avec les parmètres encodés dans l'URL
GET `/booking/:id` : affiche les détails du trajet identifié par `:id`
POST `/booking/travel/:id/:side` (AJAX only) : acte la réservation pur le trajet `:id` avec un siège côté `:side`
GET `/booking/travel/:id/confirm` : affiche le billet réservé pour le trajet `:id`
GET `/booking/travel/:id/cancel` ; annule le billet réservé pour le trajet `:id`