Blackmarket

Dossier de Projet

Titre Professionnel Développeur-se Logiciel

Yacine Chibane - De Jésus

2017

Sommaire

Remerciements

Tous d'abord, je souhaites remercier l'entreprise GFI qui m'a accueilli au cours de l'année qui vient de s'écouler.

Je tiens, ensuite, à remercier l'organisme Simplon.co de Montreuil pour m'avoir offert l'opportunité de me former au développement informatique et web, sans avoir aucune connaissance préalable dans ce domaine. Simplon.co m'a donné une seconde chance après un parcours professionnel à l'issue incertaine et m'a permis de trouver une vocation qui m'a immédiatement motivé et présenté l'occasion de construire une carrière durable dans ce domaine aussi passionnant que porteur.

Je tiens également à remercier les différents formateurs et l'équipe pédagogique de Montreuil qui ont su me faire confiance pour me sélectionner parmi les nombreux candidats à la quatrième promotion.

Je tiens aussi à remercier mes camarades qui m'ont accompagné durant ces quelques mois de formations et la période d'alternance. Pour leur aide, soutien et partage dans cette période d'apprentissage aussi dure que captivante.

Enfin, je souhaite remercier du fond du coeur ma femme sans qui rien de tout cela n'aurait été possible, pour son soutien inconditionnel dans cette période de transition professionnelle que n'aura pas été de tout repos et qui a demandé certains sacrifices non-négligeables.

Résumé du projet en Français

Je suis passionné à la fois par le développement Web et les sneakers (chaussures de sports de collections). C'est pourquoi pour ce Titre Professionnel, j'ai choisi de lancer un de mes objectifs personnels: créer une marketplace dédiée aux sneakers.

Cette marketplace prend la forme d'un application web responsive qui peut s'adapter à toutes les tailles d'écran. Les utilisateurs peuvent y créer un profil et mettre des produits en vente, tout en naviguant parmi les autres produits disponibles à l'achat et les ajouter à leur collection.

Cette application remplira les critères de compétences requises pour l'obtention de ce diplôme grâce aux fonctionnalités suivantes: une interface utilisateur permettant la création d'un compte et de produits, une authentification et l'utilisation de données enregistrées dans une base de données.

J'ai choisi de développer ce projet en Javascript dans un souci de cohérence entre mes expériences et l'uniformité des technologies utilisées. J'ai employé des frameworks JS pour les parties Front et Back de l'application, ainsi que d'autres outils de compilation et gestion de tâches.

VueJS m'a permis de générer l'application côté client alors que Node/Express/ MongoDB ont servi pour la création de l'API REST et délivrer les données. Il s'agit là d'outils et frameworks que je ne connaissais pas ou que je voulais approfondir avant de démarrer cette application. J'ai donc profité de ce projet pour découvrir ces nouveaux outils et élargir mes connaissances de manière générale.

Résumé du projet en Anglais

I am passionate about web development and I am passionate about sneakers (collectible sports shoes). So for the purpose of this Professional Title diploma I decided to kickstart one of my personal goals: launch a marketplace for sneakers.

This marketplace will take the form of a web application with a responsive design which will adapt to all screen sizes. Users will be able to create a profile and put products they own up for sale, as well as browse through other users' items available for purchase to add them to their collection.

This application will fit the requirements of this diploma through the following functionalities: user interface allowing user account and product creation, authentication and use of data stored on a database.

Coming from a Javascript training, this is the language that was used throughout the project. JS frameworks were used for the Front and Back ends of the app, along with other building and development tools. Notably, VueJS was the front-end framework used to generate the client app, while Node/Express/MongoDB were used to create the RESTful API in order to serve and store the views and data. I discovered some of these tools while I further explored others as my aim with this project was to improve my overall developper skills.

Liste des compétences déployées

Le projet Blackmarket a nécessité la mise en oeuvre de plusieurs des compétences requises pour l'obtention du Titre professionnel Développeur-se logiciel.

En effet, ce projet a fait appel à des connaissances "full-stack". C'est à dire qu'il impliquait des notions de développement à la fois "front-end", afin de réaliser une interface graphique accessible à tous, et "back-end" afin de déployer un site pouvant servir les données.

1. Maquetter une application

En ce sens, l'aspect 'Front-End' a nécessité la réalisation préalable de maquettes afin de concevoir à la fois les vues et les interactions que les utilisateurs auraient avec les différentes fonctionnalités de l'application (voir Conception Fonctionnelle).

2. Développer une interface.

Ces maquettes ont ensuite servies de guide et de base graphique pour le développement de l'interface visuelle de l'application en HTML, CSS et Javascript. Ces trois langages ont été utilisés pour donner, respectivement, une structure, un habillage et une interaction à mon app.

En l'occurence un framework a été utilisé afin de réaliser cette interface de manière optimale et ainsi éviter des répétition de codes et de fichiers. J'ai choisi d'utiliser VueJS et les motivations de ce choix sont détaillées dans les Spécifications techniques.

3. Développer une application simple de mobilité numérique (responsive).

L'avènement des appareils mobiles (smartphone et tablette) a aujourd'hui rendu difficilement concevable de proposer un site ou une application web n'offrant pas de s'adapter à ces différents formats. Il n'est plus possible de se contenter de créer un format desktop uniquement. Et même celui-ci doit répondre à une telle variété de résolutions différentes qu'il est indispensable de rendre son site/app 'responsive'. Au delà de la simple taille de l'écran, c'est toute l'interaction avec l'utilisateur qui peut être altérée.

C'est pourquoi le développement Front-End a fait appel au responsive design, une méthode de conception qui inclue une modification de certains éléments pour s'

adapter lorsque la résolution du support change. Cette méthode a également un impact sur la charge de travail concernant le maquettage, puisque l'affichage des vues doit de surcroît être conçu pour ces différents supports.

4. Concevoir une base de données

La conception des interactions qu'aurait un utilisateur avec l'application a, par la suite, conduit à une réflexion concernant l'accès et l'utilisation de données.

Ceci nous amène à la partie 'Back-end' de la conception de l'application. Elle nécessite que les utilisateurs ouvre un compte pour déposer leurs annonces, y ajouter des informations et des photos, et s'approprier les produits en ventes des autres utilisateurs disponibles sur la marketplace. La base de données doit donc être conçue pour accueillir et répertorier ces informations.

Il a donc fallu schématiser ces données pour répertorier à la fois les utilisateurs et les produits puis faire communiquer ces données entre elles (voir les schémas dans la section Realisation Technique - Back-End).

5. Mettre en place une base de données

Cette compétence a nécessairement été mise en oeuvre afin de fournir une base de données pour répertorier et sauvegarder les informations liées aux utilisateurs et aux produits. Il a ensuite fallu configurer une connexion à cette base de données.

6. Développer des composants d'accès aux données

La partie 'Back-end' est composé d'une api développée pour acheminer et/ou enregistrer les données de la base depuis et vers l'application. Ainsi, cette API comporte des modules/contrôleurs qui vont se charger d'assurer un accès pour le traitement de ces informations.

7. Développer des pages web en lien avec une base de données.

L'application Blackmarket repose quasiment entièrement sur des données puisqu'il s'agit d'une marketplace. Ces composants 'back-end' dédiées à l'accès aux données doivent donc être liées à des composants 'front-end' afin de tirer profit de ces datas au sein de l'application. La plupart des pages du site a donc été créée pour afficher ces données et ont donc nécessité de mettre en oeuvre cette compétence.

Concept et Fonctionnalités attendues

Concept

Pour l'obtention de ce Titre professionnel de Développeur-se Logiciel, j'ai choisi de réaliser une application web qui propose un service de marketplace.

Il s'agit donc de proposer une interface qui propose des produits disponibles à l'achat. Ces produits sont mis en vente par les différents utilisateurs et donc rattaché à leur compte.

En l'occurence, ces produits seront des 'sneakers'. Autrement dit, il s'agit d'acheter et de vendre des paires de chaussures de sports de collection, ayant une valeur de revente élevée. Mais celles-ci peuvent aussi être usée ou avoir une valeur de revente faible. L'intérêt de créer une telle plateforme réside dans le prélèvement d'une commission sur chaque transaction.

L'interface propose de visionner les différents produits et de les consulter librement sans nécessiter une authentification. En revanche, les interactions avec les produits nécessitent la creation d'un compte et d'être connecté. Cela va permettre aux utilisateurs d'acheter les produits et d'en proposer à la vente, ce qui va modifier la quantité de produits disponibles à l'achat.

A noter, pour les besoins de ce Titre Professionnel, les fonctionnalités réelle d'achat n'ont pas été déployées dans la réalisation de cette application. L'achat est remplacé par un ajout à la collection de l'utilisateur qui souhaite acquérir le produit. Cependant, il s'agit d'un projet personnel qui est voué à être concrétisé par une mise en ligne publique et, à terme, des moyens de paiements seront mis en place.

Fonctionnalités

1. Affichage des produits disponibles

La fonctionnalité première de cette application web est d'afficher les produits disponibles à la vente. C'est cette aspect qui doit attirer les visiteurs et les inciter à s'inscrire.

Pour cette feature, il faudra donc developper une interface qui affichera la totalité des chaussures proposées à la vente par les utilisateurs. Cette page les dispose de manière horizontale sur les écrans suffisamment larges (desktop et tablette) et verticale sur les smartphones.

Par arriver à afficher ces informations, la page d'accueil doit faire appel à la collection des produits dans la base de données.

2. Formulaire de creation de compte/inscription

Lors de sa navigation sur le site, deux cas de figures peuvent se presenter à l'utilisateur. Il peut s'inscrire par lui même ou être invité à le faire lorsqu'il souhaite interagir avec un produit (acheter ou vendre).

C'est pourquoi, une page contenant un formulaire de creation de compte/profile sera nécessaire. Certaines informations seront obligatoires. Une vérification de la validité des informations fournies sera donc nécessaire en partie 'front' afin d'informer l'utilisateur de ses erreurs et en 'back' afin d'assurer la qualité des informations à enregistrer dans la base de données. Le formulaire comportera également un upload d'image pour la photo de profil.

Un lien sera mis en évidence au travers du site et dans la bar de menu, autrement l'utilisateur sera redirigé vers ce formulaire lorsque l'inscription/connexion est nécessaire.

3. Formulaire de connexion

Lorsque l'utilisateur souhaite se connecter ou accéder à des fonctionnalités qui nécessite de le faire (et qu'il ne l'a pas déjà fait préalablement), il doit être redirigé vers une page de connexion comportant un formulaire où renseigner son adresse mail (identifiant) et son mot de passe.

Pour les cas de mot de passe oublié, cette page doit être déclinée pour permettre de renseigner son adresse mail et recevoir un lien vers une autre déclinaison de ce formulaire pour réinitialiser le mot de passe.

4. **Page Profile de l'utilisateur inscrit**

Si le visiteur est inscrit sur le site, il peut gérer et mettre à jour son profil et ses informations personnelles au travers d'une interface avec un formulaire lui permettant de les modifier.

Les profils sont tous consultables par les visiteurs afin d'en savoir plus sur les 'vendeurs'. Chaque profil a donc une page 'publique' affichant une partie non-personnelle des informations.

5. **Formulaire de creation de produit pour mise en vente**

Après avoir créé son profil, l'utilisateur peut alors commencer à mettre des produits en vente. Pour cela il aura besoin d'accéder à une page contenant un formulaire pour renseigner toutes les informations du produits et ajouter des photos.

6. **Page produit**

Afin que les 'acheteurs' potentiels puissent en savoir plus sur les produits qui les intéressent, ceux-ci disposent chacun d'une page affichant plus de details. Cette page est accessible en cliquant sur le produit dans la liste présente sur la home page.

7. **Module d'ajout des produit à sa collection (achat)**

Comme expliqué plus tôt, il ne s'agit pour l'instant que d'un concept de marketplace et le processus d'achat a été remplacé par un ajout à la collection de l'utilisateur.

Après avoir accédé à la page dédiée affichant toutes les informations du produit., s'il souhaite l'avoir dans sa collection, il pourra faire usage d'une module avec un bouton d'ajout.

Organisation et déroulement du projet

La réalisation de ce projet a nécessité la mise en place d'un certain nombre de méthodes d'organisation et de travail, ainsi que l'emploi d'outils, détaillés ci-dessous.

Organisation

Ce projet a été réalisé de manière individuelle. Il n'y a donc pas eu de répartition des tâches que j'ai toutes réalisées seul. Cela m'a permis de l'organiser de manière flexible et d''assimiler de nouvelles connaissances ainsi que de conforter celles existantes.

Le projet a été réalisé dans le contexte d'un contrat de professionnalisation comprenant, chaque mois, une semaine de formation. Il a donc fallu organiser les sprints sur le temps personnel en dehors des heures de travail. Le temps de formation a lui permis d'avancer plus vite et d'obtenir de l'aide sur les aspects qui le nécessitaient.

Outils mis en place

Inspiration des méthodes Agiles

Pour organiser la réalisation de ce projet je me suis inspiré des méthodes Agiles. Dans une version simplifiée, j'ai décomposé le projet en plusieurs petites ou moyennes tâches. Je leur ai attribué un niveau de priorité et un statut qui ont déterminer l'ordre dans lequel je les ai réalisées et le temps qui leur a été alloué.

Ces tâches ont ensuite été regroupées en fonction de leur nature et de leur thématique. Ce regroupement m'a permis d'organiser des 'sprints', c'est à dire des laps de temps courts et limités durant lesquels un groupe de tâches (ou une seule si elle est plus lourde) sera réalisé.

Certaines taches ont demandé une demi journée ou moins, tandis que certaines ont demandé plus d'une journée et demi, voir deux. Sachant que ce chiffrage est à considérer pour une implication à plein temps. Or ce projet a été réalisé sur le temps personnel en dehors des heures de travail en entreprise.

Trello

Afin de suivre l'avancée du projet j'ai utilisé l'outil Trello car il permet de créer des listes et de déplacer chacun des éléments de l'une à l'autre. J'ai ainsi pu grouper les différentes tâches, par type, priorité et nature. J'ai notamment créé les listes suivantes:

- A faire (To do) - réunissant les tâches à faire et en attente d'être effectué
- En cours (Doing) - les tâches en cours de réalisation.
- Achevée (Done) - Les tâches terminées
- Backlog - Une liste générale qui sert à identifier, au départ ou à rajouter pendant le projet, les étapes à réaliser avant qu'elles ne passent dans le statut suivant (à faire).

J'ai également annoté sur chaque tâche une estimation du temps nécessaire à sa réalisation et je les ai discernées selon les catégories suivantes :

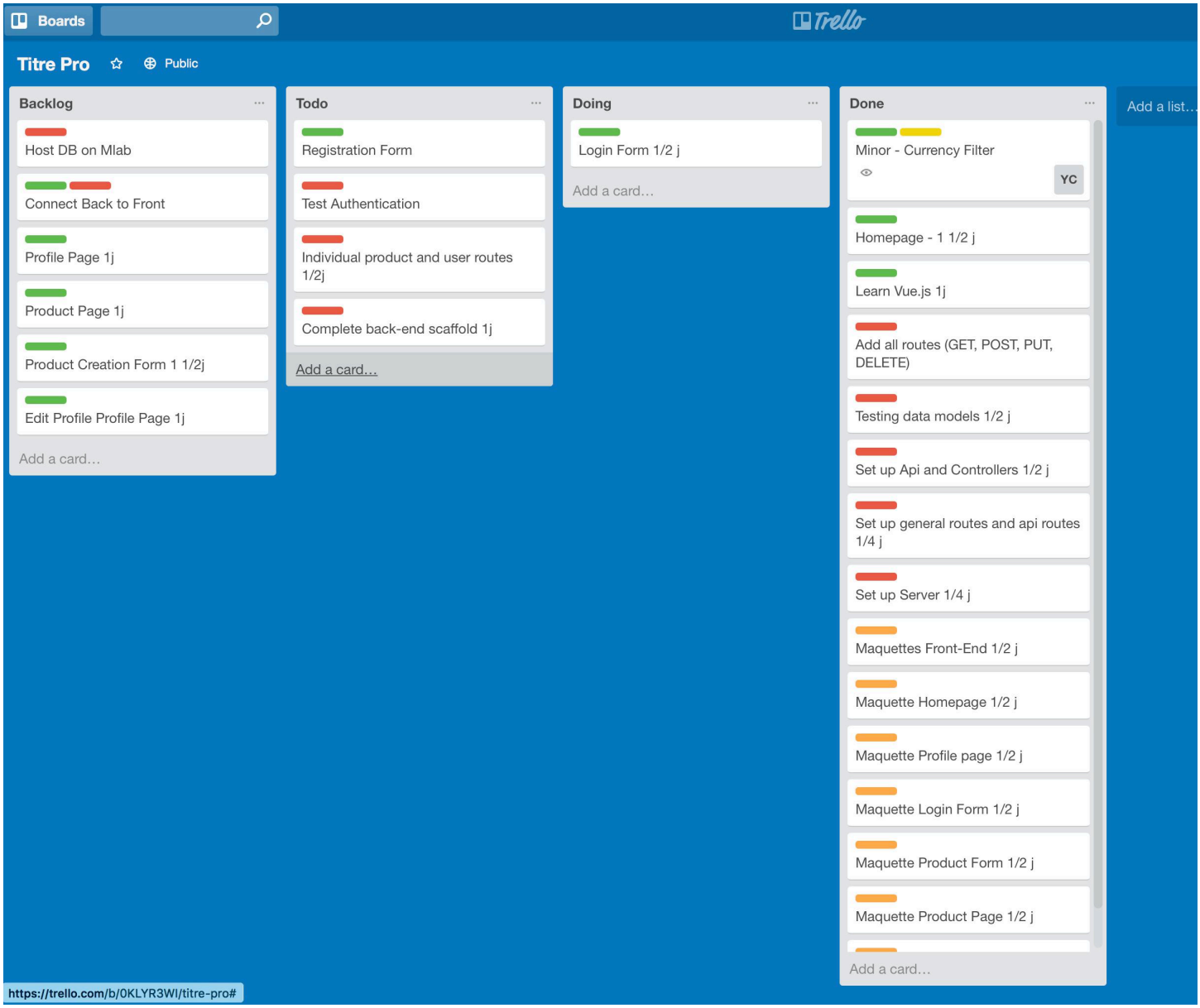
- Front-End
- Back-End
- Design

Pour cela, Trello permet d'assigner une étiquette ('label') correspondant à chacune de ces catégories avec une couleur et un nom. On peut également aux tâches une description pour y ajouter toutes les informations nécessaires.

Par ailleurs, je pouvais y annoter tout liens vers des ressources utiles au projet et ajouter des notes.

La page suivante illustre l'outil Trello par une capture d'écran.

1. *Capture du tableau Trello regroupant toutes les tâches du projet*



Git/Github

Afin de conserver le code de l'application de manière sûre et maintenable, j'ai utilisé l'outil de versionning Git et le projet a été envoyé vers la plateforme GitHub.

L'outil Git permet de sauvegarder différentes versions du code, sur l'ensemble du projet ou pour chaque fichier individuel. Cette fonctionnalité permet de marquer les différentes étapes du développement et de les communiquer aux autres contributeurs (dans un projet de groupe) ou encore de revenir à un état précédent lorsqu'un problème bloquant arrive par exemple.

Cet outil permet également de créer des branches, c'est à dire une version parallèle du projet sur laquelle on pourrait développer une fonctionnalité particulière avant de l'intégrer au corps du projet.

Il y a de nombreuses autres fonctionnalités inhérentes à l'outil Git qui permettent de gérer l'agrégation du code, vérifier des changements et bien plus encore.

La plateforme Github offre, quant à elle, de nombreux outils supplémentaires notamment sauvegarder son code en ligne en cas de perte, publier le code (sauf en cas de dossier privé), visualiser des statistiques et graphiques ainsi que les informations sur les différents ajouts et contributions au projets.

Ces outils m'ont permis de gérer mes 'sprints' au niveau de la gestion du code au travers des 'commits' (point de sauvegarde d'une version du projet/fichiers modifiés). Par exemple, un commit était créé lorsqu'une fonctionnalité était achevée ou lorsque une base de travail était posée. J'ai également pu conserver une sauvegarde de mon code en cas de perte.

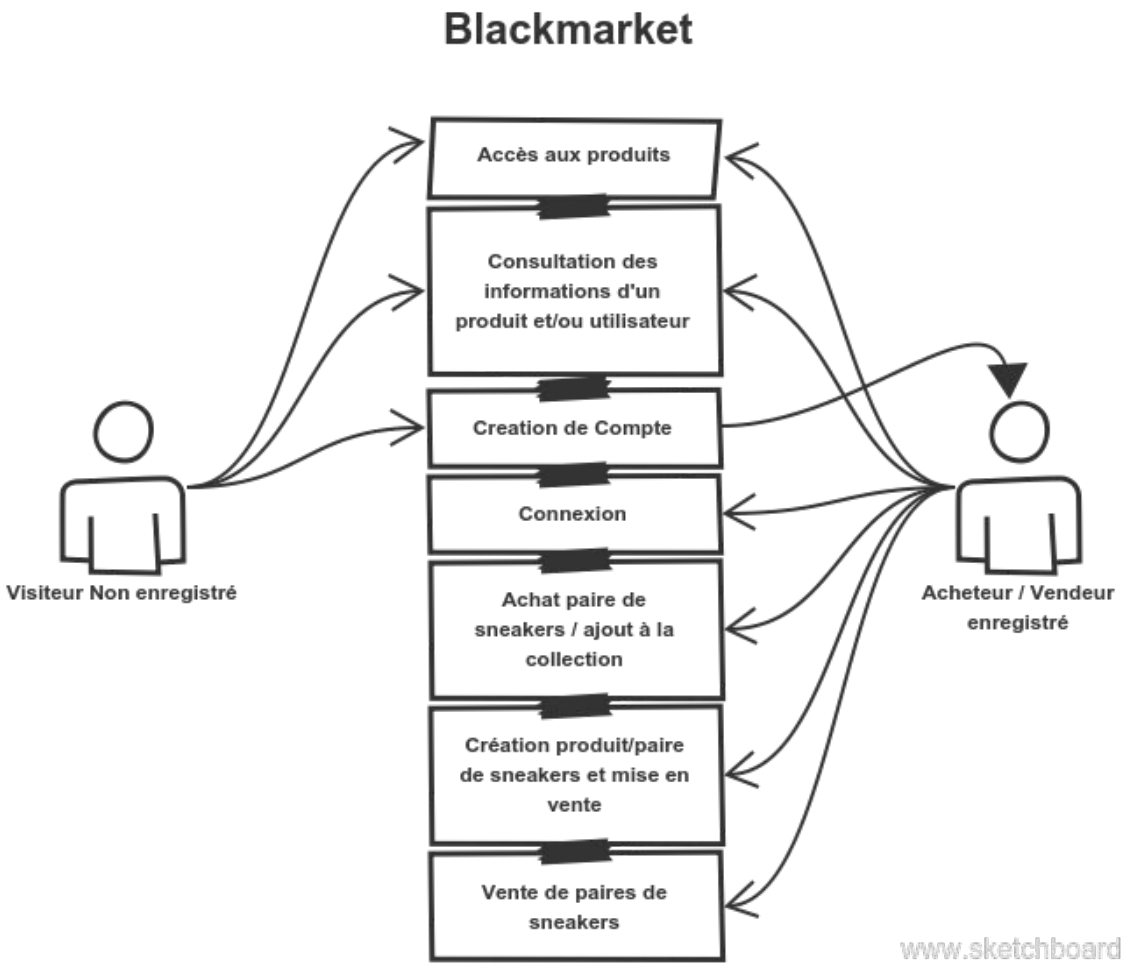
Conception Fonctionnelle et Technique

Specifications Fonctionnelles

Diagramme de Use Case

Afin de mieux comprendre le fonctionnement de l'application, voici un diagramme de cas d'utilisation (use-case) qui exprime les différents aspects auxquels doivent avoir accès les utilisateurs. Sur Blackmarket, un acheteur est aussi un vendeur potentiel et vice versa, Il n'y a donc qu'un seul model d'utilisateur qui aura accès aux mêmes fonctionnalités. La seule différenciation sera entre un visiteur non-enregistré et un titulaire d'un compte. Le premier ne pourra lui que visionner les produits et les informations des vendeurs, mais ne pourra pas interagir avec le site. Il ne pourra pas acheter, créer ou vendre de produit avant de devenir un utilisateur enregistré.

2. *Diagramme de cas d'utilisation (use-case)*



Structure fonctionnelle

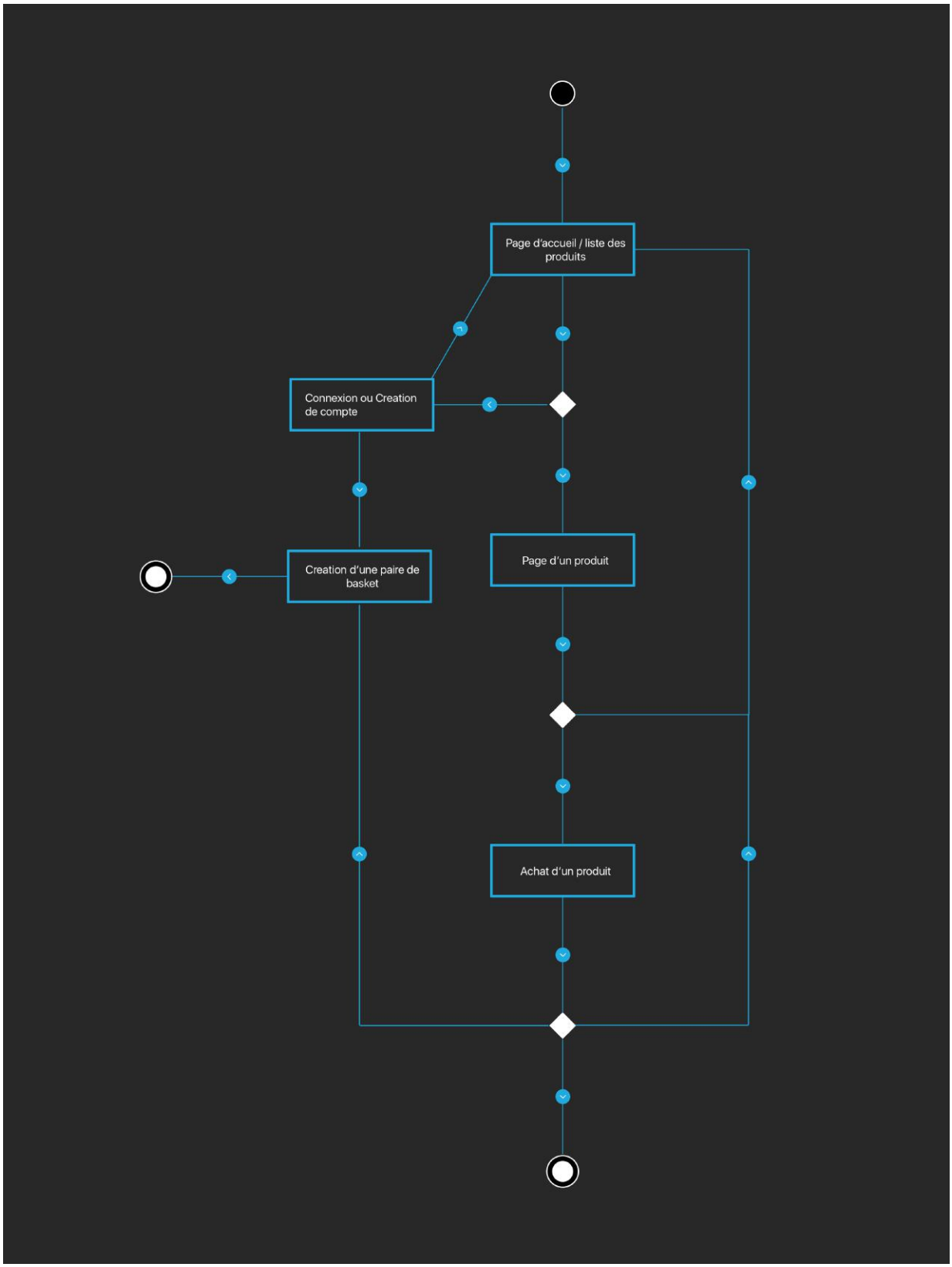
En fixant pour objectifs que l'application fournissent les fonctionnalités attendues et présentes dans le Diagramme de Use-case (page précédente), les pages suivantes ont été conçues :

- Page d'Accueil/Tous les produits - Affichant un message d'accueil ainsi que la liste des articles mis en vente.
- Page de Connexion - Permettant à l'utilisateur de se connecter ou d'être redirigé par un lien vers une page offrant la possibilité de créer un compte.
- Page d'inscription - L'utilisateur peut y trouver le formulaire pour s'inscrire et commencer à interagir avec le site.
- Page profile - Affichant les informations publiques d'un utilisateurs et qui peuvent être utiles aux autres visiteurs souhaitant en savoir plus à propose d'un article.
- Page modification profile - Formulaire reprenant les informations complètes d'un utilisateur et permettant leur modification.
- Page produit - Unique pour chaque produit, elle affiche les informations et les photos d'une paire de chaussures pour permettre son achat.
- Page de création/modification de produit - Formulaire permettant l'ajout d'un produit à mettre en vente ou reprenant les informations déjà présentes mais offrant la possibilité de les modifier.

Diagramme d’interaction

Ensuite, ce diagramme illustre le parcours type de l’utilisateur sur le site et les interactions qui lui seront présenté lors de sa visite sur la marketplace

3. *Diagramme d’interaction*

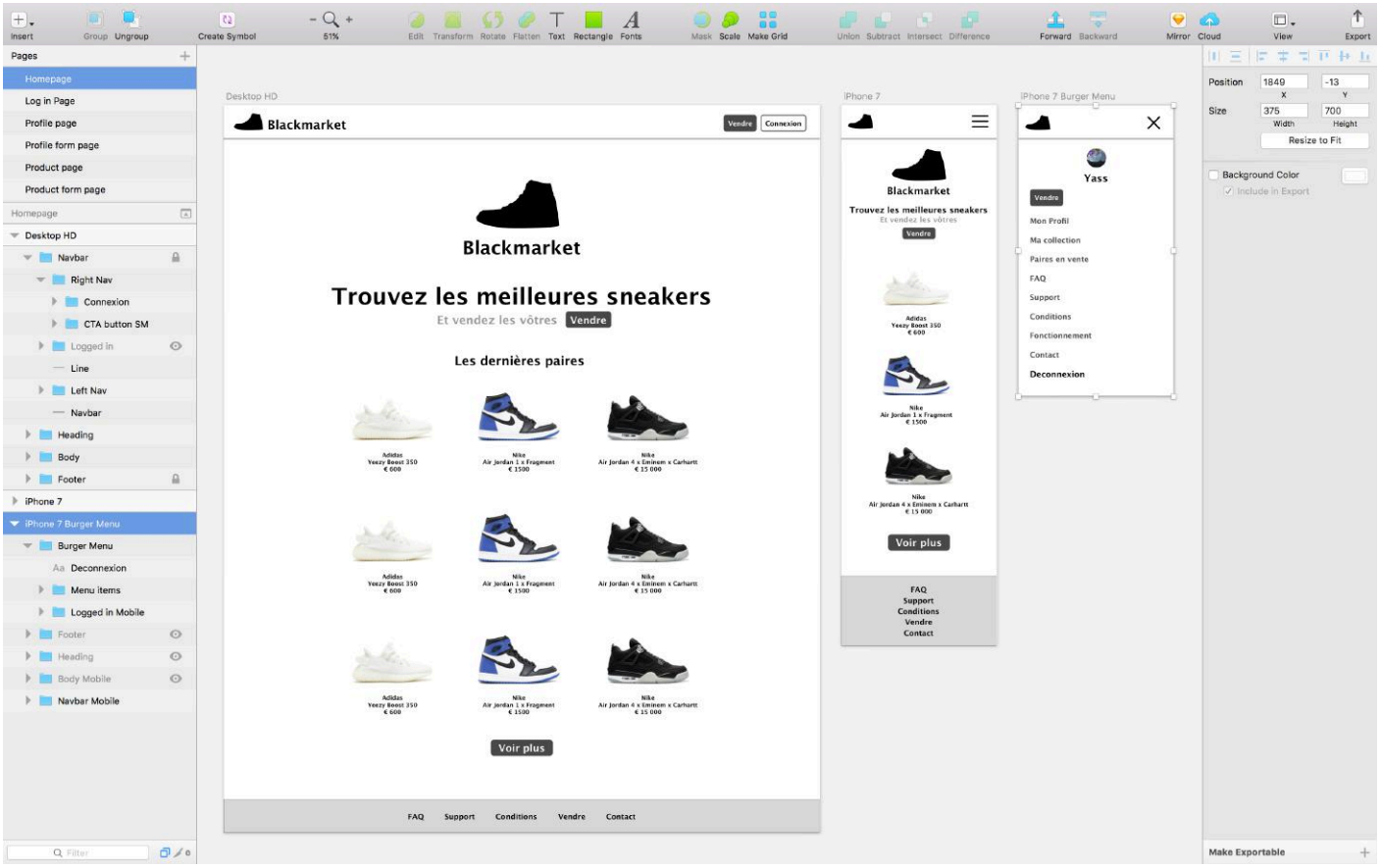


Maquettage de l'application

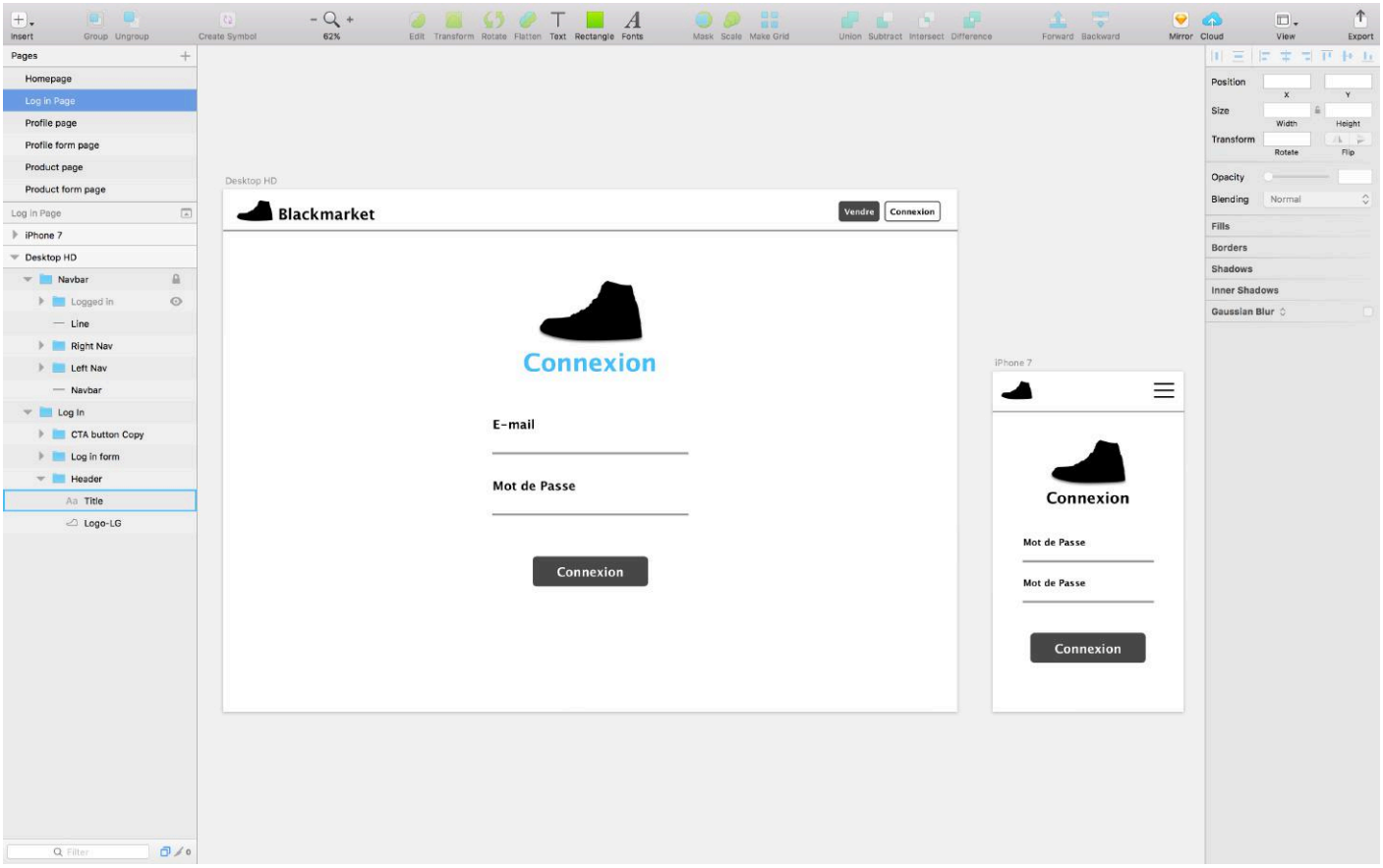
(Compétence 1: Maquetter une application)

Chaque page a ensuite été imaginée puis dessinée à l'aide de l'application de maquettage Sketch. Les pages suivantes montrent des captures d'écran permettant de visualiser les maquettes réelles de l'interface de l'application dans des déclinaisons pour les formats Desktop et mobile (le format tablette est adapté directement à partir de la version pour Ordinateur de bureau)

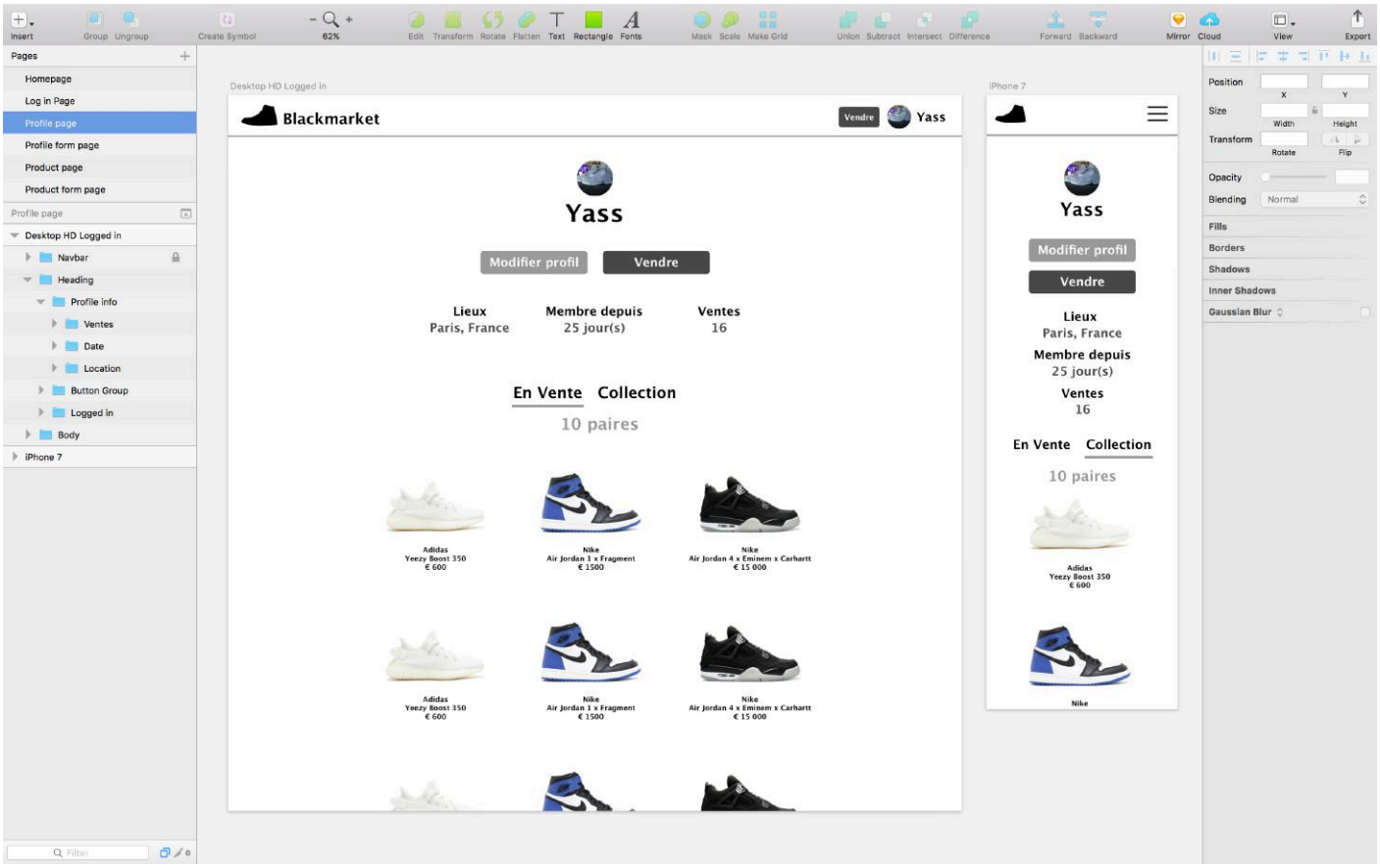
4. *Capture de la maquette Sketch de la page d'accueil/Tous les produits*



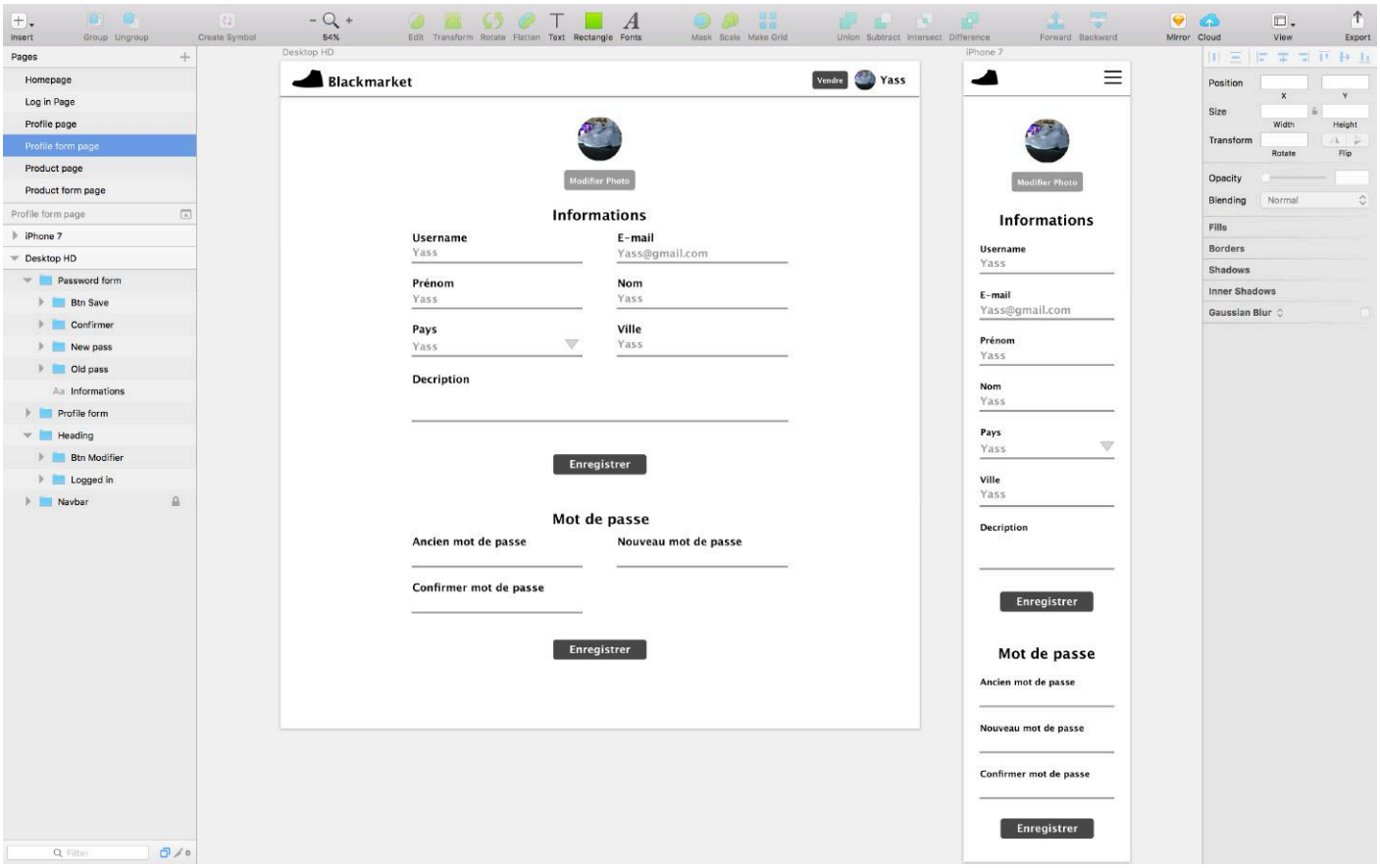
5. *Capture de la maquette de la page de connexion*



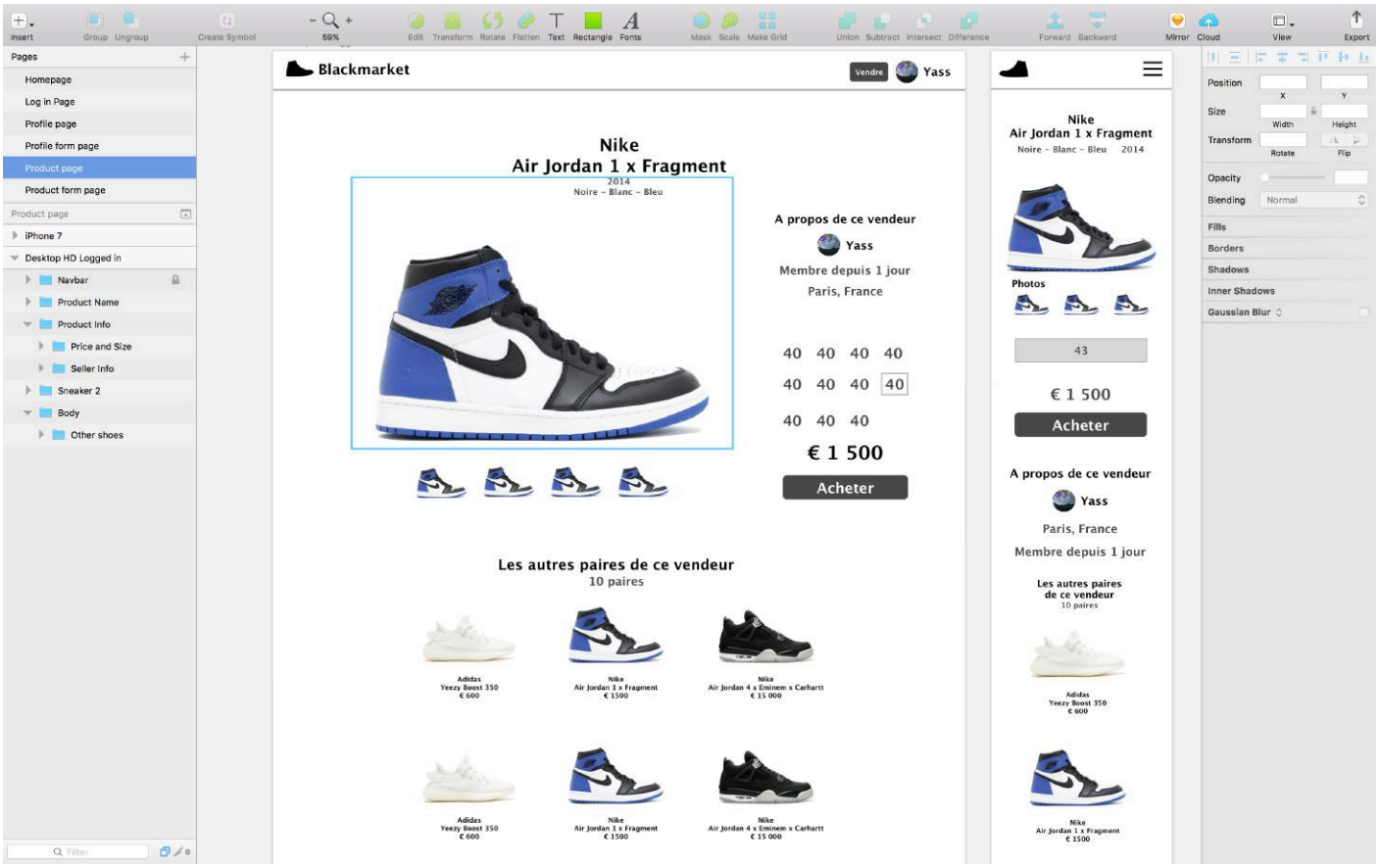
6. *Capture de la maquette de la page profil publique*



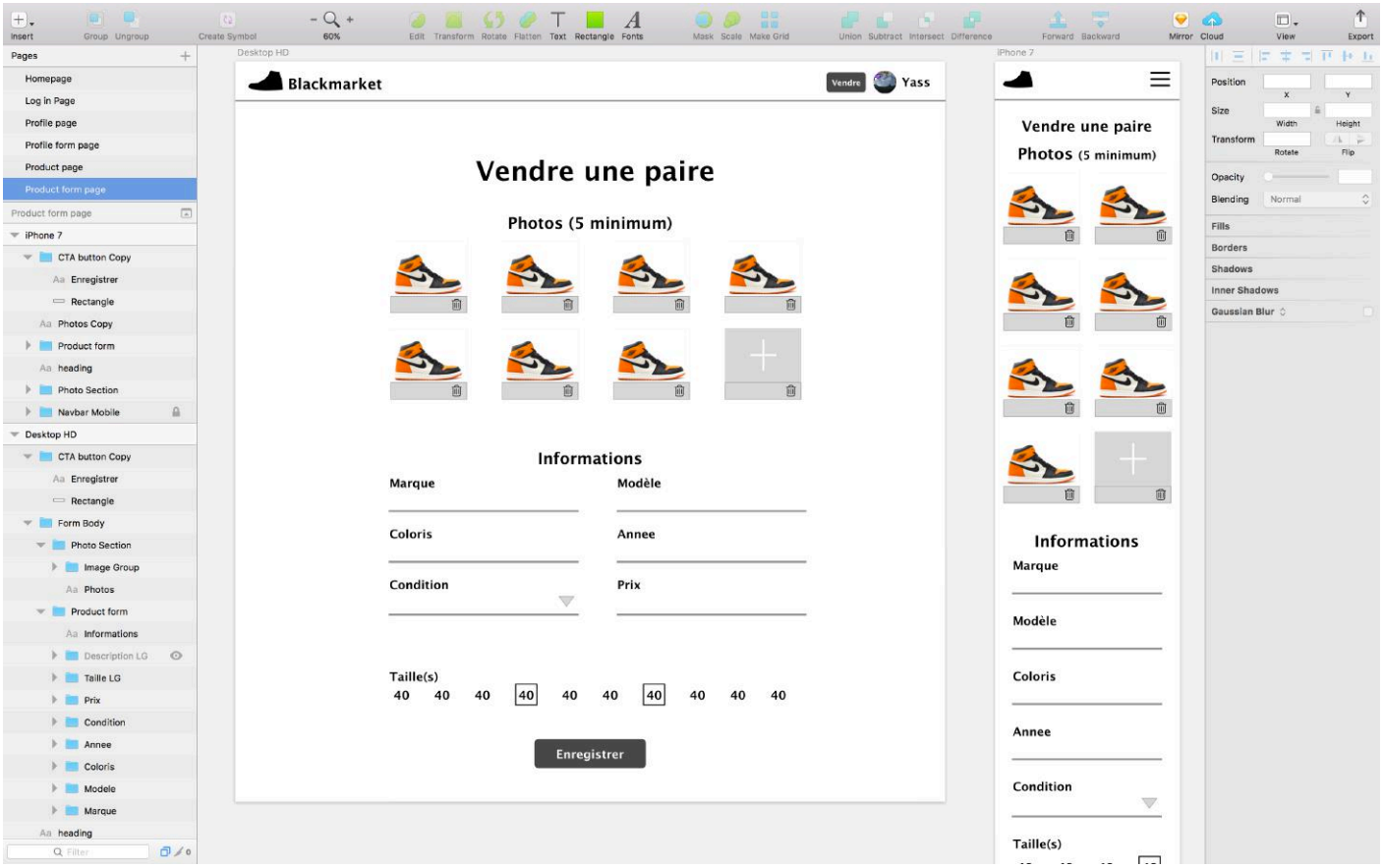
7. Capture de la maquette de la page du formulaire de modification du profil



8. Capture de la maquette de la page produit



9. Capture de la maquette de la page de creation/modification de produit



Specifications Techniques

Cette application web a principalement été réalisée dans le langage Javascript. Celui-ci a servi pour l'animation, les interactions avec l'utilisateur, l'affichage et le traitement des données provenant d'une base de donnée servie par une API REST, elle aussi en Javascript.

Ce langage a ensuite servi des composant en HTML, habillé par du CSS, puisant leurs données dans une base de données non-relationnelle MongoDB.

Choix techniques

Les choix techniques qui ont guidées la création de cette application web ont été motivés par un souci de cohérence à plusieurs niveaux.

Tout d'abord, il m'a semblé logique et plus simple d'utiliser le même langage pour la partie Front et la partie Back. En effet, ceci est aujourd'hui rendu faisable par la multitude de possibilités qu'offre le Javascript sur ces deux aspects.

Ensuite, l'écosystème Javascript a pris, ces dernières années, une ampleur majeure pour devenir aujourd'hui incontournable. On le doit notamment à une large communauté très active qui contribue de nombreux frameworks et librairies. C'est pourquoi, j'ai estimé qu'il serait d'autant plus pertinent de renforcer mes connaissances dans ce langage.

Par ailleurs, les outils 'back-end' disponibles présentent certains avantages concernant la performance et une certaine légèreté de code qui m'ont semblé intéressant pour la création d'un outil CRUD (Create Read Update Delete) dans mon application.

Enfin, réaliser cette application entièrement en Javascript me semblait cohérent compte tenu de mon bagage technique et de mon expérience. En effet, mon objectif personnel à travers ce projet était de consolider et approfondir mes connaissances dans ce langage tout en les élargissant avec de nouveaux outils et frameworks.

Front-End

Pour cette aspect de l'application, c'est le Javascript qui a servi à créer des composants générant des pages ainsi que des éléments en HTML et CSS dans le navigateur.

J'ai utilisé le framework VueJS pour générer ces composants après avoir été compilés grâce à l'outil Webpack.

Ce framework a été choisi pour sa légèreté et sa rapidité de prise en main puisqu'il présente des similarités avec AngularJS, avec lequel j'ai travaillé précédemment. Mais il propose surtout une approche 'component', plus contemporaine, qui permet de compartimenter des pages et/ou des éléments et des fonctionnalités. Ceci permet de modulariser ces derniers et les rendre plus facile à maintenir en les isolant dans des fichiers distincts, plutôt que d'avoir des larges portions de code au même endroit. Et ce, de manière beaucoup plus légère que pouvait le faire AngularJS (v 1.xx) et ses contrôleurs.

Ce framework proposait également deux approches. Une plus simple avec une utilisation libre de VueJS et un autre plus complexe générant un projet complet utilisant des outils de compilation et de génération de code plus avancées. Cette dernière permet de créer des Single File Components qui sont des fichiers comprenant a la fois les templates HTML, les scripts et le style propres au composant.

Dans un souci, comme mentionné plus tôt, d'élargir mes connaissances, c'est la deuxième approche que j'ai choisie. En effet, elle me permettait de découvrir une nouvelle façon de développer ainsi que la configuration et la compilation d'un projet avec l'outil avancé Webpack.

Dans ces Single File Component, le style a été écrit à l'aide du préprocesseur SASS, avec la syntaxe SCSS. J'ai choisi ce mode car il présente de nombreux avantages par rapport à du CSS classique. On peut le voir notamment dans l'utilisation de l'imbrication de classes, de variables (permettant de stocker des valeurs réutilisables comme en Javascript ou autre langage de programmation), de mixins (groupe de règles CSS réutilisables), de fonctions et bien d'autres encore.

Back-End

Le serveur 'back-end' a également été réalisé en Javascript à l'aide des frameworks NodeJs et ExpressJS.

En effet, NodeJS est un framework 'tout terrain' puisqu'il permet autant de constituer rapidement un serveur que de communiquer des informations à travers la console ou encore gérer des packages/plug-in, grâce à son outils NPM. Il est épaulé par la surcouche ExpressJS qui présente un nombre important de méthodes permettant rapidement de constituer des routes et des points d'accès pour une API.

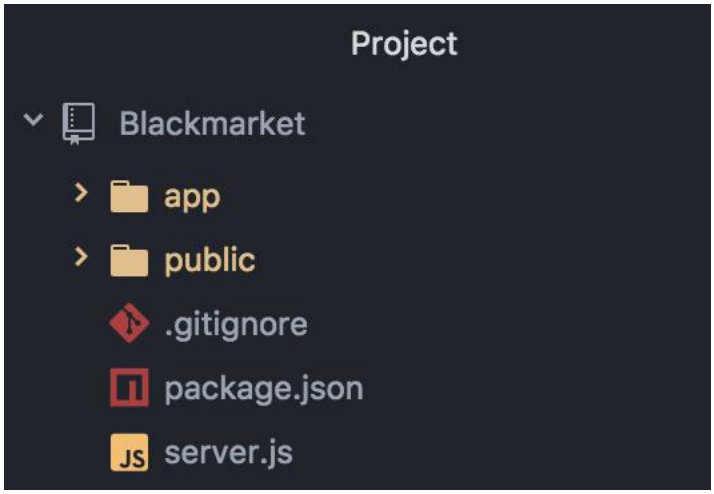
Enfin, la base de donnée a été créée sous un format non-relationnel grâce à MongoDB. Cette base a été modélisée à l'aide du plugin Javascript, Mongoose. Ce choix a également été guidé par souci de cohérence de langages mais aussi par sa rapidité de prise en main, flexibilité et performance.

Structure du projet

Architecture

Le projet a été structuré de manière bien distincte entre l'aspect Front de l'application et la partie Back. Cette partie va expliquer l'organisation des fichiers avant que leur contenu soit décrit plus tard dans la partie Realisation Technique .

Comme l'indique la capture d'écran ci-dessous, l'API a été organisée dans le dossier 'app', tandis que le dossier 'public' a hébergé les fichiers Front de l'application.



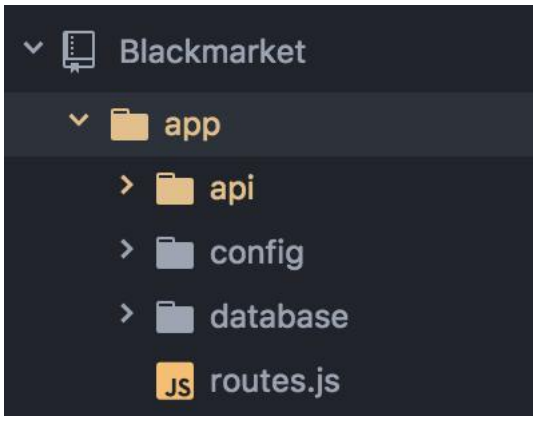
A la racine du dossier se trouve le fichier principale du serveur NodeJS aux côtés du gestionnaire des dépendances appelé 'package.json'. Ce dernier liste tous les

paquets, plug-ins ou librairies installés à l'aide de l'outil NPM (Node Package Manager) dont va avoir besoin le projet pour fonctionner. Ce fichier permet, en outre, de réinstaller tous ces paquets d'une traite si le projet est migré ou téléchargé sur une autre machine.

Le fichier '.gitignore' lui sert à lister les fichiers qu'il faut ignorer pour éviter de les inclure dans la sauvegarde du projet faite avec l'outil Git. Si le projet est installé sur une autre machine ou par un tiers, ces fichiers n'existeront pas. C'est notamment très utile pour exclure les modules NPM et ralentir le téléchargement du projet.

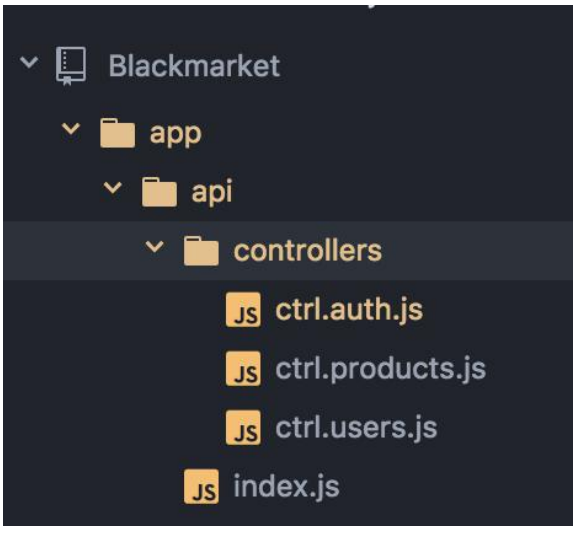
Back-end

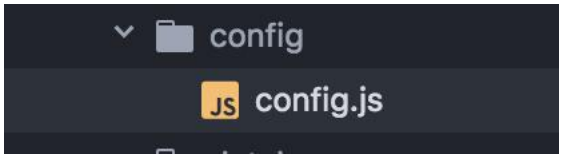
Le dossier 'app' contient les 3 facettes de l'architecture de l'API. En effet celle-ci a été compartimentée afin de la rendre plus modulable et facile à maintenir. Cela permet notamment de se rendre rapidement à l'endroit où le code peut poser problème. Mais cela permet aussi de le rendre plus lisible et compréhensible.



Premièrement, à la racine on va trouver un fichier nommé 'routes.js' qui va simplement indiquer au serveur les 2 types de routes: celle qui va servir l'interface, et celle qui fait référence aux routes spécifiques de l'API contenu dans le dossier 'api'.

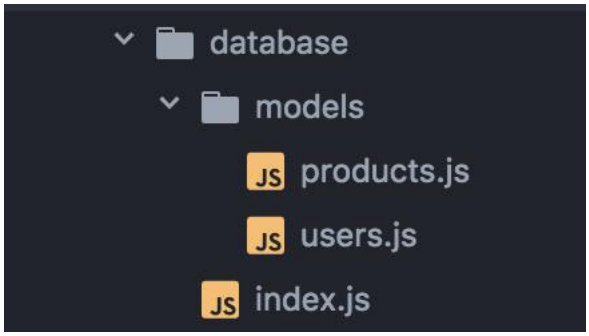
Ce dernier contient un fichier 'index.js dans lequel j'ai défini toutes les routes spécifiques à l'API et mettant en correspondance les requête HTTP avec les méthodes à exécuter. On les trouve dans le dossier 'controllers' qui les compartimente en fonction de leur utilisation. En l'occurence, il s'agit ici de l'authentification, les produits et les users.





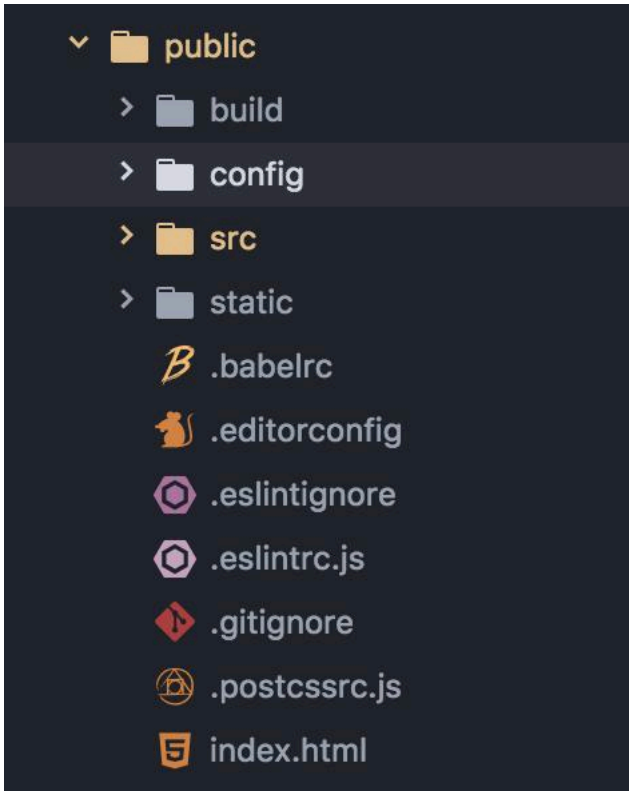
Le dossier 'config' sert lui à séparer des variables constantes et structurantes telles que l'url de connexion à la base de donnée, du reste du code.

Enfin, le dossier 'database' regroupe le fichier de configuration de la base de données et ses modèles. C'est ici que sont structurées les données pour leur stockage dans la base

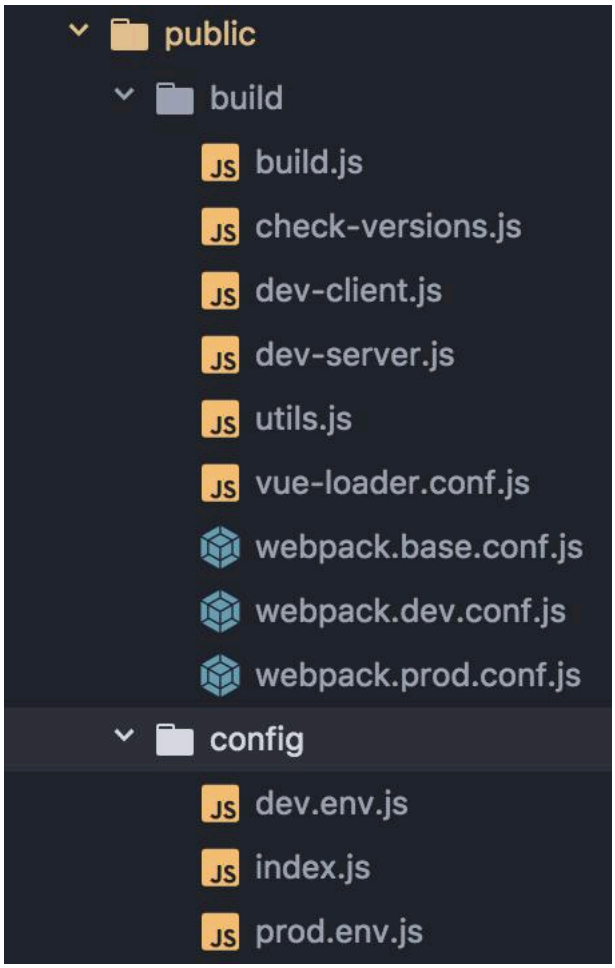


Front-end

Le dossier 'public' contient tout le code qui concerne l'interface utilisateur. Il est constitué d'un ensemble de fichier généré par le mode Vue-CLI (Command- Line-Interface). Le framework VueJS propose ce mode afin d'offrir tous les outils nécessaires au développement et à la compilation des fichiers au format '.vue' ou Single File Component qui contiennent au même endroit l'HTML, CSS et Javascript.



Comme le montre le contenu du dossier, une multitude d'outils est mis en place pour la compilation, transpilation, minification et remonté d'erreur. C'est le cas par exemple du fichier .'babelrc' qui configure l'outil 'Babel' pour traduire le javascript écrit sous la norme Ecmascript 6 (ES2015) ou 7 (2016) dans la norme Ecmascript5 (antérieur) encore utilisé par la majorité des navigateurs. Le fichier '.eslintrc.js', quant à lui configure le module de remonté d'erreurs de syntaxe ou de code.

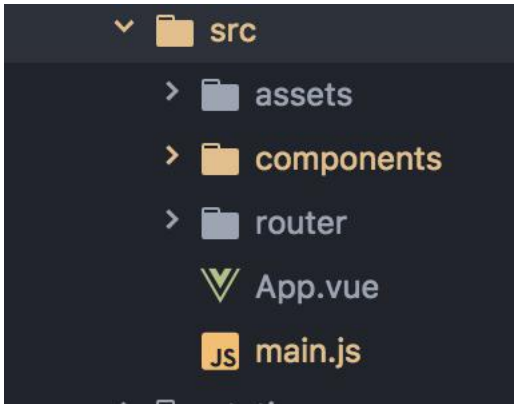


De la même manière que pour le Back-End, un dossier 'config' contient des variables fixes qui vont influencer la compilation selon le mode choisi 'production' ou 'developpement'.

Le dossier 'build', quant à lui, regroupe tous les fichiers utiles à la compilation qui est, elle, chapeautée par l'outil Webpack. Il est similaire dans son utilité à Gulp ou Grunt, car il exécute les tâches qui lui sont confiés dans le but d'optimiser le code du projet.

On notera que le projet Front-End contient lui aussi un serveur mais qui est seulement utilisé pour le développement et la compilation de l'interface.

Enfin, le dossier 'src' est celui dans lequel se situe tout le code qui va donner vie à l'application Front-End. Le dossier 'assets' contient les images et les feuilles de style globales (ici écrites en SCSS avant d'être compilées et minifiées par Webpack en CSS lisible par les navigateurs).



Le fichier 'main.js' va servir à l'importation des composants et dépendances du projet. 'App.vue' est un fichier au format propre au framework VueJS. Il se situe à la racine car c'est le composant principale de l'application. Il va définir l'application et faire le pont entre 'main.js' et les autres composants qui sont eux définis dans le dossier 'components'.

Le dernier dossier 'router' contient la configuration des 'routes' en Front-End. Celles-ci vont lier les composants/pages entre eux et les afficher lorsque l'on cliquera sur un lien dans l'application. Le fonctionnement du router 'front' est detaillé dans la Réalisation Technique.

Réalisation Technique

Comme expliqué précédemment, l'architecture du projet s'articule autour d'un socle 'back-end' en NodeJS, ExpressJS et une base de données. Et c'est le framework VueJS qui donne ensuite vie à la marketplace côté client. Nous allons voir plus en détails comment j'ai procédé pour la réalisation de ce projet.

Front-End

(Compétence 2 : Développer une interface et Compétence 7: Développer des pages web en lien avec une base de données.)

Afin que ce projet contribue à ma progression en tant que développeur, j'ai souhaité expérimenter le framework VueJS. Comme expliqué plus tôt, j'ai opté pour l'approche avancée qu'offre ce framework appelée VUE-CLI, afin de découvrir un nouveau mode récent d'organisation et de modularisation du code ainsi que les outils de compilation Webpack.

Ceux-ci, situés dans les dossiers 'build' et 'config' encapsulent déjà une configuration optimale pour convertir cette multitude d'éléments en une application complète. Cependant, ils ont nécessité quelques ajustements, principalement pour mutualiser l'importation de fichiers SCSS globaux à l'ensemble des composants. Autrement, cette approche demande une importation systématique du SCSS global là où ils sont utilisés, chose que je souhaitais éviter.

Point de départ

Mais dans l'ensemble la majeur partie du travail a eu lieu dans le dossier 'src', à la racine duquel on trouve le fichier 'main.js'. Dans ce document se situe l'importation des dépendances globale du framework VueJS comme on peut le voir dans cet extrait:

```
import Vue from 'vue'
import App from './App'
import {MediaQueries} from 'vue-media-queries'
import router from './router'
```

On peut voir à la première ligne l'importation du code source 'Vue' en vue de son utilisation dans l'application. On note, ensuite, une importation d'un élément appelé 'App' qui est le composant principale et qui va faire le pont entre le framework et le reste de l'application.

Puis, il est suivi de la création d'une nouvelle instance du framework Vue, lancée par le mot clé 'new' pour le lancement d'une application. Celle-ci est accompagnée d'un objet d'options. On peut voir une propriété 'el' qui va spécifier l'élément HTML à prendre en compte comme point d'entrée (on indique l'id de l'élément). 'router' quant à lui fait appel à un fichier externe qui va spécifier des éventuelles routes utiles à l'affichage rapide des pages. La propriété 'template' spécifie la structure HTML d'où va partir l'application, tandis que 'component' spécifie le nom du composant principal.

```
new Vue({

 el: '#app',
 router,
 template: '<App/>',
 components: { App },
 mediaQueries: mediaQueries

})
```

L' Application

Comme mentionné plus haut, c'est le composant 'App' qui constitue le point d'entrée de l'application. Il s'agit d'un composant comme un autre contenu dans un fichier au format '.vue'. Mais il est le socle de l'app car il contient l'élément HTML portant l'id que l'on a désigné plus tôt comme porteur du code et de la logique de cette app.

Son rôle est donc d'enclencher les autres composants externes que l'on peut créer par la suite. J'élaborerais plus en détails le fonctionnement des composants dans la section suivante. En tant que composant principal, du point de vue de l'interface, il va contenir des éléments fixes. J'y ai donc inclus la barre de navigation et le footer dont on a besoin sur chaque écran.

Étant donné que j'ai opté pour l'utilisation d'un routeur pour afficher les différentes pages plus rapidement, le composant 'App' contient une balise personnalisée:

```
<router-view></router-view>
```

Cette balise signale l'endroit où vont être affichés les différentes vues (views) du site. Il s'agit là d'une pratique commune aux SPAs (Single Page Apps). Le principe de ce type d'application est d'éviter de faire une requête au serveur distant pour afficher chaque page. Contrairement à l'approche 'server-side' (côté serveur), toutes

les pages vont être divisées entre le contenu commun (navbar et footer par exemple) et leur contenu propre. Elles seront ensuite préchargées en une seule fois au lancement de l'application. Cette pratique offre un rendu beaucoup plus rapide car tout se passe du côté du client.

Les composants

Comme mentionné plus haut, le développement à l'aide de VueJS s'articule autour de composants. Ils peuvent représenter soit le contenu entier soit des éléments plus simples d'une page. Dans le deuxième cas, il faut les inclure à l'intérieur d'un composant 'page'.

Ils sont écrits dans des fichiers au format '.vue'. Celui-ci est propre à l'approche, optionnelle, Single File Component de VueJS car il permet d'encapsuler, à la fois, le HTML, le Javascript et le style. Ces fichiers se présentent ainsi:

```
<template>
  <div class="maClass">
  </div>
</template>
<script>
export default {
}
</script>
<style lang='scss'>
</style>
```

La balise 'template' contient la structure HTML. Il faut noter que l'architecture 'composant' ne permet de rendre qu'un seul élément. Il faut donc englober tout les constituants de la page dans un élément parent pour que le rendu ait lieu. Ensuite, la balise 'script' contient toute la logique Javascript propre au composant. Enfin, la balise 'style' contient l'habillage du composant en CSS ou autre langage compatible tel que SCSS/SASS ou LESS (qui doivent être spécifiés dans l'attribut 'lang').

Si on s'intéresse à la partie Javascript, la logique de composants et de compilation Webpack nécessite un 'export' (qui rend le code exportable) de son contenu pour être ensuite importée ou injecté dans le Javascript compilé de l'application.

```
export default {

  data () {

    return {
      maVariable: maValeur
    }
  }
}
```

L'export va alors avoir lieu sur un objet qui va contenir divers éléments de la logique nécessaire au fonctionnement du composant. Premièrement, les variables vont se loger dans la fonction 'data ( ) '. Ensuite, c'est la propriété 'methods' qui va regrouper les méthodes et fonctions que l'on va créer pour animer le composant. Celles-ci sont alors disponibles pour être appelées par des événements grâce, entre autres, à la directive HTML 'v-on: ' ('click', 'mouseover', etc), similaire à ng-click dans AngularJS:

```
export default {

  data () {

    return {

    }
  },
  methods:{
    maMethode: ()=> {

    }
  }
}
```

Par ailleurs, j'ai pu aller plus loin et explorer d'autres propriétés de cet objet. Par exemple, j'ai décidé de compartimenter le composant de la page d'accueil. En effet, il sert aussi à lister tous les produits disponible et il est donc amené à être plus volumineux. Cela m'a permis d'utiliser la propriété 'components' qui permet de lister les sous-composants à inclure, après avoir importé leur fichier correspondant bien sûr, comme on peut le voir ici:

```
import HomeHeading from ‘./homeHeading.vue'
import HomeProducts from './homeProducts.vue'
export default {
  name: 'Home',
  data () {
    return {
    }
  },
  components: {
    HomeHeading,
    HomeProducts
  }
}
```

Cette compartimentation se matérialise ensuite par des balises personnalisées que j’ai inclus dans le template du composant page et qui signale l’endroit où les composants doivent être injectés.

```
<template>

  <div class="home view-container">
    <home-heading></home-heading>
    <home-products></home-products>
  </div>

</template>
```

Enfin, pour afficher les produits au chargement de la page d’accueil, j’ai fait appel à la fonction ‘mounted’. Elle sert à exécuter des fonctions au moment du chargement du composant pour permettre un rendu plus rapide. Cette fonctionnalité s’est montrée très pratique pour charger des données depuis le serveur et la BDD par le biais de l’API rapidement et afficher les produits disponibles dès le lancement de la page:

```
mounted () {
   loadProducts () {
      return axios.get('http://localhost:5000/api/products')
    }
  }
```

Développement responsive

(Compétence 3: Développer une application simple de mobilité numérique)

L'application a été conçue pour s'adapter aux différents écrans sur lesquels le site est susceptible d'être utilisé. J'ai donc pris en compte non seulement les tailles d'écrans de mobile et tablette mais également d'ordinateur et les différentes résolutions qu'ils peuvent avoir.

Afin de rendre cet aspect plus simple à développer, j'ai mutualisé l'interface des écrans Desktop et tablette en la rendant fluide. Par exemple, la taille de certains éléments, comme des polices de texte ou des images, est flexible et peut s'ajuster. J'ai utilisé des unités de mesure fluides, comme le 'rem' ou le 'vh', plutôt que fixes comme le pixel ('px'). Le 'rem' signifie 'relative em' (l' 'em' est aussi une mesure fixe) tandis que le 'vh' (viewport height) correspond à la hauteur du support, et son équivalent, le 'vw' (view width), à la largeur. Ici on peut voir la taille du logo qui est réactive au format d'écran:

```
.home-logo{
  width: 20vh;
  margin-bottom: 10px
}
```

On notera que certaines tailles qui n'ont pas besoin d'être adaptées sont fixées en 'px'. C'est le cas de la marge sous le logo.

La version pour Mobile, quant à elle, est différente, notamment avec le retrait de certains éléments ou la modification de leur taille pour éviter d'altérer l'expérience.

Grâce au principe des mediaqueries, j'ai spécifié des règles de CSS à appliquer lorsque le support correspond à certains points de ruptures ('breakpoints'). Il peut s'agir d'une largeur et/ou hauteur d'écran précise, une orientation (paysage ou portrait) ou encore si le support est inférieur ou supérieur à une certaine taille. Ici on peut observer comment appliquer des mediaqueries en (S)CSS:

```
@media screen and (max-height: 500px) and (orientation : landscape){

    .home-logo-container{
      margin-bottom: 20px
    }

    .home-logo{
      width: 25vh;
    }
    .home-logo-branding{
      font-size: 5vh;
    }
  }
```

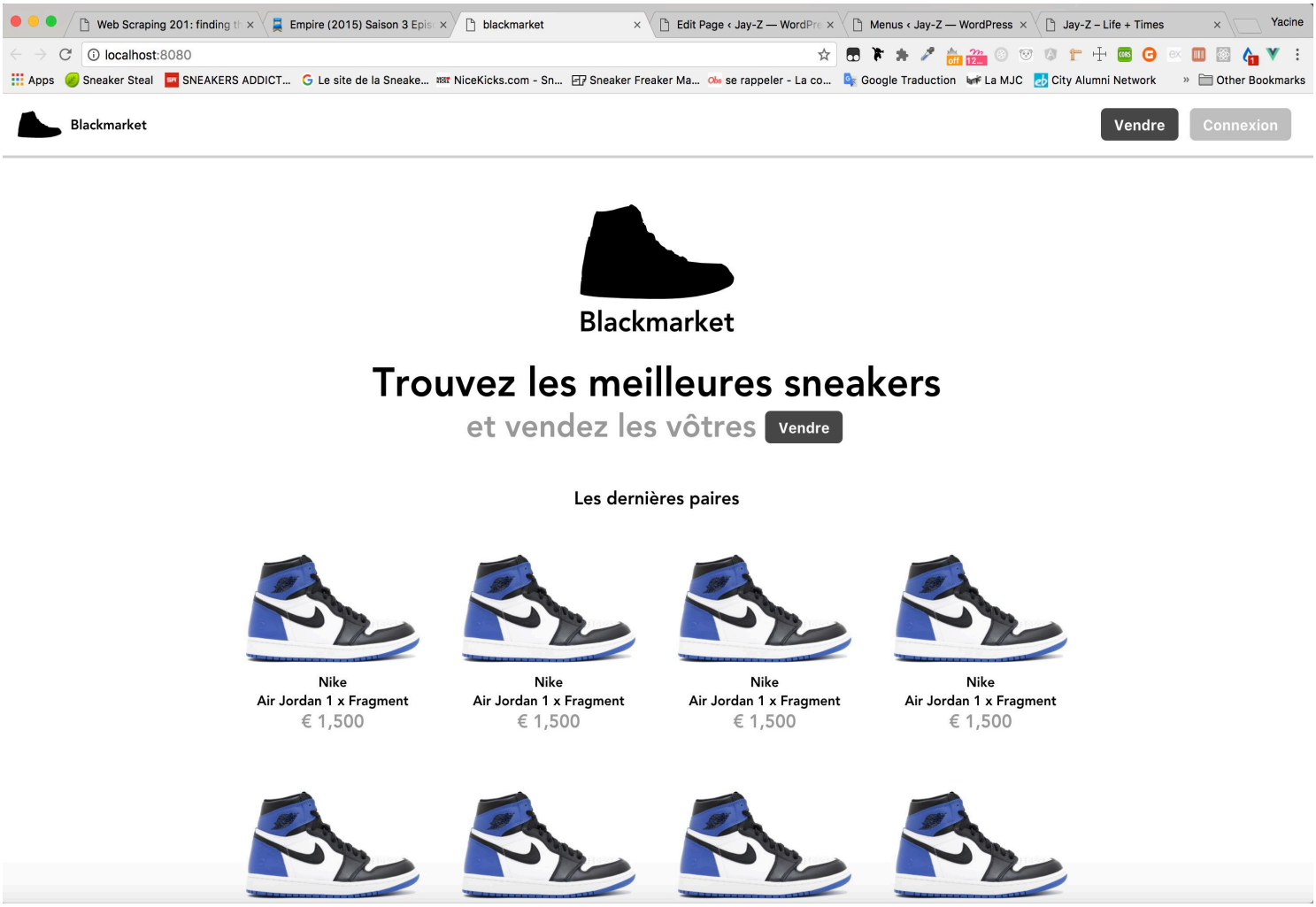
On peut observer ici des règles qui doivent s'appliquer à condition que l'écran ait une hauteur maximum de 500px et une orientation en paysage. Ce sont les supports Mobiles qui sont visés et le résultat est visible dans les captures plus bas.

Enfin, pour éviter d'écrire certaines règles, j'ai fait appel à un plug-in du framework VueJS appelé 'vue-media-queries'. Cet outil utilise la directive 'v-if' de VueJS, directement dans le HTML, car elle permet d'agir sur la présence d'un élément si une condition (if-else) est vraie. Ici, j'ai précisé que l'élément ne devais apparaitre que si l'écran a une largeur de 768px ou plus.

```
<h1 v-if="$resize && $mq.above(768)" class="nav-branding black">Blackmarket</h1>
```

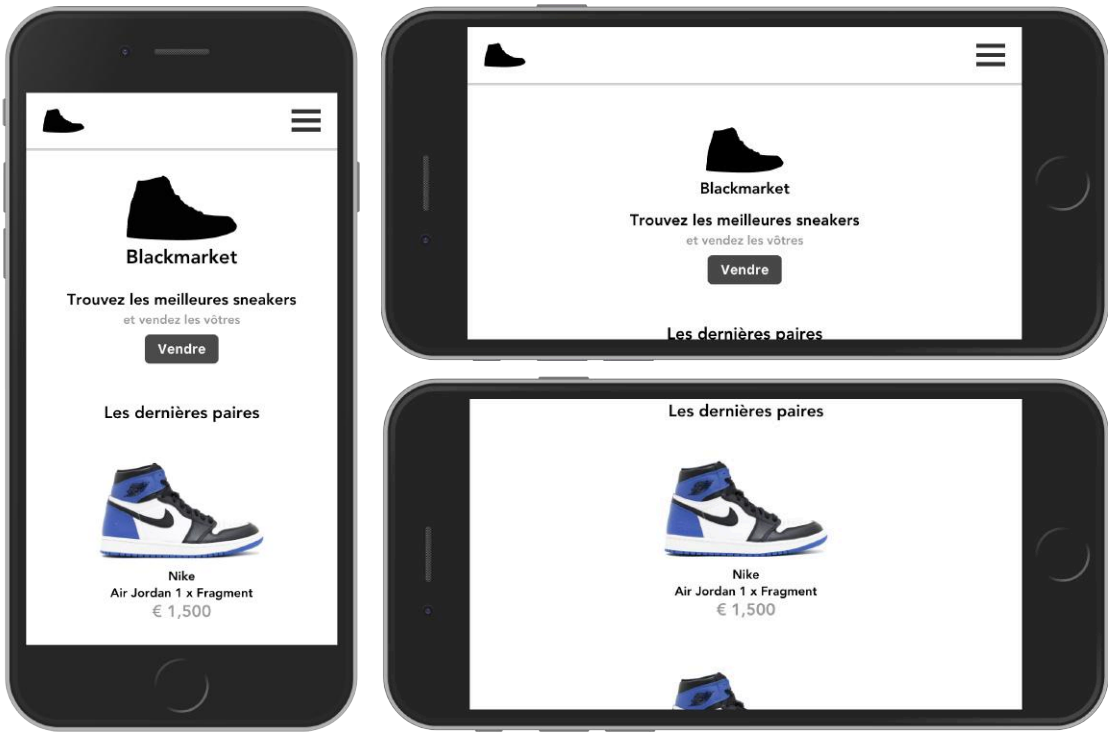
On peut voir, dans les captures qui suivent, l'effet de ces outils en comparant les interfaces desktop, tablette et mobile:

1. *Version Desktop*



2. *Version tablette (iPad)*

3. *Version Mobile (iPhone) en mode Portrait et Paysage*



Back-End

L'application présentée pour ce Titre Professionnel est dite 'full-stack' et intègre non seulement une interface utilisateur mais aussi une partie 'back-end' avec une base de données et des composants permettant d'y accéder.

Conception de la base de données

(Compétence 4 :Concevoir une base de données)

Ainsi, la réalisation de ce projet a nécessité la conception d'une telle base. Le point de départ de la réflexion autour de sa modélisation a été de nommer les différents aspects qui nécessiteraient des données. Le projet étant une marketplace, j'ai identifié deux parties concernées par l'ajout, lecture, modification et suppression de data: les utilisateurs et les produits.

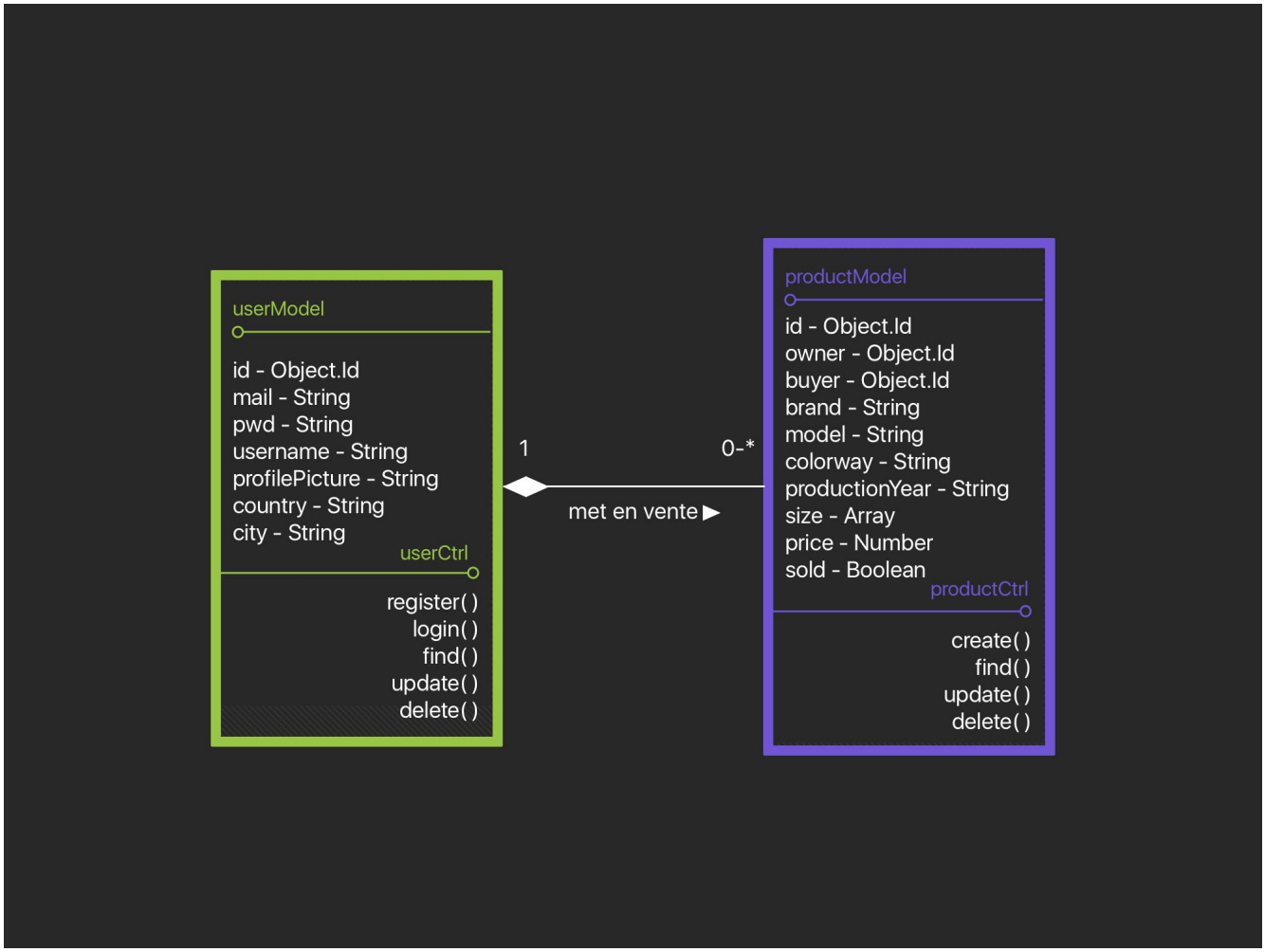
En effet, les users vont avoir besoin de créer un compte pour s'identifier puis, éventuellement, le modifier. Ils vont également avoir besoin de lire des informations concernant d'autres utilisateurs.

De plus, la fonctionnalité première d'une marketplace étant d'acheter, de créer ou de modifier des produits, il était indispensable de créer un modèle pour ces derniers permettant d'accéder à ces fonctions.

Il y a cependant une certaine hiérarchie dans cet ensemble de modèles puisque ce sont les users qui vont alimenter la base de données en ajoutant les produits. Au delà de leur profil qu'ils rempliront d'informations personnelles, ils vont créer des produits pour les mettre en vente. Ils vont également les modifier s'ils ont fait une erreur ou s'ils souhaitent les modifier, que ce soit pour baisser le prix ou ajouter de meilleures photos par exemple.

Pour modéliser cette data, je me suis inspiré des diagrammes UML afin de mieux conceptualiser cette structuration des données. J'ai créé un diagramme de classe pour définir les diverses propriétés et méthodes que je pouvais rattacher à chaque modèle et les interactions qui pourraient avoir lieux entre ceux-ci. On peut voir sur la page suivante ce diagramme pour mieux visualiser ces modèles et leur contenu.

4. Diagramme de classe illustrant les modèles



Dans ce diagramme, on distingue le fonctionnement de la relation entre ces deux modèles. Il y a une relation de composition entre le modèle User et celui de Product. En effet, ce dernier est entièrement dépendant du premier puisqu'il n'existe que suite à sa création par un User.

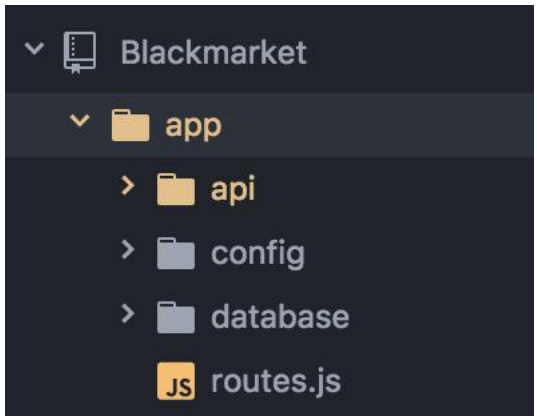
De plus, le userModel a une multiplicité qui s'étend de Zéro à l'infini envers le productModel. La multiplicité provient du fait qu'il peut avoir soit aucune soit un nombre infini d'interaction avec le productModel. En effet, il peut lire, créer, modifier ou supprimer autant de produit qu'il le souhaite, sans limites mais n'est pas obligé d'en mettre en vente. En revanche, le modèle produit a une seule et unique relation avec le modèle utilisateur car il est unique et n'a qu'un seul instigateur.

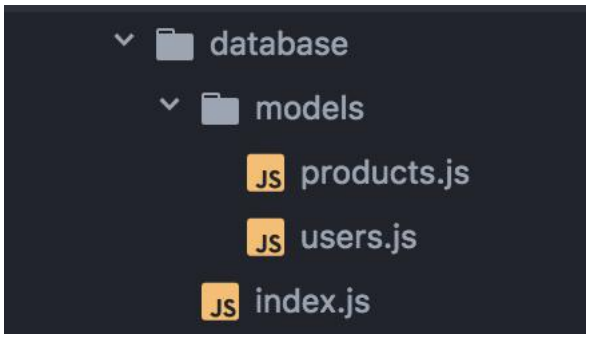
Mise en place

(Compétence 5: Mettre en place une base de données)

Pour la mise en place d'une base de données, j'ai opté pour le type non-relationel offert par MongoDB. Son installation est assez simple et se fait grâce au gestionnaire de paquet Homebrew. J'ai ensuite fait appel au package Mongoose (obtenu via NPM) afin de gérer la connexion à cette base de données et modéliser mes Schémas de data.

Si l'on revient à l'architecture du projet, tous les fichiers 'back-end' se situent dans le dossier 'app'. C'est donc à cet endroit que l'on va trouver le dossier 'database' où j'ai donné naissance à cette base de données.





C'est donc ici que j'ai séparé le code 'général' de la creation de la base de donnée, de celui de ses modèles. Cette approche modulaire permet de maintenir plus facilement les modifications que l'on peut apporter à la base. Ainsi, chaque modèle a son propre fichier répertorié dans le dossier 'models' et est ensuite appelé dans l'index.js à la racine de 'database'.

Dans le fichier principal, on commence par importer le package Mongoose afin de pouvoir l'utiliser, en l'assignant à une variable constante ('const' est un mot clé provenant de la norme ES6).

```
const mongoose = require('mongoose');
mongoose.connect(global.config.database);
```

Ensuite, la première chose à faire est de renseigner l'URL de connexion à la base de donnée mongoDB et l'appliquer dans la méthode '`.connect()`' de Mongoose. En l'occurence, l'URL est contenue dans un objet de configuration contenu dans un fichier externe qui se situe dans le dossier 'config' (voir l'architecture). Il contient des variables de configuration générale du projet qui sont amenées à changer selon

l'environnement, si celui-ci est 'développement' ou 'production' par exemple. Cela va permettre de modifier plus facilement l'URL de connexion de la base de données, notamment lorsqu'elle sera migrée vers un hébergement externe.

Dans un deuxième temps, j'ai appelé les fichiers contenant les modèles:

```
const Users = require('./models/users.js');
const Products = require('./models/products.js');
```

Cela m'a permis, enfin, de définir la base de données en spécifiant ses éléments constituants, appelés collections, définies par les modèles que j'ai établis, à insérer dans la base de données.:

```
const db = {

  users: mongoose.model('Users', Users),
  products: mongoose.model('Products', Products)

}
```

Parallèlement à la création de la base, je me suis aidé du diagramme qui a servi à la conception des modèles pour les concrétiser (voir diagramme des modèles). Dans le dossier 'models', on trouve un fichier pour chacun d'eux et qui se compose ainsi:

Dans un premier temps, je doit faire appel au composant 'Schema' du package Mongoose:

```
const {Schema} = require('mongoose');
```

Celui-ci va permettre de créer une nouvelle instance de 'Schema' où j'ai donc pu définir ces modèles:

```
const monModel = new Schema({

   {...}

});
```

Et c'est dans cette nouvelle instance que j'ai constitué toutes les propriétés du modèle, telles que je les ai établies dans le diagramme :

```
mail:{

  type: String,
  lowercase:true,
  unique: true,
  required:true

},
hash:{

  type: String,
  required:true

},
username:{

  type:String,
  unique:true,
  required:true

},

{...}
```

Chaque propriété doit spécifier le type de donnée qui est attendue. mais, j'ai voulu spécifier un objet d'options facultatives afin que ces données soient enregistrées de manière plus stricte. Parmi ces options, on peut indiquer si la donnée est obligatoire (`required: true/false`) ou si elle doit être unique. C'est à dire si une donnée égale peut exister dans un autre compte utilisateur par exemple, ou si un produit peut porter le même nom.

Ainsi, dans le modèle Users on trouvera des propriétés qui vont permettre de stocker les informations utiles à sa connexion telles que son ID, son adresse e-mail (propriété '`mail`'), son mot de passe ('`hash`') et qui devront donc être uniques. Ensuite, on y stockera des informations utiles aux autres users, et qui ne devront pas forcement l'être, telles que son nom d'utilisateurs, sa ville, son pays (afin de savoir où si situe un produit) et l'url vers sa photo de profil.

Le modèle Produit, quant à lui, va contenir la marque, le modèle, l'année de production, le coloris, la ou les tailles, ainsi que le prix de la paire de chaussure.

Au passage, on notera la présence, au sein de nombreux fichier du 'Back-end', d'un export:

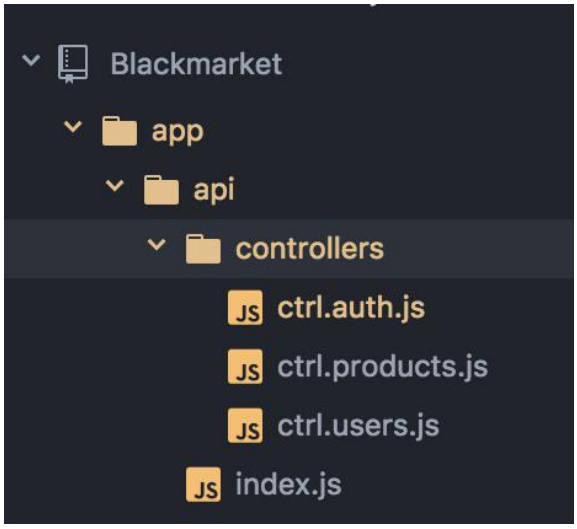
```
module.exports = maVariable;
```

Il sert à exporter leur contenu afin qu'ils puissent ensuite être importés ailleurs. C'est notamment le cas, ici, des modèles qui ont été importés dans le fichier principal pour constituer la base de données.

Accès aux données

(Compétence 6: Développer des composants d'accès aux données)

Une fois la base de données mise en place, j'ai créé des modules permettant d'y accéder afin de traiter et effectuer des actions sur ces informations. Pour cela, j'ai mis en place des contrôleurs qui vont contenir les fonctions à exécuter sur la base de données selon la requête qui sera faite au serveur.



Si on revient encore une fois à l'architecture du projet, on trouvera le dossier 'api'. A sa racine, est situé le fichier qui spécifie les routes de cette API. J'ai ensuite créé un dossier 'controllers' dans lequel sont placés les contrôleurs de chaque modèle.

Contrôleurs

A l'intérieur de ceux-ci, j'ai défini un objet auquel j'ai assigné des méthodes, après avoir fait un appel à la collection concernée. Si on regarde le contrôleur Users, on peut observer cet objet et l'importation de la collection du modèle qui le précède :

```
const Users = require('../../database').users;

const userController = {
  find: (req,res) => {
    Users.find()
    .exec()
    .then (data =>{
      res.send(data);
    })
    .catch(err =>{
      res.send(err);
    })
```

Ici, la propriété ‘find:’ sert à récupérer les users. Elle prend pour argument la requête (‘req’) et sa réponse (‘res’) avant d’appliquer une méthode propre à Mongoose appelée également ‘find()’ sur la base de données des Users.

Sur cette requête, j’ai ensuite recours à des promesses (‘promise’) pour indiquer ce qui doit avoir lieu après réception de la réponse, par le biais de fonctions ‘callback’. Dans ce cas précis, je demande à ce que les données (‘data’) reçues soient envoyées à l’application afin de les traiter ensuite en ‘front’. La promesse ‘catch()’ permet, elle, d’exécuter une fonction callback en cas d’erreur. Ici, j’ai fais en sorte que cette erreur soit renvoyée afin de pouvoir l’exploiter.

Pour enregistrer une nouvelle donnée, j’ai encore une fois fait appel à une méthode propre à Mongoose appelée ‘save()’ et qui va être utile pour la création d’un produit:

```
create: (req,res) => {
    Users
      .find({ mail: req._userMail })
      .then(user => {
        const userId = user[0]._id;
        req.body._owner = userId;
        const newProduct = new Products(req.body);
        newProduct.save()
        .then(data =>{
          res.send('product successfully created' + data);
        })
        .catch(err =>{
          res.send('err' + err + 'in creating product');
        })
```

Ici, j’ai créé la propriété ‘create: ’ dans l’objet ‘productsController’ qui commence par une recherche du user par adresse e-mail grâce à la méthode Mongoose ‘find()’, mentionnée plus tôt. Une fois l’utilisateur trouvé, on stocke son identifiant ‘_id’ dans une variable constante dans l’optique d’associer ce produit au user qui veut le créer. Cet identifiant va alors être utilisé pour définir la valeur de la donnée ‘_owner’ du modèle Product. Elle fait référence au propriétaire de ce produit et va concrétiser cette association avec l’utilisateur dans la base.

La variable constante ‘newProduct’, quant à elle, appelle une nouvelle instance du modèle Product (visible par le mot clé ‘new’). Celui-ci va prendre en argument le corps de la requête (‘req.body’) dont le contenu va provenir d’un formulaire dans l’interface utilisateur. Et c’est à ce moment là qu’intervient la méthode Mongoose ‘save()’ pour enregistrer le document dans la collection des produits. Enfin, s’en suivent les promesses comme vu précédemment, qui sont présentes sur chacune

des méthodes de l'objet du contrôleur. Elles vont renvoyer la data ou bien l'erreur selon le cas qui se produira.

Par ailleurs, cette application contient un module d'authentification pour lequel j'ai également développé un contrôleur. Son intérêt est de me permettre de crypter le mot de passe de l'utilisateur et de gérer les tokens pour indiquer à l'application que le user est bien connecté avec un compte. C'est pourquoi j'ai utilisé les packages Bcrypt et JSON Web Token:

```
const jwt = require('jsonwebtoken');
const bcrypt = require('bcrypt');
```

Etant donné que l'authentification concerne les données des utilisateurs, j'ai importé la collections des users:

```
const User = require('../../database').users;
```

Puis, j'ai défini des fonctions qui utilisent ces plug-ins afin d'en tirer profit dans l'assignation de méthodes à l'objet contrôleur. J'ai écrit une fonction pour générer un token:

```
const generate_token = (user) => {
  const payload = {
    exp: moment().add(14,'days').unix(),
    iat: moment().unix(),
    iss: user.mail,
    sub: user.hash
  }
  return jwt.sign(payload, global.config.APP_SECRET);

};
```

Ici, je défini un objet appelé 'payload' qui contient des informations relatives à l'utilisateur telles que la date de creation et d'expiration du token, son e-mail et son mot de passe. Ensuite, la méthode 'sign()' de JSON Web Token va combiner ces informations avec un mot de passe défini dans la configuration du projet. Cela va lui permettre de générer le token.

Ensuite, j'ai mis en place la fonction qui va 'hasher', ou crypter, le mot de passe. Pour cela, bcrypt propose de générer un 'salt', un agrément au mot de passe pour le sécuriser davantage. La fonction retourne ensuite l'objet de connexion du user qu'elle a reçu en argument. Elle lui applique la méthode 'hashSync()' qui va combiner l'e-mail, le mot de passe et le 'salt' pour générer le 'hash':

```
const formatPassword = (user) => {

  const salt = bcrypt.genSaltSync(10);
  return {

    mail: user.mail,
    hash: bcrypt.hashSync(user.mail + user.password, salt)

  }
};
```

Une fois ces fonctions de sécurisation mises en place, j'ai pu créer la première propriété du contrôleur d'authentification, '`register:`'. Celle-ci prend la place d'une méthode '`create: `' qu'on aurait apposé dans le contrôleur Users:

```
register: (req, res) => {
  const newUser = new User(formatPassword(req.body));
  newUser.save()
  .then (user => {
    const token = generate_token(user[0]);
    res.send(token)
  })
  .catch(err => {
    res.send('Registration operation failed because' + err)
  })
```

Le point de départ est la création d'une nouvelle instance du modèle User mais avec, en argument, la fonction d'encryptage du mot de passe. Elle sera exécutée sur les informations fournies dans l'interface via un formulaire d'inscription. Puis, ce nouvel utilisateur est enregistré grâce à Mongoose avant que la promesse ne génère un token, grâce à la fonction prévue à cet effet, et le renvoi. Il pourra alors accompagner le visiteur tout au long de sa navigation pour permettre d'accéder aux pages nécessitant d'être authentifié.

La méthode '`login: `', quant à elle, va effectuer une recherche du user en se basant sur son adresse e-mail grâce à la méthode '`find()`' de Mongoose. Une fois trouvé, la promesse va prendre le relais pour comparer le mail et le mot de passe entrés par l'utilisateur à l'aide de la méthode '`CompareSync()`' qui y appose le salt. Si cette comparaison est validée, je vais de nouveau générer un token pour donner accès aux zone restreintes du site:

```
 login: (req, res) => {
    User.find({mail: req.body.mail})
    .then(user =>{
      if (user.length > 0 && bcrypt.compareSync(req.body.mail +
req.body.password, user[0].hash)){
        const token = generate_token(user[0]);
        res.send(token);
      }
    })
    .catch(err =>{
      res.send('Login failed because of this error ' + err)
    })
  }
```

Enfin, une troisième propriété appelée ‘`require_token: `‘ du contrôleur d’authentification va servir de middleware (voir pages suivantes) pour autoriser l’accès à certaines routes et données et n’est donc pas liée à la procédure d’inscription ou de connexion:

```
require_token: (req, res, next) =>{
    const token = req.query.token;
    if (!token) {
      res.send('Authorization required');
    } else {
      jwt.verify(token, global.config.APP_SECRET, (err, decoded) =>{
        if (err || decoded.exp < moment().unix()) res.send('Token
Expired');
        else if (err) res.send('Unauthorized');
        else {
          req._userMail = decoded.iss;
          next();
        }
      });
    }
  }
```

Cette méthode va d’abord tenter de récupérer un token dans le Header de la requête. J’ai alors établi plusieurs conditions qui vont vérifier à la fois la présence effective d’un token et s’il est encore valable. La méthode ‘`.verify()`’ de JSON Web Token permet de décoder ce token pour vérifier s’il n’a pas expiré ou si une erreur s’est produite. Si aucun problème n’est avéré sur ce token, alors je peux définir l’e-mail de l’auteur de la requête comme la valeur de l’auteur du token. Enfin, la méthode ‘`next()`’ est propre aux middlewares car son rôle est de transmettre l’instruction d’exécuter la méthode qui suit après la validation des conditions dictées.

Routes

Toutes ces méthodes sont destinées à être exécutées lorsqu'une requête au serveur a lieu. Par exemple, lorsque, dans l'interface, un bouton est programmé pour afficher des données, il doit alors lancer une requête de type GET. Et il doit y avoir une fonction associée qui ira récupérer les données attendues pour les lire afin qu'elles puissent être affichées.

C'est dans le fichier 'index.js' à la racine du dossier 'api' que l'on va définir les routes de l'API qui vont mettre en correspondance les méthodes issues des contrôleurs avec les type de requêtes.

Apres avoir fait appel au composant '`Router`' du framework ExpressJS et créé une nouvelle instance (stockée dans une variable) ici:

```
const {Router} = require('express');
const apiRoutes = new Router();
```

J'ai importé chacun des contrôleurs pour pouvoir employer leurs méthodes:

```
const userController = require('./controllers/ctrl.users');
const productsController = require('./controllers/ctrl.products');
const authController = require('./controllers/ctrl.auth');
```

Grâce à ExpressJS, j'ai pu appliquer les méthodes de requêtes `.get()`, `.post()`, `.put()`, et `.delete()` facilement et directement sur son composant `Router()`. Ces fonctions permettent de faire appel aux méthodes de requêtes HTTP du même nom pour respectivement récupérer, écrire, mettre à jour ou supprimer des données.

C'est donc ainsi que j'ai défini les routes comme ici, par exemple, afin d'accéder à la collection des produits:

```
apiRoutes.get('/products', productsController.find);
```

Pour obtenir les données de tous les produits, j'ai associé une requête de type GET sur l'url '`/products`' (l'url sera complétée par le nom de domaine automatiquement), avec la méthode '`find:` ' du contrôleur productsController que j'avais créé précédemment. Ensuite, dans mon application 'front-end', je vais stocker les données obtenus dans une variable sur laquelle je vais pouvoir effectuer des opérations.

De même, pour enregistrer un nouveau produit dans cette collection, j'ai procédé par le biais de la méthode POST, sur la même URL mais avec la propriété 'create: ' du contrôleur, cette fois:

```
apiRoutes.post('/products', authController.require_token,
productsController.create);
```

A la différence de la route précédente, celle-ci contient dans ces arguments, une méthode appartenant au contrôleur de l'authentification, appelé 'middleware'. Ce type de méthode a un rôle d'intermédiaire. C'est pourquoi elle apparait entre l'url et la fonction recherchée. Et dans le cas d'une authentification, il s'agit de vérifier que le visiteur souhaitant créer un produit est bien authentifié sur le site avant d'écrire des données. Ce middleware peut être placé dans toutes les routes qui doivent être accessibles uniquement aux users connectés à l'application.

Enfin, pour faire des requêtes qui concerne un document en particulier, j'ai ajouté une variable à l'url pour spécifier l'ID de l'élément concerné. Ces requêtes vont toucher tous les types à l'exception de POST, puisque ce mode va ajouter un élément à une collection. Tandis que les autres types de requêtes peuvent concerner un seul ou plusieurs documents. L'url ressemblera donc à cela :

```
apiRoutes.get('/products:id', productsController.findOne);
```

':id' a été annexé au chemin '/products' et sera remplacé par l'ID de l'élément que l'on souhaite cibler.

Ici, c'est la méthode du contrôleur 'findOne' qui est utilisée pour traiter la data de manière particulière contrairement à la méthode 'find' qui, elle, traite une collection entière.

Pour finir, l'ensemble de ces routes, appartenant au dossier 'api', est importé dans un fichier appelé 'routes.js' et qui va configurer l'utilisation de celles-ci dans l'application. Ce fichier se situe a la racine du dossier 'back-end' ('app') et va permettre d'utiliser les routes définies dans 'api' grâce à la méthode '.use()' :

```
routes.use('/api', apiRoutes);
```

C'est également au sein de ce fichier que j'ai configuré la route qui va servir de point d'entrée vers l'application 'front-end':

```
routes.use(express.static('./public/dist/index.html'));
```

Serveur

Pour les besoins de cette application, j'ai également développé un serveur à la racine du projet. Dans ce serveur NodeJS, j'ai importé les dépendances globales du projet telles que le fichier de configuration mentionné plus tôt et le framework ExpressJS. Une variable 'app' vient ensuite créer une nouvelle instance de ce framework:

```
const app = express();
```

C'est sur cette instance que le fonctionnement 'back-end' sera configuré, notamment pour spécifier les routes que le server va utiliser, encore grâce à la méthode ExpressJS '`.use()`'.

```
app.use(routes);
```

J'y ai également spécifié le port à utiliser, à partir de la configuration globale, et avec la méthode '`.listen()`'.

```
app.listen(global.config.port, err => {
  console.log('Server running on ' + global.config.port);
});
```

Bilan et conclusions

Pour conclure, ce projet a été très bénéfique pour ma progression en tant que développeur. J'ai pu consolider et approfondir mes connaissances dans le langage Javascript, tout en apprenant une quantité considérable d'autres outils et méthodes.

J'ai notamment découvert le framework 'front-end' VueJS car je souhaitais profiter de l'opportunité de ce projet pour appréhender une nouvelle approche plus récente dans un écosystème qui évolue à grande vitesse. En effet, j'ai aussi pu utiliser de nouveaux outils comme Webpack pour apprendre une nouvelle façon de compiler et générer des applications, complètement différente des projets plus traditionnels.

J'ai donc développé une interface complète et responsive grâce à VueJS et mes connaissances avec le préprocesseur SASS. Cela a renforcé mes capacités avec cet outil que j'ai mis au profit d'une véritable application web.

Cette application a aussi été l'occasion de réellement connaitre le développement Javascript 'back-end' avec NodeJS et ExpressJS que je n'avais abordé que de manière succincte pendant ma formation qui est plutôt orientée 'Front-end'. Cela m'a permis d'expérimenter le développement 'full-stack', c'est à dire autant 'front' que 'back' et l'utilisation d'une base de donnée.

Au delà de mise en oeuvre technique, la formulation et l'explication de celle-ci pour les besoins de ce dossier a été bénéfique pour renforcer mon apprentissage. J'ai du décomposer tous les éléments du projet pour indiquer leur utilité et leur fonctionnement.

Enfin, ce projet m'a permis de mettre en route un projet personnel que j'amènerai certainement plus loin afin de le réaliser.