Rapport de projet
« Café sans-fil »

Axel ZAREB et Southidej OUDANONH

IFT 3150 – Projet Informatique

2023

Supervisé par Louis-Édouard LAFONTANT

Table des matières

Table de figures

Introduction

Contexte

Les cafés étudiants de l'Université de Montréal (UdeM) jouent un rôle majeur dans la vie de campus et dans l'expérience globale et la satisfaction des étudiants au sein de l'université. Ils sont répartis à travers les pavillons de l'UdeM et sont entièrement gérés par des étudiants bénévoles. Ils offrent un espace de rencontre, de détente, et permettent surtout de se nourrir à bas coût. Toutefois, l'infrastructure actuelle limite leur potentiel et cause frustrations et confusion auprès des étudiants.

Problématique

Actuellement, il y a un manque d'information au sujet des différents cafés étudiants de l'UdeM. Il existe une page web (sur le site de l'UdeM) qui présente l'ensemble des cafés étudiants, mais l'information présentée est incomplète et parfois obsolète. Les cafés étant gérés par des bénévoles, les horaires sont parfois imprévisibles et les locaux peuvent être fermés en période d'examen. De plus, l'approvisionnement en produits n'est pas toujours le même, le menu n'est parfois pas clairement affiché, ou notre sandwich préféré est en rupture de stock. Certains cafés ne sont pas en mesure d'accepter tous les moyens de payement, ou bien ont un montant minimum d'achat pour payer par carte de crédit. Les clients doivent avoir connaissance de ces informations. Des cafés disposent d'une page web ou d'un compte Facebook ou Instagram qui leur est propre, mais il n'est pas évident pour les clients de consulter des informations dispersées, et la découverte de nouveaux cafés n'est pas mise en avant. Il y a un réel besoin de changement.

Proposition

Afin d'améliorer le service des cafés étudiants de l'UdeM et répondre aux problèmes identifiés précédemment, nous proposons **Café sans-fil**, une plateforme commune pour les cafés étudiants et une source d'information fiable.

La plateforme dispose d'une API et d'une application web. L'API peut être interrogé par tout client cherchant à récupérer de l'information sur les cafés. Elle peut également être utilisée pour développer d'autres canaux de présentation comme une application mobile. L'application web offre une interface simple et intuitive qui facilite la recherche et la découverte de cafés, la consultation des menus à jour, la prise de commandes, et la gestion des informations liées aux cafés bénévoles de l'UdeM. En plus de satisfaire et améliorer l'expérience des clients (étudiants, professeurs, etc.), l'application offre aux bénévoles et aux responsables (gérants) des cafés des outils de gestion efficaces, permettant une organisation plus efficace et agréable.

Analyse

Afin de comprendre au mieux les exigences du projet, nous avons identifié les acteurs principaux des cafés et élaboré des personas capturant les divers profils de nos acteurs. Durant ce processus, nous avons fait deux entrevues avec des clients potentiels telles que l'un des gérants du café *Tore et fraction*[1] pour affiner les besoins du projet et obtenir une meilleure compréhension du domaine.

Utilisateurs

Acteurs

Les cafés étudiants font intervenir quatre acteurs : **grand public** (personne non authentifiée ou non-membre de l'UdeM), **membre de l'UdeM** (étudiant, employé, professeur, chercheur, etc.), **bénévole de café** (étudiant travaillant au café) et **gérant de café** responsable de la gestion des bénévoles, du menu, et de l'inventaire, assurant la fluidité des opérations quotidiennes.

Personas

Une persona, dans le cadre de la conception centrée sur l'utilisateur (CCU) et du marketing, est une représentation fictive d'une personne dotée de caractéristiques sociodémographiques et psychologiques spécifiques, destinée à représenter un segment précis de la clientèle ou de l'audience. Ci-dessous, nous présentons les personas élaborées pour Café sans-fil :

- **Jeune étudiant de l'UdeM**: Un étudiant en première année, habituellement pressé et cherchant à prendre rapidement son café entre deux cours. Il utilise l'application pour commander à l'avance, optimisant ainsi son temps.
- **Professeur de l'UdeM**: Un professeur senior qui commande régulièrement son déjeuner et son café au même endroit. Il utilise l'application pour retrouver ses commandes favorites et passer sa commande sans avoir à refaire tout le processus.
- **Doctorant de l'UdeM**: En plein milieu de ses recherches, ce doctorant utilise souvent l'application pour commander une collation ou un déjeuner tout en travaillant dans la bibliothèque universitaire.
- **Bénévole au Café**: Un étudiant en troisième année qui travaille comme bénévole au café pour se rapprocher de la communauté universitaire. Il utilise l'application pour gérer les commandes et vérifier l'inventaire.
- **Visiteur occasionnel**: Une personne de passage à l'UdeM pour un événement ou une conférence. Il découvre l'application via une recommandation et l'utilise pour commander quelque chose à boire ou à manger pendant sa visite.
- **Employé de l'UdeM**: Travaillant à l'administration de l'université, cet employé commande son café chaque matin via l'application, appréciant la régularité et la prévisibilité du service.

[1] http://cafemathinfo.iro.umontreal.ca/

Besoins fonctionnels

Ci-contre, nous listons les fonctionnalités recherchées par les quatre acteurs identifiés plus haut. Une présentation plus détaillée est présentée en annexe sous forme de cas d'utilisation (voir annexe 2).

Grand Public

Café sans-fil permet à tous d'explorer et découvrir les cafés étudiants. Ainsi tout utilisateur doit être en mesure de :

- ✓ Voir la liste des cafés
- ✓ Chercher un café par nom ou à l'aide de filtres
- ✓ Voir les informations d'un café
- ✓ Voir le menu d'un café

Membres de l'UdeM

Café sans-fil permet aux membres de l'UdeM de placer des commandes pour ramassage, les permettant de gagner du temps. En plus des fonctionnalités accessibles au grand public, un membre doit être en mesure de :

- ✓ Se créer un compte : ceci nécessite un matricule et une adresse courriel de l'UdeM
- ✓ Se connecter avec son matricule ou son adresse courriel
- ✓ Voir et modifier son profil
- ✓ Passer ou annuler une commande (pour ramassage)
- ✓ Voir l'historique de ces commandes

Fonctionnalités pour les Bénévoles

Café sans-fil permet aux bénévoles de café de traiter les commandes reçues et facilite la gestion du menu. En plus des fonctionnalités accessibles au membre, un bénévole doit être en mesure de :

- ✓ Afficher les commandes à traiter
- ✓ Modifier le statut d'une commande: *Placée*, *Prête*, *Complétée*, *Annulée*
- ✓ Ajouter, modifier ou supprimer un item du menu

Fonctionnalités pour les Gérants

Café sans-fil permet aux gérants de café de gérer efficacement les informations, le personnel et le menu du café et leur fournit des informations sur les activités du café. En plus des fonctionnalités accessibles au bénévole, un gérant doit être en mesure de :

- ✓ Modifier les informations de base d'un café telles que nom, description, horaire
- ✓ Créer une annonce visible par tous telle que l'annonce d'un évènement ou une présentation
- ✓ Ajouter, modifier ou supprimer un bénévole (staff)
- ✓ Générer des rapports de vente : journalières, hebdomadaires, mensuelles
- ✓ Ajouter, modifier ou supprimer un item du menu

Besoins non fonctionnels

Afin d'offrir une très bonne expérience utilisateur (UX) et faciliter le développement de l'application, nous considérons aussi les besoins suivants :

- **Performance :** L'application web doit être réactive et minimiser le temps de chargement pour offrir une UX fluide. L'API doit avoir un délai de réponse maximale de 500 ms.
- **Sécurité :** Les données des utilisateurs doivent être stockées de manière sécurisée, et l'application doit prévenir tout accès non autorisé.
- **Traitement des commandes :** Une commande doit être récupérée dans un délai d'une heure. Au-delà de cette limite, elle est automatiquement annulée pour éviter l'attente inutile du personnel et libérer l'item pour un autre client.
- **Utilisabilité :** L'interface utilisateur doit être intuitive, facile à utiliser, esthétiquement plaisante et compatible avec différents formats d'écrans. Elle doit être accessible aux personnes ayant des handicaps mineurs, et plus particulièrement celles utilisant un lecteur d'écran.
- **Maintenabilité :** Le code de l'application doit être documenté, structuré et facile à maintenir. Il doit utiliser des langages et librairies populaires et accessibles aux étudiant du DIRO.
- **Fiabilité :** L'information présentée par l'application doit être à jour et exacte avec une probabilité de 95%.

Risques

Nous avons identifié et évalué les divers risques associés à la réalisation du projet Café sans fil. La gestion proactive de ces risques est essentielle pour garantir le succès du projet et minimiser les impacts négatifs sur son déroulement.

Le succès du projet dépend fortement de la compréhension des besoins des divers acteurs et de l'adoption de la plateforme par ceux-là. Donc, une erreur dans l'élaboration des exigences ou un faible taux d'adoption est un risque majeur pour ce projet. Afin d'atténuer ce risque, nous avons fait des retours fréquents auprès des utilisateurs et des cafés pour recueillir des commentaires, et ajustements en fonction des besoins.

La plateforme permet de passer des commandes en ligne qui change le fonctionnement actuel des cafés. Cette fonctionnalité pourrait créer de la surcharge de travail pour les bénévoles, qui doivent traiter à la fois les commandes en ligne et en personne. Pour y remédier, nous avons mis en place un système simple et efficace de gestion des commandes, et optimisation des processus. Un second risque associé à cette fonctionnalité est la confusion qui pourrait survenir lorsqu'une commande fait intervenir plusieurs cafés. Pour minimiser cette confusion, l'interface présente une division claire des commandes par café en indiquant explicitement les items à récupérer de chaque café. Finalement, la commande en ligne risque de réduire les interactions au café et ainsi impacter l'aspect social des cafés étudiants. Quoique nous n'envisageons pas un impact majeur, nous adressons ce risque en intégrant des fonctionnalités favorisant la communication entre les clients et les bénévoles, et faisant la promotion d'événements sociaux dans les cafés.

Finalement, la plateforme jouant le rôle de référence doit s'assurer de l'exactitude des informations présentées. Il y a le risque que les cafés ne suivent pas leurs horaires, entraînants des désagréments pour les clients ou qu'un item épuisé soit présenté à l'utilisateur. Pour atténuer ce risque, l'application

permet aux cafés de mettre à jour manuellement leur statut d'ouverture/fermeture en temps réel sur l'application et d'indiquer si un item est épuisé.

Choix technologique

L'infrastructure de l'application repose sur le FARM stack, comprenant *FastAPI*, *React* et *MongoDB*, qui sont tous gratuit et open source. Cette architecture a été choisie pour sa robustesse, sa réactivité et sa facilité d'extension. La base de données MongoDB assure une gestion efficace des données, tandis que FastAPI et React fournissent une interface utilisateur dynamique et une API performante.

MongoDB

Le choix de **MongoDB** comme système de gestion de base de données pour le projet Café sans-fil s'est imposé pour plusieurs raisons stratégiques, notamment sa flexibilité et sa capacité à gérer la diversité des données.

Gestion de la diversité des cafés : Les cafés de l'Université de Montréal présentent des caractéristiques uniques, avec des besoins et des offres variées. MongoDB, grâce à sa nature flexible, nous permet d'adapter facilement la base de données à la spécificité de chaque café. Contrairement aux bases de données relationnelles comme PostgreSQL, l'évolution de la structure de données est moins complexe avec MongoDB, ce qui facilite l'ajout ou la modification des informations relatives à chaque café.

Scalabilité et évolutivité : En anticipant l'évolution du projet et la possible extension à d'autres zones du campus ou même à d'autres universités, MongoDB offre une meilleure scalabilité. Cette caractéristique est essentielle pour gérer efficacement l'augmentation du volume de données et le nombre croissant d'utilisateurs.

FastAPI

L'adoption de **FastAPI**[2] pour développer l'interface de programmation de notre projet Café sans-fil a été guidée par des considérations stratégiques et techniques clés.

Popularité de Python : Envisageant que des étudiants de l'université pourraient éventuellement reprendre ou contribuer au projet, FastAPI présente l'avantage d'être facile à apprendre et à utiliser. Reposant sur Python, nous tirons profit de la popularité grandissant du langage[3] et de l'apprentissage de Python assuré par le cours de programmation I (IFT1015). L'utilisation de FastAPI, étroitement intégré avec Python, nous permet de tirer parti de cette vaste communauté. Sa simplicité et sa clarté rendent le code plus accessible aux contributeurs potentiels, favorisant ainsi la continuité et l'évolutivité du projet au sein de la communauté universitaire.

Performance élevée : FastAPI se démarque des autres *framework* Python tel que Flask par sa haute performance. Cette performance est essentielle pour gérer les requêtes en temps réel dans notre application, assurant une expérience utilisateur fluide et réactive.

---

[2] https://fastapi.tiangolo.com/

[3] Python est reconnu comme le langage le plus populaire selon un sondage de IEEE Spectrum (2023) : https://spectrum.ieee.org/the-top-programming-languages-2023

Facilité de développement : FastAPI simplifie le développement grâce à sa facilité d'utilisation et sa lisibilité. Il prend en charge la génération automatique de documentation (avec *Swagger UI* et *ReDoc*), ce qui facilite la collaboration et la maintenance du code. Cela réduit le temps de développement et améliore l'efficacité de notre équipe.

React

Pour le développement l'application (front-end), nous avons choisi **React**[4] pour plusieurs raisons stratégiques et techniques, dont notamment sa reprise et son évolution par de futurs développeurs.

Popularité : React est l'une des librairies JavaScript les plus populaires et largement utilisées, souvent considéré comme le choix de facto pour le développement d'application web. Sa popularité assure un vaste écosystème de développeurs et une abondance de ressources d'apprentissage, ce qui est essentiel pour la facilité de reprise et de contribution au projet par les étudiants et les développeurs futurs. Cette popularité favorise la maintenance et l'évolution de l'application.

Développement de Single-Page Applications (SPA) : Aujourd'hui la plupart des petites applications web se développent sous forme de SPA qui évite le chargement d'une nouvelle page à chaque action demandée. React est particulièrement efficace pour le développement de single-page applications. Cela contribue à une expérience utilisateur rapide et réactive, un aspect crucial pour l'engagement des utilisateurs de l'application Café sans-fil.

Modularité : Le cadre de travail en composants distincts de React facilite la maintenance et l'évolution du code. Chaque composant encapsule sa propre logique, rendant les modifications et l'ajout de nouvelles fonctionnalités plus gérables et intuitifs. Cette structure modulaire permet une réutilisation efficace du code et simplifie la tâche pour les futurs développeurs qui reprennent le projet.

TailwindCSS

En plus de React, nous avons choisi d'intégrer **TailwindCSS**[5] pour plusieurs raisons stratégiques, alignées avec notre objectif de créer une interface utilisateur efficace et attrayante.

Collaboration facile : TailwindCSS est conçu pour faciliter la collaboration et la reprise du code. Sa facilité de prise en main et sa popularité croissante en font un choix judicieux pour un projet où la maintenance et l'évolution du code sont des considérations clés. Avec TailwindCSS, les modifications sont plus prévisibles et moins susceptibles de causer des effets indésirables sur d'autres parties du projet, assurant ainsi une plus grande stabilité du code.

Léger et customisable : TailwindCSS adopte une approche utilitaire, minimisant son empreinte dans le code, rendant l'application plus légère et performante. Contrairement à des frameworks plus génériques comme Bootstrap, TailwindCSS permet de créer des interfaces uniques propres aux besoins spécifiques de l'application, favorisant une bonne expérience utilisateur.

---

[4] https://react.dev/

[5] https://tailwindcss.com/

Conception

Base de données

L'Annexe 1 présente le schéma de la base de données (BD). Il est important de noter que le schéma est présenté suivant le modèle Entité-Relation (ERD) à des fins de présentation uniquement. Notre base de données est de type NoSQL. Elle n'est pas contrainte de respecter les standards des systèmes relationnelles, ce qui facilite l'évolution horizontal de la BD et permet de structurer les données de manière plus fidèle au domaine.

Entités principales

Notre système est structuré autour de trois entités clés, formant la base de l'application :

- *Cafe* : Cette entité contient toutes les informations relatives à chaque café, incluant des détails tels que l'emplacement, les heures d'ouverture, et le menu. Le menu est incorporé à cette entité pour refléter directement la variété des offres disponibles dans chaque café.
- *User*: Cette entité gère les données des utilisateurs de l'application, y compris l'authentification et les préférences personnelles. Elle joue un rôle central dans la personnalisation de l'expérience utilisateur et dans la gestion des interactions avec l'application.
- *Order* : L'entité *Order* traite toutes les commandes passées par les utilisateurs. Elle inclut des informations détaillées sur les articles commandés, le statut de la commande, et les relations avec les entités *Cafe* et *User*.

Identifiant

Chaque entité majeure de notre base est identifiée par un identifiant universel unique (*UUID*), garantissant leur unicité au sein de la base de données.

Slug

Le *slug* est une chaîne de caractères qui reflète généralement le nom de l'entité de manière simplifiée et peut être utilisé pour identifier de manière unique une ressource en URL. Il est présent dans les entités *Cafe* et *MenuItem* afin de créer des URL lisibles et optimisées pour le référencement.

Heures d'ouverture

L'attribut *opening_hours* utilise des blocs pour permettre aux cafés de définir des horaires avec des pauses intermédiaires si nécessaire. Cela offre une flexibilité pour que les cafés puissent définir des horaires non continus, tels que des heures d'ouverture le matin et l'après-midi séparées par une pause.

Personnel du café

L'attribut *staff* dans l'entité *Cafe* répertorie le personnel (bénévole et gérant) d'un café. Étant une BD NoSQL, nous pouvons définir *staff* comme un *array* de ID, sans table (entité) additionnel.

Menu

Dans les faits, chaque café possède son propre menu. Afin de refléter cette relation en BD, nous tirons profit de la notion d'imbrication retrouvé dans les BD NoSQL où une entité de contient des sous-entités. Chaque café contient des *MenuItems*. Les *MenuItems* représentent les différents produits disponibles

dans chaque café, comme des boissons ou des collations. Chaque *MenuItem* peut avoir des *MenuItemOptions*, qui offrent des choix supplémentaires comme la taille (petit, moyen, grand) ou des garnitures (par exemple, tomate, fromage, etc.).

Commande

L'entité *Order* encapsule les commandes des clients et présente quatre statuts principaux :

- *PLACED / Placée* **:** L'état initial lorsque la commande est passée par le client.
- *READY / Prête* **:** Indique que la commande est prête à être récupérée.
- *COMPLETED / Complétée* **:** Signifie que la commande a été récupérée et complétée.
- *CANCELLED / Annulée* **:** Signifie que la commande a été annulée.

Mesure de sécurité

Pour prévenir les tentatives de connexion frauduleuses ou les spams, nous utilisons les attributs *failed_login_attempts* et *last_failed_login_attempt*. Ces attributs aident à suivre les échecs de connexion consécutifs et à bloquer temporairement l'accès en cas de comportements suspects. L'attribut *lockout_until* permet de définir une période de verrouillage durant laquelle l'utilisateur ne peut pas se connecter, renforçant ainsi la sécurité contre les attaques de force brute.

Suppression douce (Soft delete)

Nous utilisons un système de suppression « douce » pour traiter la suppression des utilisateurs. Lorsqu'un utilisateur est désactivé (*is_active* = **false**), il n'est plus possible de rechercher ou d'interagir avec son compte. Cette approche permet de conserver les données pour d'éventuelles restaurations ou analyses ultérieures.

Stockage des données

La plupart des données utilisées par l'application, telles que les informations utilisateur, sur les cafés et les commandes sont stockés dans la BD et retournées par l'API.

Afin d'améliorer l'expérience utilisateur, nous utilisons aussi la cache du navigateur pour préserver certaines données de l'utilisateur. Par exemple, le panier d'achat de l'utilisateur est enregistré en *localStorage*[6] dans le navigateur. Cela permet à l'utilisateur de retrouver son panier, et de conserver son contenu avant et après l'authentification et lorsqu'il ferme et rouvre ce navigateur. Voici un exemple de ce qui est stocké. Nous utilisons aussi le *localStorage* afin de conserver les informations utilisateur, dont ses identifiants (*username* / *matricule*) et les tokens qui servent à effectuer un grand nombre d'autres requêtes dans l'application. En sauvegardant ces données, nous réduisons la quantité de requêtes à faire à l'API, améliorant ainsi la performance de l'application et l'expérience utilisateur. Par exemple, la page de profil charge beaucoup plus vite en sauvegardant ces données de côté. Voici un exemple de ce qui est sauvegardé en plus du panier d'achat.

[6] https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

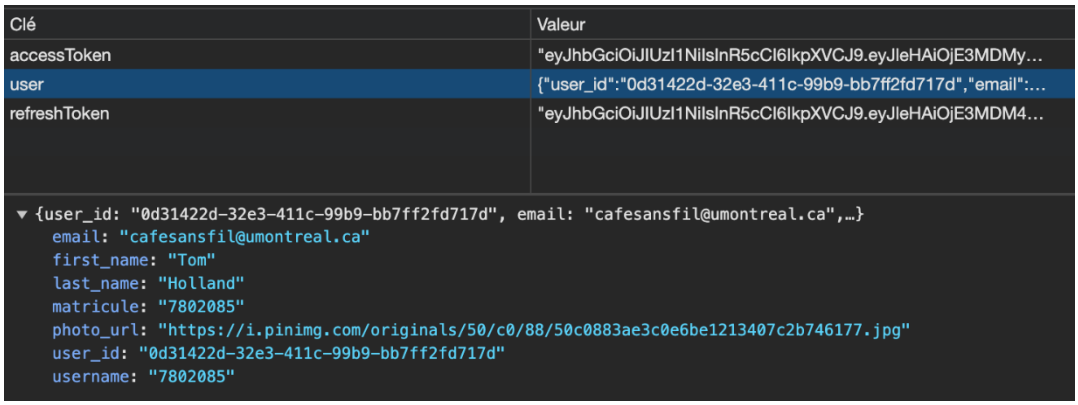


Figure 1 - Stockage local des tokens d'authentification et des informations utilisateur

Cette solution de stockage présente un inconvénient. En effet, si l'utilisateur modifie ses informations de profil, les changements ne se propagent pas automatique sur les autres appareils où il est connecté. Les modifications prendront effet uniquement lorsqu'il se reconnectera, puisque nous stockons ces données à la connexion, et les effaçons à la déconnexion.

API

L'Annexe 3 présente les composantes principales de l'API sous forme de diagramme de paquets. Ce diagramme illustre l'organisation des composantes de l'API et la manière dont elles interagissent pour fournir les fonctionnalités requises. Ci-dessous, nous présentons chacun de ces paquets :

- *handlers* : Ce paquet contient les contrôleurs de notre API. Ils gèrent les requêtes entrantes et sortantes, en orchestrant la logique de l'application. Chaque handler est responsable d'une route spécifique et fait appel aux services appropriés pour traiter les données.
- *services* : Ce paquet encapsule la logique métier de l'application. Il sert de couche intermédiaire entre les *handlers* et les *models*, exécutant les opérations nécessaires telles que la création, la mise à jour ou la récupération de données.
- *auth* : Ce paquet est dédié à l'authentification et à la sécurité. Il contient les mécanismes de gestion des tokens JWT, de hachage des mots de passe et de validation des sessions utilisateurs.
- *models* : Ce paquet regroupe les structures de données utilisées pour communiqué avec la BD.
- *schemas* : Ce paquet regroupe les structures utilisées pour la communication avec les clients de l'API. Il présente aussi les schémas servant à valider les données reçues et transmises.

Composants du front-end

La structure de l'application web, suit une structure standard de projet React. Le point d'entrée principal est *main.jsx*, où les routes de l'application sont définies. À ce niveau, plusieurs dossiers courants de React sont présents:

- *assets* **:** Ce dossier contient les ressources statiques telles que les icônes utilisées dans l'application.

- *components* **:** C'est dans ce dossier que sont regroupés les composants React réutilisables. Chaque composant est conçu pour accomplir une tâche spécifique, favorisant la modularité du code. Ils sont regroupés en sous-catégories en fonction de leur utilité, par exemple : *Cafe*,

Items, *Search*, *Orders*. Dès qu'une partie d'une page devient complexe, ou est réutilisée dans plusieurs pages, nous la transformons en composant séparé, afin de séparer la logique et avoir un code plus clair.

- *helpers* **:** Des fonctions logiques sont regroupées ici. Ces *helpers* sont des modules réutilisables facilitant la cohérence et la lisibilité du code. Il peut s'agir de la logique permettant de restreindre certaines pages aux gérants ou membres d'un café seulement, ou gérer les requêtes authentifiées à l'API.

- *hooks* **:** Les *hooks* personnalisés sont regroupés dans ce dossier. Les *hooks* permettent de réutiliser de la logique d'état dans les composants fonctionnels et sont une partie intégrante de React. Il peut s'agir de logique pour gérer un compte à rebours de 1h avant d'annuler une commande, ou changer l'UI en fonction de si un élément est visible sur l'écran de l'utilisateur.

- *routes* **:** Ce dossier contient les définitions des routes de l'application. Chaque route est associée à un composant spécifique à afficher lorsqu'elle est atteinte. Nous avons donc ici toutes les pages qui ont leur propre URL, tel que *Home*, *Cafe*, *Login*, *Signup*, etc.

- *utils* **:** Les utilitaires généraux, tels que des fonctions permettant de manipuler des dates, des horaires, ou définir des constantes, sont stockés ici. Ces utilitaires permettent de réutiliser des valeurs fixes ou des petites fonctions très simples dans plusieurs composants.

Cette organisation structurelle facilite la maintenance, la compréhension, et favorise l'évolution de l'application en suivant les meilleures pratiques de développement React. Chaque dossier joue un rôle défini, permettant aux développeurs de travailler de manière collaborative et efficace sur différentes parties de l'application.

Implémentation

Pour développer l'API, un environnement virtuel Python a été mis en place avec pipenv, afin d'avoir un environnement stable pour gérer les dépendances Python. Pour visualiser la base de données, nous avons utilisé MongoDB Compass. Pour le Front-end, l'application React a été initialisée avec Vite. C'est l'outil le plus répandu actuellement pour exécuter un serveur local de développement et compiler les fichiers JavaScript en un bundle qui sera déployé ensuite en production.

Vite permet également le « hot-reloading », qui va recharger l'application web dynamiquement et instantanément à chaque changement dans le code, en fonction de ce qui a été modifié. Cela permet de voir ses modifications en temps réel sans avoir à recharger l'ensemble de l'application à chaque fois.

Durant le développement, plusieurs modifications à la structure des données ont été faites. Ces changements n'étaient pas très compliqués grâce à la grande flexibilité de MongoDB.

Nous avons par exemple dû ajouter des champs dans la gestion des commandes. En effet, pour récupérer la liste des commandes d'un utilisateur, la réponse GET orders contenait les UUID des *MenuItems*, et UUID du café où la commande était passée. Cependant, pour afficher toutes les informations comme le nom du produit, et son image, nous devions faire une trentaine d'appels API, une pour chaque produit...

De plus, il était possible de tomber sur un produit qui avait été depuis supprimé par le café, dans ce cas on affichait « Produit supprimé ». Mais ceci n'est pas optimal, il fallait que les informations de base sur les produits d'une commande soient stockées dans les données de la commande. Nous avons donc implémenté ces modifications et réduit le temps de chargement de la page « Mes commandes » de plus de 80%, et le nombre de requêtes de parfois plus de 30 à seulement 2 (une pour récupérer les commandes, une pour récupérer le nom et l'image du café où est passée la commande.

Une des autres difficultés que nous avons rencontrées est la gestion des heures. En effet, il fallait faire en sorte que l'application renvoie si un café est ouvert en fonction de ses horaires d'ouverture, et cela a causé beaucoup de problèmes. Il fallait s'assurer que l'heure était bien convertie en timezone de Montréal, et que ça fonctionne pour les gens qui ont l'heure sur leur appareil en format 12 heures et en format 24 heures.

Nous vous invitons à regarder les captures d'écran de l'application web en annexe, mais surtout à la tester par vous-même!

API

Paramètres de requête

Les paramètres de requête sont utilisés pour filtrer, trier et paginer les résultats retournés par l'API. Voici des paramètres communs que notre API peut prendre en charge :

- *is_open* : Permet le filtrage des ressources basé sur leur statut d'ouverture.
- *sort_by* : Permet de trier les résultats basé sur un champ spécifique. Un préfixe '-' indique un ordre décroissant.
- *page* : Permet de spécifier le numéro de page lors de l'utilisation de la pagination pour séquencer les résultats.
- *limit* : Définit le nombre maximal de résultats à retourner dans une seule requête.

Par exemple, pour obtenir une liste des cafés fermés, limitée à deux résultats et triée par nom en ordre décroissant, l'URL de la requête serait :

https://cafesansfil-api.onrender.com/api/cafes?is_open=false&limit=2&sort_by=-name

Gestion des erreurs

L'API utilise des codes d'état HTTP standard pour communiquer le résultat des requêtes aux clients. Voici des exemples de réponses pour différents codes d'erreur :

- 401 Non Autorisé: Indique que la requête a besoin d'une authentification valide et qu'elle a été refusée ou n'a pas été fournie.
  `{ "detail": "Not authenticated" }` ou
  `{ "detail": "Incorrect email or password" }`
- 403 Interdit: La demande est valide, mais le serveur refuse l'action. L'utilisateur n'a peut-être pas les droits nécessaires pour une ressource, ou cela pourrait nécessiter un compte de niveau supérieur.
  `{ "detail": "Access forbidden" }`

- 404 Introuvable: Le serveur n'a pas trouvé ce que l'utilisateur a demandé. Il est possible que la ressource n'existe pas ou soit indisponible pour d'autres raisons.
  `{ "detail": "Café not found" }`
- 409 Conflit: Il y a un conflit dans la demande, comme tenter de créer une ressource qui existe déjà.
  `{ "detail": "User with this matricule already exists" }`

Gestion des autorisations

La gestion des autorisations est un pilier fondamental pour assurer une utilisation sécurisée de l'application. Nous avons élaboré cette gestion en prenant appui sur le diagramme de cas d'utilisation, mentionné dans l'Annexe 2, qui détaille les interactions possibles entre les utilisateurs et le système en fonction de leurs rôles respectifs.

Le processus d'authentification des utilisateurs et la validation des tokens JWT sont pris en charge par le middleware d'authentification situé dans le module *user_deps.py*. Cette composante clé vérifie l'intégrité et la validité des tokens JWT, affirmant l'identité des utilisateurs et sécurisant l'accès aux fonctionnalités de l'API.

L'API intègre des contrôles d'accès basés sur les rôles dans ses différents composants dans le paquet *handlers*. Ces contrôles régulent l'accès aux fonctionnalités en fonction du rôle de chaque utilisateur. Ainsi, chaque endpoint effectue des vérifications spécifiques pour déterminer si l'utilisateur courant a les autorisations nécessaires pour réaliser l'action demandée. Par exemple, la fonction *is_authorized_for_cafe_action_by_slug* du paquet services est fréquemment utilisée pour valider les rôles des utilisateurs avant de permettre l'exécution d'opérations critiques, assurant ainsi que les droits d'accès sont respectés à chaque interaction avec l'API.

Module de sécurité JWT

La sécurisation des échanges d'informations entre le client et le serveur est garantie par l'utilisation de Jetons Web JSON (**JWT**), qui sont générés via l'algorithme **HS256** lors de l'authentification des utilisateurs. Nous produisons deux types de tokens : un token d'accès à durée limitée pour l'authentification immédiate (**Access token**) et un token de rafraîchissement conçu pour une validité prolongée (**Refresh token**), permettant ainsi aux utilisateurs de maintenir leur session active en toute sécurité.

La gestion des tokens est capitale dans notre processus de sécurité. Les tokens ont une date d'expiration définie, et tout token invalide ou expiré provoque la déconnexion de l'utilisateur, ce qui garantit que seules les personnes authentifiées et autorisées accèdent aux ressources sécurisées de l'API.

Infrastructure de sécurité

Le module *security.py*, situé dans le dossier *core* de notre architecture, est responsable du chiffrement et de la validation des mots de passe. En parallèle, le fichier *jwt.py* dans le dossier *auth* est chargé de la création et de la validation des tokens JWT. Quant à *user_deps.py* dans le dossier *deps*, il définit les dépendances pour extraire l'utilisateur courant à partir du token JWT.

Cette infrastructure de sécurité JWT est essentielle pour protéger l'application contre les accès non autorisés et pour garantir que les données utilisateur restent confidentielles et sécurisées tout au long de leur interaction avec le système Café sans-fil.

Tests et déploiement

Tests

Le processus de test a été conçu pour assurer la qualité de notre application et vérifier les comportements de l'API et l'application web.

Tests fonctionnels

Nous avons principalement utilisé **Postman**[7] pour tester les différentes routes de notre API. Cela nous a permis de simuler des requêtes client et de vérifier les réponses de l'API, assurant ainsi que toutes les fonctions fonctionnent comme prévu. Postman offre une interface conviviale pour créer, partager et documenter les tests d'API, ce qui facilite la collaboration entre les membres de l'équipe.

Tests utilisateur

Quoique nous n'avons pas fait des tests utilisateurs formels, nous avons recueilli des retours informels des gérants de cafés et de notre entourage concernant l'interface utilisateur et les fonctionnalités de l'application. Ces retours n'ont pas été collectés dans le cadre d'une démarche formelle, mais se sont avérés extrêmement précieux. Ces commentaires spontanés nous ont fourni des perspectives authentiques et directes sur l'expérience utilisateur réelle, mettant en évidence des aspects spécifiques de l'interface et des fonctionnalités nécessitant des ajustements. Ces retours ont joué un rôle clé dans l'affinement de l'intuitivité et de l'efficacité de notre application, nous permettant d'apporter des améliorations significatives basées sur les expériences réelles des utilisateurs.

Tests unitaires

Nous avons mis en place 82 tests unitaires en utilisant **pytest**[8], un framework de test puissant et flexible pour Python. Ces tests couvrent chaque point de terminaison possible de notre API.

Chaque test vérifie les réponses spécifiques de l'API, incluant les codes de statut HTTP comme 200 (***Success/Succès***), 404 (***Not Found/Non trouvé***), 403 (***Forbidden/Interdit***), et d'autres réponses pertinentes telles que 401 (***Unauthorized/Non authentifié***). Cette approche exhaustive nous a permis de tester rigoureusement le comportement de l'API dans une multitude de situations, garantissant sa fiabilité et sa robustesse face à différentes requêtes et scénarios d'utilisation.

[7] https://www.postman.com/

[8] https://docs.pytest.org/en/7.4.x/

```python
def test_create_menu_item_success(client, list_cafes, cafe_data, auth_login):
    menu_item_data = cafe_data["menu_items"][0]
    tokens = auth_login
    headers = {
        "Authorization": f"Bearer {tokens['access_token']}"
    }
    cafe_slug = list_cafes[0]["slug"]
    response = client.post(f"/api/cafes/{cafe_slug}/menu", json=menu_item_data, headers=headers)
    assert response.status_code == 200

def test_create_menu_item_unauthorized(client, list_cafes, cafe_data):
    cafe_slug = list_cafes[0]["slug"]
    response = client.post(f"/api/cafes/{cafe_slug}/menu", json=cafe_data)
    assert response.status_code == 401

def test_create_menu_item_forbidden(client, list_cafes, cafe_data, auth_login):
    menu_item_data = cafe_data["menu_items"][0]
    tokens = auth_login
    headers = {
        "Authorization": f"Bearer {tokens['access_token']}"
    }
    cafe_slug = list_cafes[1]["slug"]
    response = client.post(f"/api/cafes/{cafe_slug}/menu", json=menu_item_data, headers=headers)
    assert response.status_code == 403

def test_create_menu_item_not_found(client, cafe_data3, auth_login):
    menu_item_data = cafe_data3
    tokens = auth_login
    headers = {
        "Authorization": f"Bearer {tokens['access_token']}"
    }
    cafe_slug = "dont exist" # Non-existant slug
    response = client.post(f"/api/cafes/{cafe_slug}/menu", json=menu_item_data, headers=headers)
    assert response.status_code == 404
```

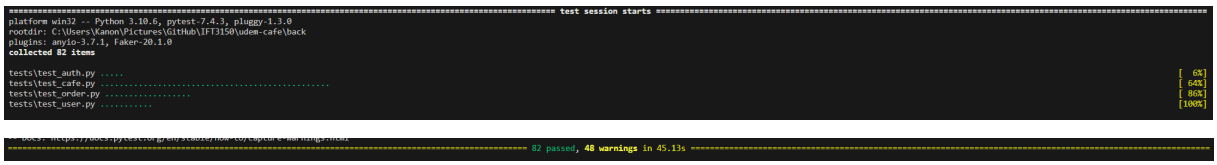


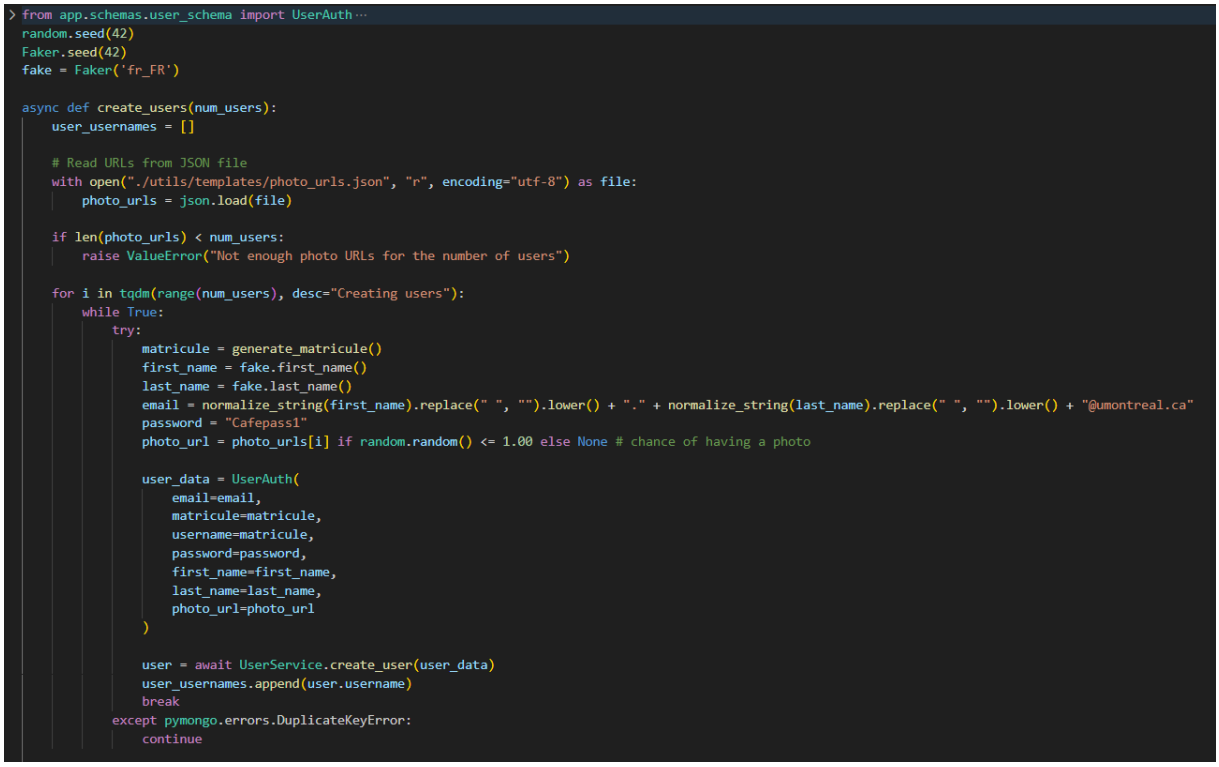
Figure 2 - Couverture des tests unitaires de l'API avec pytest

Générateur de données

Le script de génération de données, intégré dans le package *utils* de notre projet, joue un rôle essentiel dans le développement de l'application. Il remplit **deux fonctions** : (1) peupler la BD **pour les démonstrations**, créant un environnement riche et varié pour les utilisateurs potentiels ; (2) fournir des données **pour les tests unitaires**. Ces fonctions clés garantissent que nos tests couvrent un large éventail de scénarios, assurant ainsi le bon fonctionnement de l'API dans des conditions variées et proches de l'utilisation réelle.

Une caractéristique notable du script est son utilisation de la bibliothèque **Faker**[9] pour générer des données réalistes telles que des noms, des adresses courriels et d'autres informations utilisateur. Ces données, produites de manière aléatoire mais plausible, contribuent à créer un environnement de test authentique.

[9] https://faker.readthedocs.io/en/master/



```python
from app.schemas.user_schema import UserAuth···
random.seed(42)
Faker.seed(42)
fake = Faker('fr_FR')

async def create_users(num_users):
    user_usernames = []

    # Read URLs from JSON file
    with open("./utils/templates/photo_urls.json", "r", encoding="utf-8") as file:
        photo_urls = json.load(file)

    if len(photo_urls) < num_users:
        raise ValueError("Not enough photo URLs for the number of users")

    for i in tqdm(range(num_users), desc="Creating users"):
        while True:
            try:
                matricule = generate_matricule()
                first_name = fake.first_name()
                last_name = fake.last_name()
                email = normalize_string(first_name).replace(" ", "").lower() + "." + normalize_string(last_name).replace(" ", "").lower() + "@umontreal.ca"
                password = "Cafepass1"
                photo_url = photo_urls[i] if random.random() <= 1.00 else None # chance of having a photo

                user_data = UserAuth(
                    email=email,
                    matricule=matricule,
                    username=matricule,
                    password=password,
                    first_name=first_name,
                    last_name=last_name,
                    photo_url=photo_url
                )

                user = await UserService.create_user(user_data)
                user_usernames.append(user.username)
                break
            except pymongo.errors.DuplicateKeyError:
                continue
```

*Figure 3 - Fonction **create_users** dans **generate_user.py***

Le script de génération de données utilise également des informations provenant du dossier *templates* pour créer une représentation fidèle des cafés de l'UdeM :

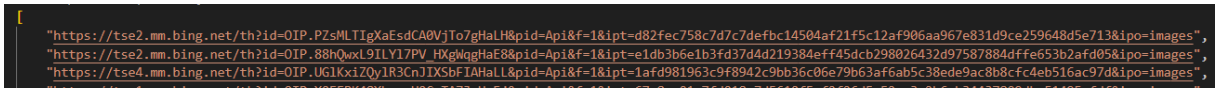
- *cafes_updated.json* : Ce fichier modèle comprend des informations authentiques sur les emplacements, pavillons, facultés, descriptions, contacts sociaux et e-mails des cafés de l'UdeM. Les images sont les seules données directement collectées à partir des réseaux sociaux des cafés, telles que Facebook. Pour assurer la pérennité des images, nous avons opté pour leur téléchargement sur **Cloudflare**. Cette étape garantit que les liens vers les images ne seront pas sujets à expiration et resteront accessibles pour les démonstrations et les tests de l'application. Les horaires sont générés de manière fictive pour répondre aux exigences des scénarios de démonstration et de test de l'application.
- *menu_items.json* : En s'inspirant du menu du café Cafcom, nous avons peuplé ce modèle avec des descriptions et des images créées manuellement, ainsi qu'avec des options de menu détaillées, pour enrichir l'expérience utilisateur avec des données de menu crédibles.
- *photo_urls.json* : Les URLs pour les photos de profil des utilisateurs sont obtenues via un script de web scraping distinct qui extrait des images de recherche DuckDuckGo, ajoutant ainsi à la diversité et au réalisme des profils utilisateurs dans notre application.

```
[
  {
    "name": "Acquis de droit",
    "description": "Sandwichs maisons pr\u00e9par\u00e9s sur place. Succulents bagels au saumon fum\u00e9. De nombreux choix v\u00e9g\u00e9. Carte de fid\u00e9lit\u00e9 disponible. Baby-foot. Ouvert l'\u00e9t\u00e9.",
    "image_url": "https://imagedelivery.net/70kpvhLybTnH06xFu8o2DQ/92b3119c-ada7-44c8-f8c3-72a40f867300/public",
    "faculty": "DROIT",
    "location": {
      "pavillon": "Pavillon Maximilien-Caron",
      "local": "local A-2470"
    },
    "contact": {
      "email": "jeremy.dinh@droit.coop",
      "phone_number": "(514)-998-9008",
      "website": "https://www.droit.coop/fr/nav/Cafe.html"
    },
    "social_media": [
      {
        "platform_name": "Facebook",
        "link": "https://www.facebook.com/cafe.acquis.de.droit"
      },
      {
        "platform_name": "Instagram",
        "link": "https://www.instagram.com/cafeacquis/?hl=fr"
      }
    ]
  },
```


*Figure 4 - **cafes_updated.json** dans le dossier **templates***

```
[
  {
    "name": "LE MOZZA",
    "tags": [
      "Fromage fondu",
      "Simple"
    ],
    "description": "Un grilled-cheese au fromage mozzarella",
    "image_url": "https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Fcdn.apartmenttherapy.info%2Fimage%2Ffetch%2Ff_auto%2Cq_auto%3Aeco%2Fhttps%3A%2F%2Fstorage.googleapis.com%2Fgen-atmedia%2F3%2F2018%2F04%2F649c960cb7640b740141678e4095261149e60693.jpeg&f=1&nofb=1&ipt=7cef0ce8e609c288f783b480e2cca2f67eab08617c8474db6eb34882be3db0c8&ipo=images",
    "price": 2.0,
    "in_stock": true,
    "category": "Grilled Cheese",
    "options": [
      {
        "type": "extra",
        "value": "tomate",
        "fee": 0.25
      }
    ]
  },
```


*Figure 5 - **menu_items.json** dans le dossier **templates***



```
[
  "https://tse2.mm.bing.net/th?id=OIP.PZsMLTIgXaEsdCA0VjTo7gHaLH&pid=Api&f=1&ipt=d82fec758c7d7c7defbc14504af21f5c12af906aa967e831d9ce259648d5e713&ipo=images",
  "https://tse2.mm.bing.net/th?id=OIP.88hQwxL9ILY17PV_HXgWqgHaE8&pid=Api&f=1&ipt=e1db3b6e1b3fd37d4d219384eff45dcb298026432d97587884dffe653b2afd05&ipo=images",
  "https://tse4.mm.bing.net/th?id=OIP.UGlKxiZQylR3CnJIXSbFIAHaLL&pid=Api&f=1&ipt=1afd981963c9f8942c9bb36c06e79b63af6ab5c38ede9ac8b8cfc4eb516ac97d&ipo=images",
```


*Figure 6 - **photo_urls.json** dans le dossier **templates***

Déploiement

Pour réaliser nos tests et recueillir efficacement les retours des utilisateurs, nous avons choisi **Render** pour le déploiement de notre application Café sans-fil.

Avec la récente évolution des services d'hébergement comme Heroku, qui ne propose plus de plans gratuits, nous avons trouvé dans Render une alternative gratuite idéale. Cette gratuité a été un facteur déterminant, en ligne avec les contraintes budgétaires de notre projet en phase de développement et de test. Elle nous a permis d'accéder à des ressources d'hébergement de qualité sans frais supplémentaires.

Intégration et déploiement continu

Render offre des capacités avancées d'intégration et de déploiement continu, qui se sont avérées essentielles pour le déploiement fluide et la maintenance régulière de notre application. Cela nous a permis de mettre à jour notre application sans interruption, intégrant efficacement les dernières améliorations et retours d'utilisateurs. La configuration des variables d'environnement sur Render a été un aspect crucial. Cela comprenait la définition des clés JWT, les configurations CORS, et les informations de connexion à MongoDB Atlas.

Utilisation de la branche preview

Nous hébergeons spécifiquement la branche ***preview*** sur Render. Cela nous permet de tester les nouvelles fonctionnalités et mises à jour avant leur déploiement. L'hébergement de cette branche preview est crucial pour nous assurer de la fonctionnalité et de la stabilité des mises à jour avant de les partager plus largement. Render est utilisé pour héberger à la fois notre service web (API back-end) et notre site statique (front-end).

Difficultés rencontrées

Pour contrer le comportement de spin-down des instances gratuites sur Render, où les services peuvent passer en mode veille après une période d'inactivité, nous avons mis en place un cron job externe. Ce cron job envoie une requête à notre API toutes les 10 minutes pour garder l'instance active. Ce service ping l'URL https://cafesansfil-api.onrender.com/api/health, assurant ainsi une disponibilité constante et une expérience utilisateur sans interruption. Vous pouvez consulter l'état de notre cron job à https://v4szkwlx.status.cron-job.org. Cette méthode nous permet de bénéficier pleinement des 750 heures gratuites par mois offertes par Render, garantissant ainsi que notre API fonctionne en continu sans interruption. Cette stratégie assure que notre API reste accessible et réactive pour les utilisateurs, tout en maximisant les avantages des ressources gratuites fournies par Render.

Documentation

Comme mentionné dans les besoins du projet, la documentation joue un rôle crucial dans la compréhension, l'utilisation et la maintenance de l'application. Pour documenter le projet, nous avons créé un wiki[10] (référence officielle du projet) tenu à jour tout au long du projet et disponible sur le répertoire GitHub du projet. Il contient également, sous forme de fichier README, des instructions détaillées sur le déploiement et la maintenance de l'application sur Render ainsi que les procédures (guides) d'installation et de d'exécution pour le *front-end* et le *back-end*.

La documentation de l'API est générée à partir de Swagger UI et Redoc. La documentation fournie par Swagger UI[11] permet aux utilisateurs de visualiser et d'interagir avec notre API. Elle offre la possibilité de tester directement les routes de l'API, facilitant ainsi la compréhension de ses fonctionnalités. La documentation fournie par Redoc[12] présente une version plus épurée de la documentation de l'API, offrant une lisibilité améliorée pour une consultation rapide.

[10] Wiki du projet : https://github.com/ceduni/cafe-sans-fil/wiki

[11] Documentation API (Swagger) : https://cafesansfil-api.onrender.com/docs

[12] Documentaiont API (Redoc) : https://cafesansfil-api.onrender.com/redoc

Conclusion

Le projet Café sans-fil vise à améliorer la qualité de la vie étudiante en offrant une plateforme commune et conviviale pour les cafés étudiants. Grâce à une analyse approfondie des besoins, une conception robuste et une documentation complète, l'application se positionne comme un outil de choix pour améliorer l'expérience des étudiants dans leur quotidien universitaire.

Ce projet nous a permis de découvrir la gestion d'un projet du début à la fin avec des échéances, un suivi, du travail d'équipe et des choix importants à faire, comme en entreprise. Nous avons expérimenté et appris énormément lors de ce projet sur les méthodes d'organisation, manier les divers outils tels que Git, GitHub et ses fonctions tel que *Issues*, *Actions* et *Projects*. De plus, nous avons perfectionné significativement nos compétences de développement en front-end et en back-end.

Dans l'optique de livrer une première version (finale) de l'application, nous devrions améliorer l'infrastructure de l'application web afin de faciliter la maintenance et l'ajout de fonctionnalités comme le multilingue. Nous devrions aussi procéder à des tests d'utilisabilité et d'acceptation afin de valider le comportement et design de l'application.

Finalement, nous avons aussi identifié quelques axes à explorer pour tout développement futur. Actuellement, l'actualisation des états de commandes nécessite un rafraichissement manuellement la page. Un système push pourrait améliorer significativement l'expérience utilisateur. L'un des risques identifiés dans notre analyse est la réduction de socialisation chez les étudiants. L'ajout de canaux de communication entre café et étudiant permettant en autre l'organisation d'évènements pourrait remédier à cela.

Remerciements

Nous tenons à remercier Louis-Édouard LAFONTANT qui a supervisé le projet, nous a grandement aidé tout le long et a suivi l'évolution de très près. Ce projet nous donne un bagage très solide qui sera valorisé en entreprise, dans la recherche d'un emploi dans le futur.

Un grand merci également à Jérémy, responsable du café Tore et Fraction, qui nous a accordé son temps afin de nous aider à cernes les exigences du projet. Merci aussi aux autres responsables de cafés qui nous ont donné un retour et des suggestions sur l'outil lors de son développement. Nous sommes reconnaissants pour le soutien et l'enthousiasme de chacun, et nous sommes impatients de voir comment Café sans-fil contribuera positivement à la vie de campus à l'Université de Montréal.

Bibliographie

Code With Prince. (2022, Mai 18). *Python API Development With FastAPI - Comprehensive Course for Beginners P-5 Password reset*. Récupéré sur YouTube: https://www.youtube.com/watch?v=Y7FCJF48Obk&list=PLU7aW4OZeUzxL1wZVOS31LfbB1VNL3MeX&index=7&ab_channel=CodeWithPrince

abdadeel. (2022, Mai 4). *The ultimate FARM stack Todo app with JWT PART I - FastAPI + MongoDB*. Récupéré sur YouTube: https://www.youtube.com/watch?v=G8MsHbCzyZ4&ab_channel=ABDLogs

Aota, T. (2018, Septembre 30). *Creating a Pipfile for multiple versions of Python*. Récupéré sur dev.to: https://dev.to/tomoyukiaota/creating-a-pipfile-for-multiple-versions-of-python-9f2

Beanie. (s.d.). *Beanie Documentation*. Récupéré sur Beanie: https://beanie-odm.dev/api-documentation/query/

Colvin, T. D. (2023, Juin 30). *Pydantic V2 Is Here!* Récupéré sur Pydantic: https://docs.pydantic.dev/dev/blog/pydantic-v2-final/

FastAPI. (s.d.). *FastAPI Documentation*. Récupéré sur FastAPI: https://fastapi.tiangolo.com/learn/

freeCodeCamp.org. (2021, Juillet 15). *FARM Stack Course - FastAPI, React, MongoDB*. Récupéré sur YouTube: https://www.youtube.com/watch?v=OzUzrs8uJl8&list=PLAt-l74BsucNBwFANkqwisPMSLE62rKG_&index=2&t=2912s&ab_channel=freeCodeCamp.org

LogRocket. (2022, Mai 10). *Complete guide to authentication with React Router v6* . Récupéré sur LogRocket: https://blog.logrocket.com/complete-guide-authentication-with-react-router-v6/

LogRocket. (2023, Mai 23). *Building reusable React components using Tailwind CSS*. Récupéré sur LogRocket: https://blog.logrocket.com/building-reusable-react-components-using-tailwind-css/

Maurya, A. (2023, Juin 12). *Handling JWT Access and Refresh Token using Axios in React App*. Récupéré sur Anish's Talk: https://blog.theashishmaurya.me/handling-jwt-access-and-refresh-token-using-axios-in-react-app

Media, L. (2022, Août 4). *Small and reusable Tailwind components with React* . Récupéré sur Lucky Media: https://www.luckymedia.dev/blog/small-and-reusable-tailwind-components-with-react

Render. (2023, Mars 24). *Mail server on render.com doesn't permit SMTP*. Récupéré sur Render: https://community.render.com/t/mail-server-on-render-com/10529

Render. (2023). *Spin-down behavior of free instance types*. Récupéré sur Render: https://community.render.com/t/requests-to-back-end-take-long/10059

Render. (s.d.). *Redirects and Rewrites*. Récupéré sur Render: https://render.com/docs/redirects-rewrites

Special Coder. (2022, Décembre 17). *command not found: pipenv Resolved*. Récupéré sur YouTube: https://www.youtube.com/watch?v=Bzn_MZ0tNXU&ab_channel=SpecialCoder

Stack Overflow. (2023, Février 3). *Cron Job fix for Spin-down*. Récupéré sur Stack Overflow: https://stackoverflow.com/questions/75340700/prevent-render-server-from-sleeping

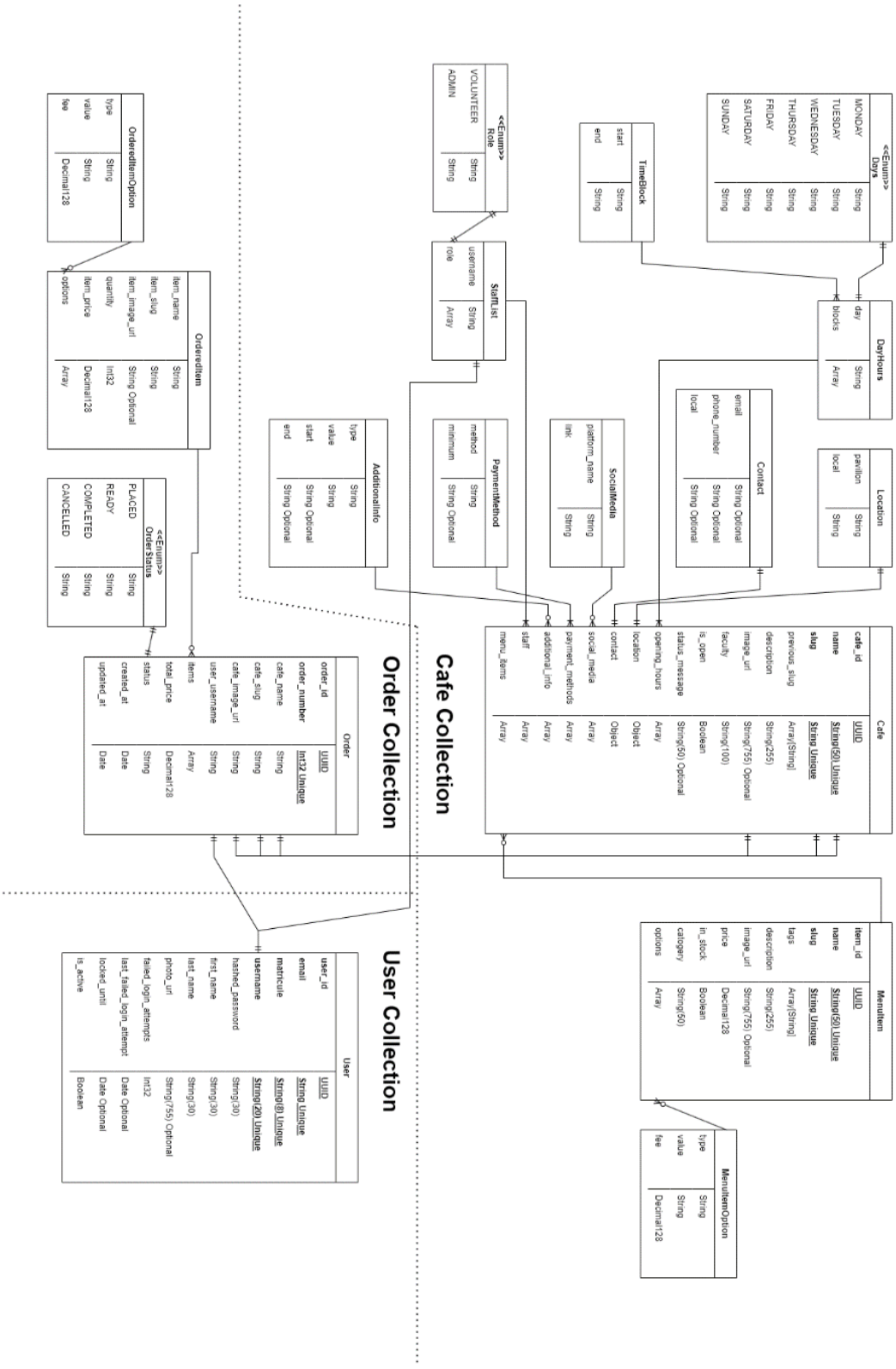
Stevens, A. (2019, Juin 29). *useApi React Hook*. Récupéré sur Andrew's Dev SIte: https://andrewstevens.dev/posts/useApi-react-hook/

TailwindCSS. (s.d.). *Tailwind Documentation*. Récupéré sur TailwindCSS: https://tailwindcss.com/docs

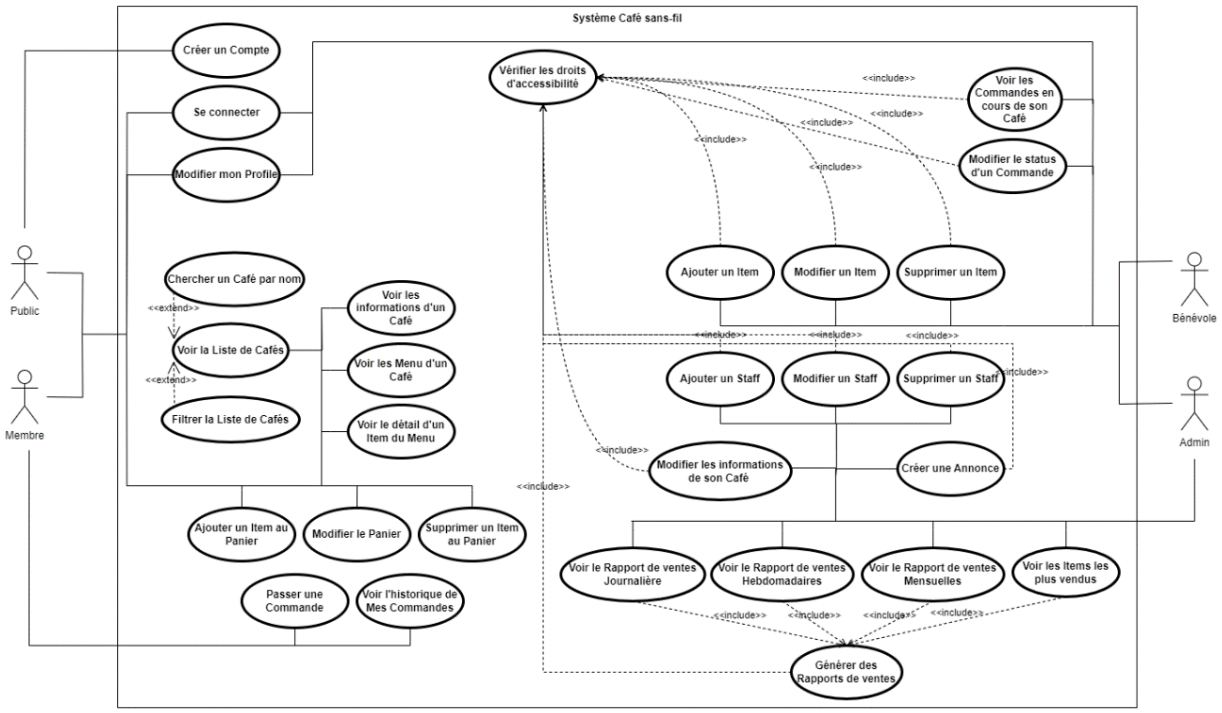
Wieruch, R. (2022, Janvier 25). *React Router 6: Authentication*. Récupéré sur RWieruch: https://www.robinwieruch.de/react-router-authentication/

Annexes

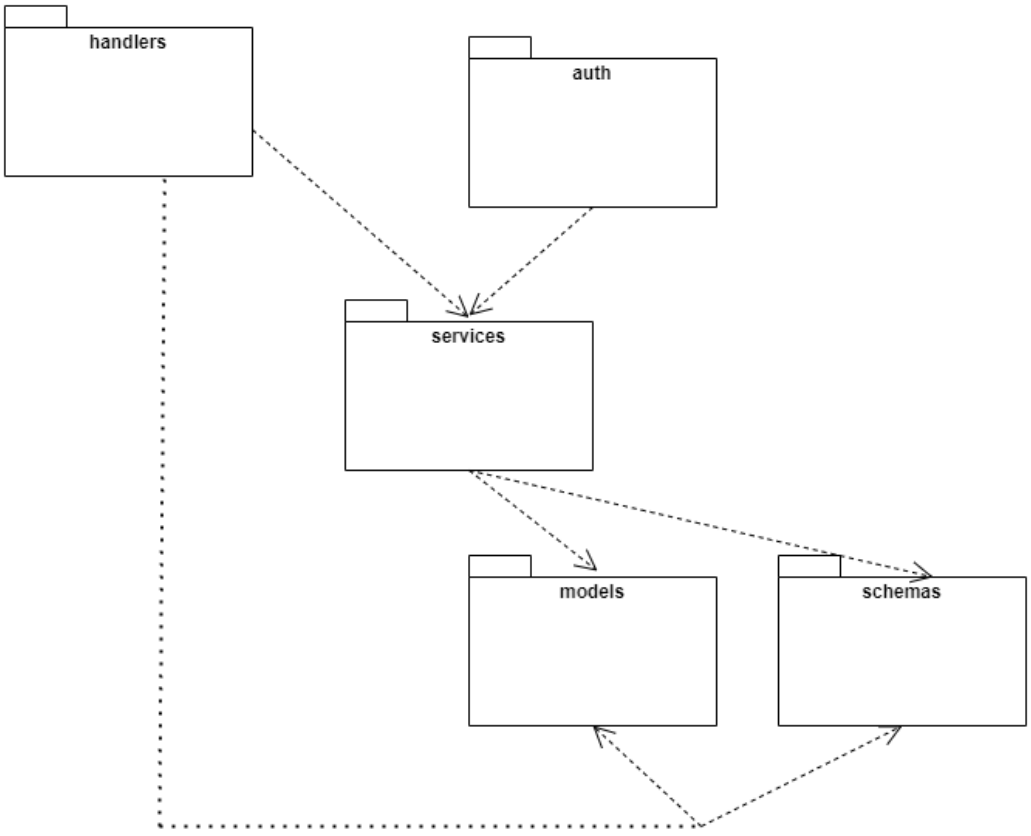
Annexe 1: Schéma de la base de données



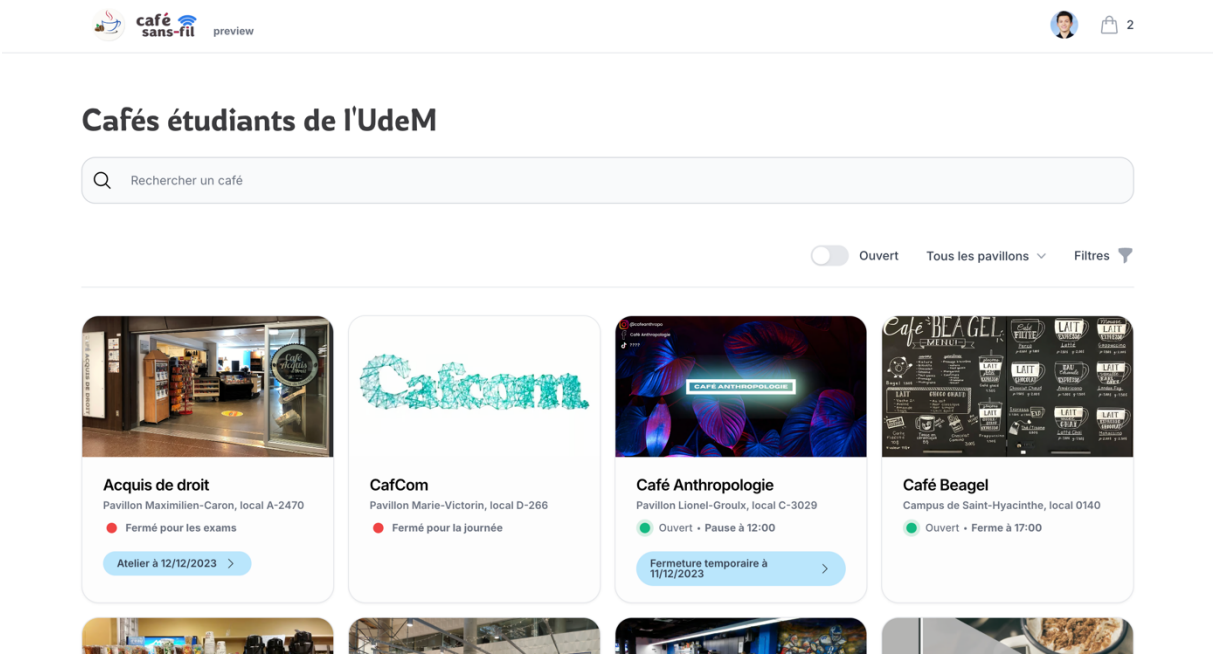
Annexe 2: Cas d'utilisations

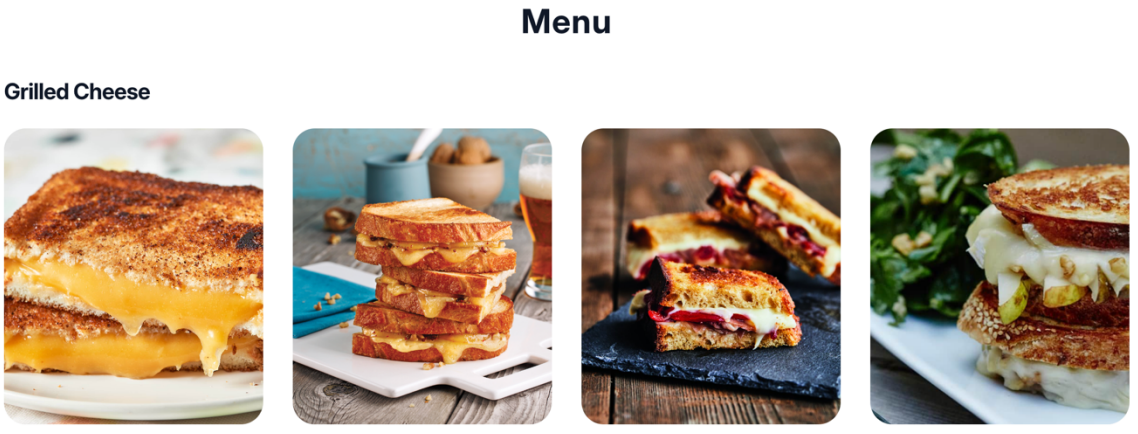


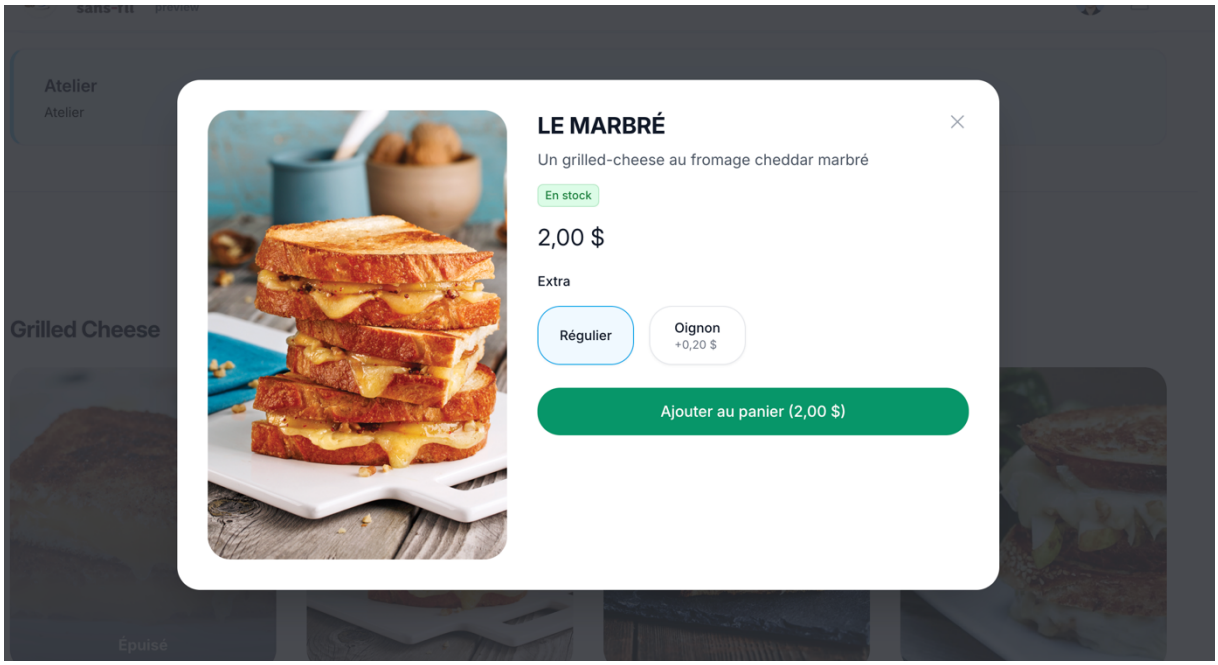
Annexe 3: Diagramme de paquets

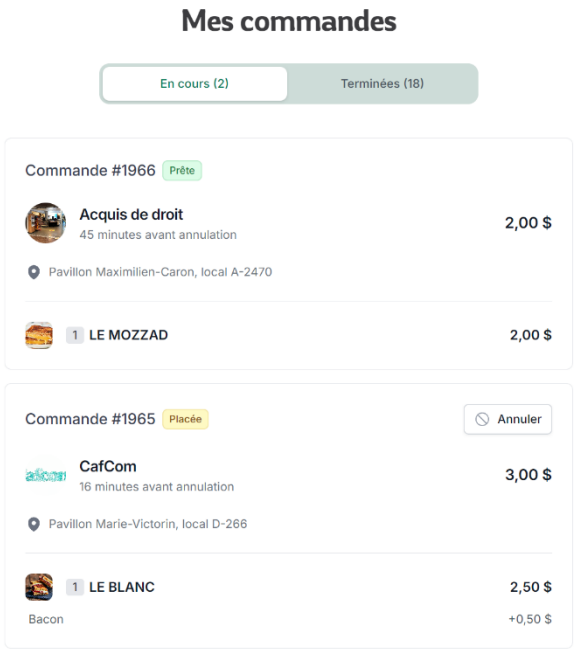


Annexe 4 : Captures d'écran de l'application






Atelier
Atelier
Grilled Cheese
Épuisé
LE MARBRÉ
Un grilled-cheese au fromage cheddar marbré
En stock
2,00 $
Extra
Régulier
Oignon
+0,20 $
Ajouter au panier (2,00 $)


Mes commandes
En cours (2)
Terminées (18)
Commande #1966
Prête
Acquis de droit
45 minutes avant annulation
2,00 $
Pavillon Maximilien-Caron, local A-2470
1 LE MOZZAD
2,00 $
Commande #1965
Placée
Annuler
CafCom
16 minutes avant annulation
3,00 $
Pavillon Marie-Victorin, local D-266
1 LE BLANC
2,50 $
Bacon
+0,50 $