Dossier Projet

Par **Ledez Grégory**

Concepteur développeur d’applications

Entreprise d'accueil	**Lost Mechanics**
Projets réalisés	**Olim, Nuclear-Blast**
Centre de formation:	**Wild Code School**

Abstract

During my studies, I had the opportunity to work on a project that involved design, UI/UX, front-end, and back-end, where I also delved into database modeling and authentication. This project is called Olim, aiming to provide users with a platform to buy or sell high-quality vintage furniture. We chose ReactJS for the front-end and utilized NodeJs with Express.Js for the back-end to establish our server. Developing a secure REST API that communicates with a MySQL database was a key aspect. To ensure the application's robustness, a series of tests are executed using Cypress and Jest during the build process, facilitating ongoing maintenance.

Furthermore, I had the privilege of joining a company through an internship, allowing me to deepen my knowledge and, more importantly, learn to collaborate within a team alongside project managers, designers, and devOps professionals. I contributed to the front-end of a project and participated in its deployment to the company's staging environment. Taking charge of creating schedules and user journeys, I also played a role in the thorough validation process, reviewing and approving each pull request with my internship supervisor before merging into the development branch.

Présentation de l’entreprise

Lost Mechanics

En Avril dernier, dans le cadre de mon alternance, j’ai eu la chance de rejoindre l’entreprise Lost Mechanics.

Cette agence de communication spécialisée dans le web, propose quasiment uniquement des **sites expérientiels**, parfois en 3D, parfois sous forme d’**escape game**, ou de **serious games**. Parmi nos clients, on compte notamment **AirBus**, **Ricard**, **MSI** ou encore l’**École de l’air et de l'espace**.

Mon rythme d’alternance est de **3 semaines en entreprise** et **une semaine** au centre de formation (Wild Code School).

L’entreprise compte 11 employé.es :

- Un **CEO**: Nicolas
- **Equipe technique** :
    - Un **lead développeur** / développeur front-end (et aussi tuteur) : Valentin.
    - Un **développeur 3D**: Sylvain
    - Un **développeur backend / devOps** : Cédric
    - Une **développeur** en alternance : moi-même
- **Gestion de projet :**
    - Une **chargé de production** : Aurélie
    - Deux **cheffes de projet** : Aurore et Camille
- **Equipe créative**
    - Un **directeur artistique** : Clément
    - Une **designer 3D**: Élodie
    - Une **webdesigneuse** en alternance : Leïly

Mon rôle dans l’équipe technique consiste à développer la partie **front-end** sous la direction du lead développeur, en suivant le planning pré établie et en s'appuyant sur les maquettes graphiques désignées par l’équipe créative. De nombreux échanges sont nécessaires avec tous les acteurs qui s'articulent autour

du projet, afin de livrer nos projets dans le temps imparti, avec un standard qualité très élevée.

Pour réaliser ces missions, je suis en constant échange avec l'équipe créative afin de décider ensemble les animations ou les micro interactions à mettre en place. Durant la phase de recette, nous échangeons très attentivement aussi avec les client.es afin de satisfaire leurs exigences.

Partie 1

1. Présentation du Projet OLIM

Ce projet nous a été proposé par notre formateur lors de mon passage à la Wild Code School. La consigne était simple, développer une application, de A à Z, qui met en relation des vendeurs/acheteurs. À la manière du Bon Coin. Ayant une forte appétence pour les interfaces web simples, et bien désignée, mais aussi pour les beaux objets, nous avons choisi avec ma camarade Ophélie Bausse de designer une interface aux formes et couleurs modernes. L'expérience utilisateur était au cœur de nos discussions tout au long du processus de développement.

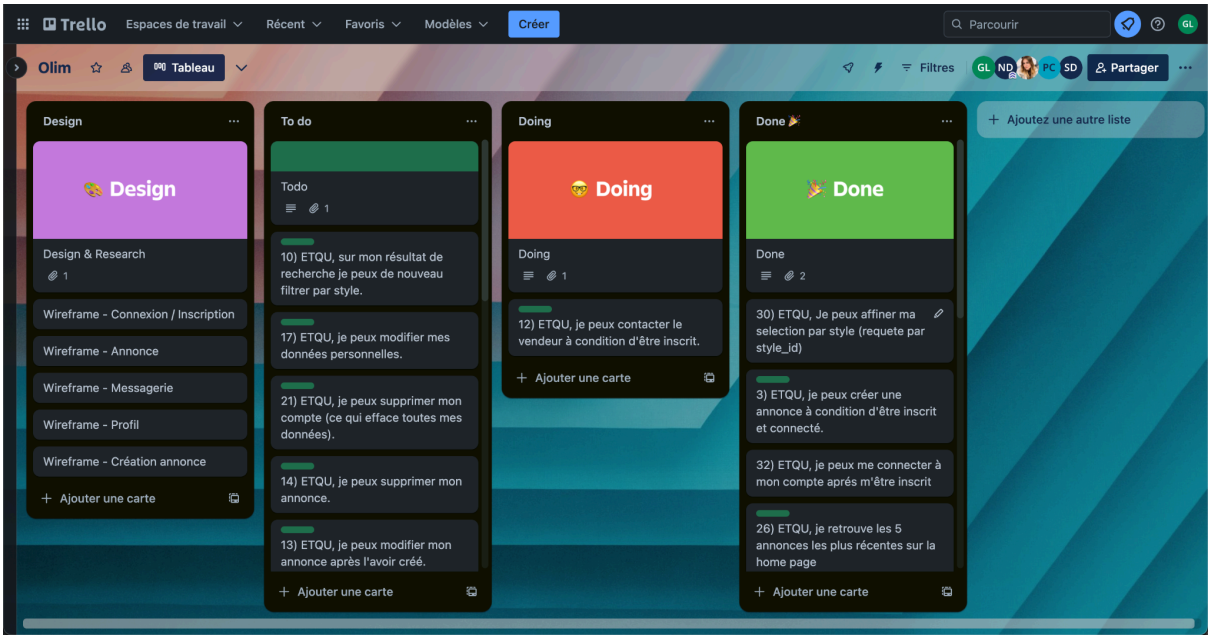
L'application doit permettre aux utilisateurs d'acheter ou de vendre des meubles de hautes qualités, vintages ou de grands designers. Ils doivent pouvoir, entre autres, créer une annonce, ajouter une annonce en favoris, supprimer une annonce et également entrer en contact avec les vendeurs. Les utilisateurs doivent aussi pouvoir rechercher facilement un produit, par couleurs, styles ou encore matiére.

Tout cela, de manière fluide et intuitive, en respectant l'accessibilité et les standards de sécurité.

1.1 Organisation du projet

Pour l'organisation du projet, nous avons choisi une méthode adaptée au projet. C'est-à-dire une méthode AGILE: nous avons listé les fonctionnalités dans un product backlog, les avons découpées en petite taches dans un board sur Trello, puis nous nous sommes répartis les tâches en fonction des envies et du niveau de maîtrise des participants, en choisissant les tâches nécessaire à la livraison d'un MVP (minimum viable project).

Voici quelques taches et les différentes listes d'actions de notre projet sur trello:



1.2 Workflow

Pour travailler à plusieurs sur un projet, nous avons versioné notre code. Pour cela nous avons utilisé Git en local et GitHub en ligne. Voici les étapes classique du workflow:

1. **Clone du repo fournis par l'école (template)** : Utilisation de git clone pour obtenir une copie locale du référentiel sur notre machine.
2. **Création d'une branche dédiée** : Avant de commencer à travailler sur une nouvelle fonctionnalité ou correction, nous devons créer une branche avec le nom de la fonctionnalité: `git checkout -b feature/loginForm`
3. **Travailler sur la branche dédiée** : Effectuer les modifications, ajouts ou corrections sur la branche créée.
4. **Commit régulièrement** : Faire des commits réguliers avec des messages de commit clairs et concis pour documenter vos changements:

   git add .

   `git commit -m 'created the login form component'`
5. **Push sur la branche distante** : Push de la branche sur le repo distant (GitHub) avec `git push origin nom-de-la-branche`.

6. **Création d'une Pull Request (PR)** : Une fois la tâche terminée, il faut créer une PR pour fusionner les modifications dans la branche principale. Il est généralement recommandé de ne pas pusher les features directement sur la branche 'main', surtout pour des fonctionnalités en cours de développement. Au lieu de cela, il est préférable de travailler sur une branche de développement, telle que **develop.** La pull request permet la revue du code et les discussions.

7. **Révision / Discussion** : Le formateur et deux membres de l'équipe devaient valider les PR.

8. **Merge de la PR** : Une fois que la PR est approuvée, voici venue la fusion de la branche de la feature, dans la branche develop, directement depuis l'interface de Github.

9. **Suppression de la branche** : Après la fusion, on peut supprimer la branche dédiée localement et distante.

2. Compétences du référentiel couvertes

Ce projet couvre les compétences suivantes :

- Maquetter une application
- Développer des composants d'accès aux données
- Développer la partie front-end d'une interface utilisateur web
- Développer la partie back-end d'une interface utilisateur web
- Concevoir la sécurité dans chaques couches
- Concevoir une base de données
- Mettre en place une base de données
- Développer des composants dans le langage d'une base de données de développement
- Concevoir une application
- Développer des composants métier
- Construire une application organisée en couches
- Développer une application mobile
- Préparer et exécuter les plans de tests d'une application

3. Les fonctionnalités

Pour définir les fonctionnalités d'Olim, et répondre au cahier des charges précis fourni par notre formateur, nous avons tout d'abord fait un brainstorming. Nous avons posé sur la table les fonctionnalités que nous recherchons sur un site de petites annonces, ce que nous n'aimons pas également. Nous avons échangé avec des utilisateurs sur ces points, puis nous avons défini les besoins spécifiques, les fonctionnalités prioritaires, et les fonctionnalités secondaires.

Nous avons pour cela écris des user stories, en tant que 3 utilisateurs/utilisatrices distinct:

- *Utilisateur inscrit et connecté*
- *Utilisateur non inscrit ou non connecté*
- *Admin*

Les user stories suivent un schémas simple:

"En tant qu'utilisateur je veux [faire quelque chose] afin de [objectif]."

Par exemple : "**En tant que vendeur, je veux poster une annonce depuis mon canapé, afin de le vendre.**"

Voici les fonctionnalitées prioritaires retenue:

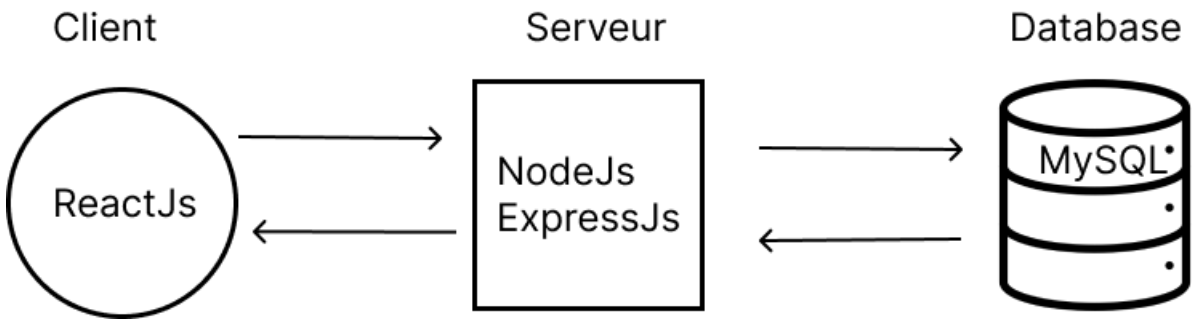
- Un **Utilisateur inscrit et connecté** peut:
    - Poster une annonce ou des annonces
    - Modifier tout ou partie de son ou ses annonces
    - Supprimer son ou ses annonces
    - Voir son ou ses annonces
    - Consulter des annonces
    - Rechercher des annonces
    - Ajouter une photo de profil
    - Modifier ces données (mot de passe, adresse, ville...)
    - Réinitialiser son mot de passe
    - Entrer en contact avec de potentiels acheteurs
    - Entrer en contact avec de potentiels vendeurs
- Un **Utilisateur non inscrit ou non connecté** peut:
    - Consulter des annonces
    - Rechercher des annonces
- Un **Utilisateur connecté avec le rôle d'admin** peut:
    - Consulter toutes les annonces
    - Consulter la liste des utilisateurs/utilisatrices

- Supprimer une annonce si elle ne respecte pas les conditions d'utilisation
- Supprimer un utilisateur
- Attribuer le rôle admin à un autre utilisateur

4. Spécifications techniques

Pour développer cette application j'ai participé à toutes les étapes de conceptions, du choix des technologies, à l'architecture. Pour la partie application Mobile et la base de données, j'ai travaillé seul.

Voici l'architecture globale de l'application:



4.1 Client

ReactJs

Pour mettre en place la partie front-end de notre application, nous avons opté pour React JS, une bibliothèque développée par Facebook en 2013. Ce choix s'est justifié par la modularité et la légèreté de ReactJS, qui permet aux développeurs de sélectionner les modules nécessaires en fonction de leurs besoins. Par exemple, l'utilisation de React Router se fait uniquement si le développeur nécessite un système de navigation entre les composants, évitant ainsi le surchargement d'éléments inutiles lors du build final, contrairement à certains frameworks. Par ailleurs, l'architecture d'une application React en tant que Single Page Application offre une expérience utilisateur rapide. Le système de composants permet un affichage efficace des interfaces, où seul le composant mis à jour est affiché sans nécessiter le chargement de la page. Pour la partie mobile de l'application, l'utilisation de React Native a été privilégiée en raison de sa similarité avec React, simplifiant ainsi le développement multiplateforme.

Scss

SCSS, une extension syntaxique pour CSS, offre des fonctionnalités avancées telles que les variables et les mixins et l'imbrication. Ce qui facilite la gestion des règles CSS spécifiques à un élément particulier, contribuant ainsi à un développement plus efficace et structuré. Cependant, pour être interprété par les navigateurs, le code SCSS doit être compilé en CSS standard. Ce processus, géré par des outils tels que Sass, génère un fichier CSS résultant. La spécificité de SCSS réside dans sa capacité à simplifier l'écriture des styles tout en nécessitant une étape de compilation pour produire un code CSS compréhensible par les navigateurs web.

Ky

Le choix de la bibliothèque JavaScript 'Ky', qui est une alternative légère et performante à Axios, est une bibliothèque pour effectuer des requêtes HTTP. Elle offre une syntaxe concise et expressive, simplifiant la gestion des appels réseau. Ky prend en charge les fonctionnalités modernes comme les promesses, la gestion des erreurs, et permet de configurer facilement les options de requête. Grâce à sa simplicité et à ses fonctionnalités avancées, Ky est un choix plus adapté que d’autres bibliothèques.

4.2 Serveur

NodeJs

Pour le back-end, nous avons choisi Node.js, qui est un environnement d'exécution JavaScript côté serveur basé sur le moteur V8 de Google Chrome. Il est asynchrone, non bloquant et événementiel, ce qui lui permet de gérer efficacement des opérations simultanées. En utilisant JavaScript, il offre un langage unifié pour le développement côté serveur et côté client. Il est idéal pour les applications en temps réel grâce à son modèle asynchrone. Il est cross-platform, compatible avec divers systèmes d'exploitation, et commence à être largement utilisé dans le développement web pour créer des serveurs et des applications côté serveur.

ExpressJs

Pour faciliter la création du serveur, nous avons opté pour ExpressJs. Express.js est un framework web pour Node.js. Il facilite la création d'applications web. En fournissant des fonctionnalités essentielles, telles que la gestion des routes, des middlewares, et des requêtes HTTP, Express simplifie le processus de création

d'applications serveur. Sa nature légère et modulaire en fait un choix parfait pour le développement de notre API RESTful.

Voici un exemple de création de serveur Express:

index.js

```
require('dotenv').config();

const app = require('./src/app');

const port = parseInt(process.env.PORT ?? '8001', 10);

app.listen(port, (err) => {
  if (err) {
    console.error('Server failed to start:', err);
  } else {
    console.log(`Server is listening on http://localhost:${port}`);
  }
});
```

MySQL

Pour la gestion des données, nous avons utilisé MySQL en tant que système de gestion de base de données relationnelle.

MySQL offre une performance élevée, une stabilité et une compatibilité étendue, en faisant un choix solide pour stocker et organiser les informations de notre application.

Communication avec MySQL

Pour établir la communication entre le serveur Node et MySQL, nous avons intégré le module mysql2. Ce module fournit une interface Node pour interagir avec MySQL, facilitant ainsi les opérations telles que l'exécution de requêtes SQL et la gestion des résultats. Cette combinaison de technologies nous a permis de mettre en place un backend performant et fiable, capable de gérer efficacement les requêtes de l'application frontend tout en assurant une gestion optimale des données via MySQL.

4.4 Sécurité

La sécurité d'une application revêt une importance cruciale afin de préserver l'intégrité des données qui y circulent. À chaque étape du processus de développement, le développeur doit mettre en œuvre les meilleures pratiques de sécurité pour prévenir les principales vulnérabilités, notamment les injections SQL, les attaques XSS, le vol de cookies, et d'autres risques potentiels. Cette approche proactive garantit un environnement robuste et résilient face aux menaces de sécurité.

JWT

Pour gérer la sécurisation du processus d'authentification au sein de l'application, nous avons utilisé des Json WebToken.

argon2

Dans une application sécurisée, le hachage de mot de passe est nécessaire avant d'être stocké en base de données. Pour cela nous avons fait appel à argon2.

Argon2 est une fonction de dérivation de clefs et un algorithme de hachage.

Point sécurité

Le hachage de mot de passe est un processus de transformation d'un mot de passe en une séquence aléatoire de caractères, généralement une chaîne de longueur fixe, à l'aide d'une fonction de hachage. Cette séquence résultante est souvent appelée le "haché" ou "hash". Le processus de hachage de mot de passe est conçu pour être irréversible, ce qui signifie qu'il doit être difficile ou impossible de remonter du haché au mot de passe d'origine. Cela ajoute une couche de sécurité importante dans le stockage des mots de passe, en particulier dans les bases de données. Les caractéristiques clés du hachage de mot de passe comprennent :

- **Irréversibilité : Il ne doit pas être possible de déterminer le mot de passe d'origine à partir du haché.**
- **Résistance aux collisions : Deux mots de passe différents ne doivent pas produire le même haché.**

- **Déterminisme : Le même mot de passe doit toujours produire le même haché.**
- **Résistance aux attaques : La fonction de hachage doit être conçue pour résister aux attaques par force brute (tenter de découvrir un mot de passe ou une clé secrète en essayant systématiquement toutes les combinaisons possibles jusqu'à ce qu'ils trouvent la bonne.), par dictionnaire (deviner un mot de passe en essayant une liste de mots prédéfinis. Contrairement à une attaque par force brute, qui teste toutes les combinaisons possibles de caractères, une attaque par dictionnaire cible des mots de passe plus courants ou prévisibles.), et aux autres méthodes utilisées par les pirates. En utilisant le hachage de mot de passe, même si la base de données est compromise, les mots de passe réels restent difficiles à récupérer, renforçant ainsi la sécurité des comptes utilisateur.**

Voici la fonction utilisé pour hacher un mot de passe:

hashPassword.js

```
const argon = require('argon2');

const hashOptions = {
  // Type d'algorithme de hachage argon2id, considéré comme le plus sécurisé
  type: argon.argon2id,
  // Coût en mémoire (2^16 kilobytes)
  memoryCost: 2 ** 16,
  // Coût temporel, le nombre d'itérations effectuées lors du hachage
  timeCost: 5,
  // Parallélisme, spécifie le nombre de threads pour le traitement parallèle
  parallelism: 1,
};

const passwordHash = async (password) => {
  const hashedPassword = await argon.hash(password, hashOptions);

  return hashedPassword;
};

module.exports = passwordHash;
```

4.5 Tests

Cypress

Pour les tests e2e, j'ai utilisé Cypress. Cypress est un framework de test d'interface utilisateur qui est principalement utilisé pour les tests end-to-end et les tests d'intégration. Il est spécialement conçu pour tester des applications web modernes.

Jest

Jest est un framework de test JavaScript qui est utilisé pour écrire des tests unitaires et d'intégration. Jest facilite l'écriture de tests unitaires en fournissant une syntaxe simple et en intégrant des fonctionnalités telles que l'exécution de tests en parallèle, l'observation des fichiers pour les changements, etc.

4.6 Variables d'environnement

Dans mon projet, j'ai opté pour l'utilisation de variables d'environnement pour gérer de manière sécurisée et flexible des configurations. Cela me permet de stocker des informations sensibles, telles que des clés d'API ou des identifiants de base de données, en dehors du code source, renforçant ainsi la sécurité. Les variables d'environnement offrent également la possibilité d'ajuster facilement la configuration de l'application sans nécessiter de modifications directes dans le code, facilitant ainsi le déploiement et garantissant une portabilité accrue entre différents environnements. En résumé, elles contribuent à une gestion plus efficace des paramètres d'application et à une conformité aux meilleures pratiques de sécurité.

Le fichier de variables d'environnement, .env, est ensuite ajouté dans le fichier .gitignore. Son fichier "jumeau", qui lui contient des fausses valeurs, est présent et envoyé sur github.

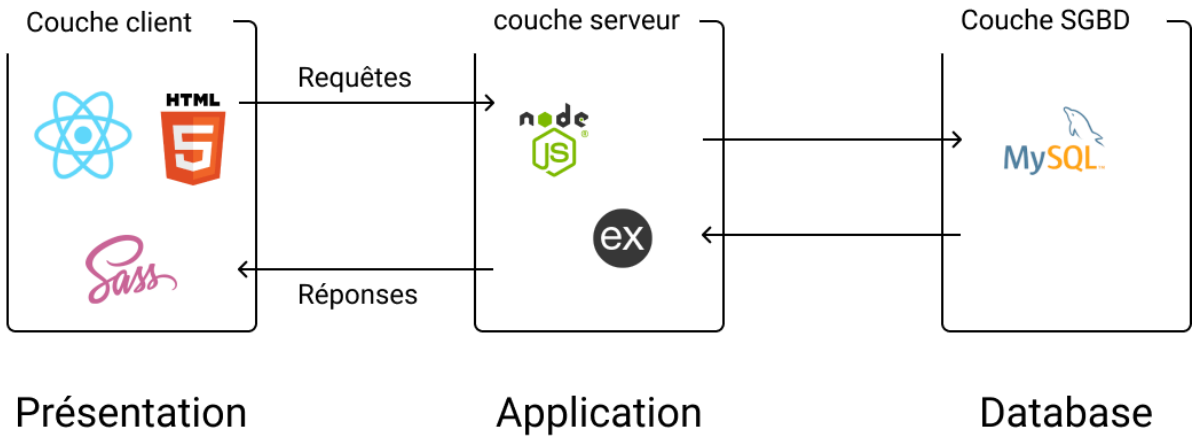
5. Application multi-couches

Une architecture de projet bien conçue est cruciale pour la gestion efficace d'un projet. Elle prévient la duplication de code, renforce la stabilité et facilite l'évolution de l'application en cas de besoin. Par exemple, en adoptant une architecture à trois couches, une équipe de développeurs peut effectuer des modifications liées à la base de données sans altérer les autres couches du projet.

5.1 Architecture 3 couches.

Pour Olim, nous avons adopté l'architecture 3 couches:

- Couche de **Présentation**: Ce sont les composants de l'interface utilisateur : pages HTML, CSS, JavaScript, etc. Bibliothèques ou frameworks frontend tels que React, Angular, ou Vue.js. Elle gère notamment l'affichage et la saisie.
- Couche de **Logique Métier** : Ici ce sont les composants de logique métier qui traitent les règles applicatives. Fonctions ou classes qui valident les données des tâches, appliquent des règles spécifiques, préparent les données à stocker etc. Indépendant de la technologie de présentation.
- Couche de **Persistance des Données** : Base de données (par exemple, MySQL, MongoDB) pour stocker les informations sur les tâches. Composants ou modules d'accès aux données qui communiquent avec la base de données pour insérer, mettre à jour, récupérer et supprimer des tâches. Indépendant de la technologie de présentation et de la logique métier. En adoptant cette architecture multicouche, on peut facilement changer la technologie de présentation (par exemple, passer de React à Vue.js) sans affecter la logique métier ou la persistance des données. De même, on peut changer la base de données sans toucher à la logique métier ou à l'interface utilisateur.



6. Base de données

La conception d'une base de données dans une application web est une étape cruciale qui intervient généralement dans la phase initiale du développement, lors de la planification et de la conception de l'architecture globale de l'application.

6.1 Conception de la BDD

6.1.1 Modélisation du MCD

MCD signifie Modèle Conceptuel des Données. Le MCD est souvent utilisé dans la phase de conception d'une base de données pour décrire de manière abstraite la structure des données et les interactions entre elles. Il peut être représenté sous la forme de diagrammes entité-association qui mettent en évidence les entités, les relations et les cardinalités. Ce modèle conceptuel fournit une vue claire et compréhensible des données avant leur implémentation concrète dans une base de données. Nous avons donc commencé par écrire les règles de gestion, afin de faire ressortir les besoins de notre application. Par exemple:

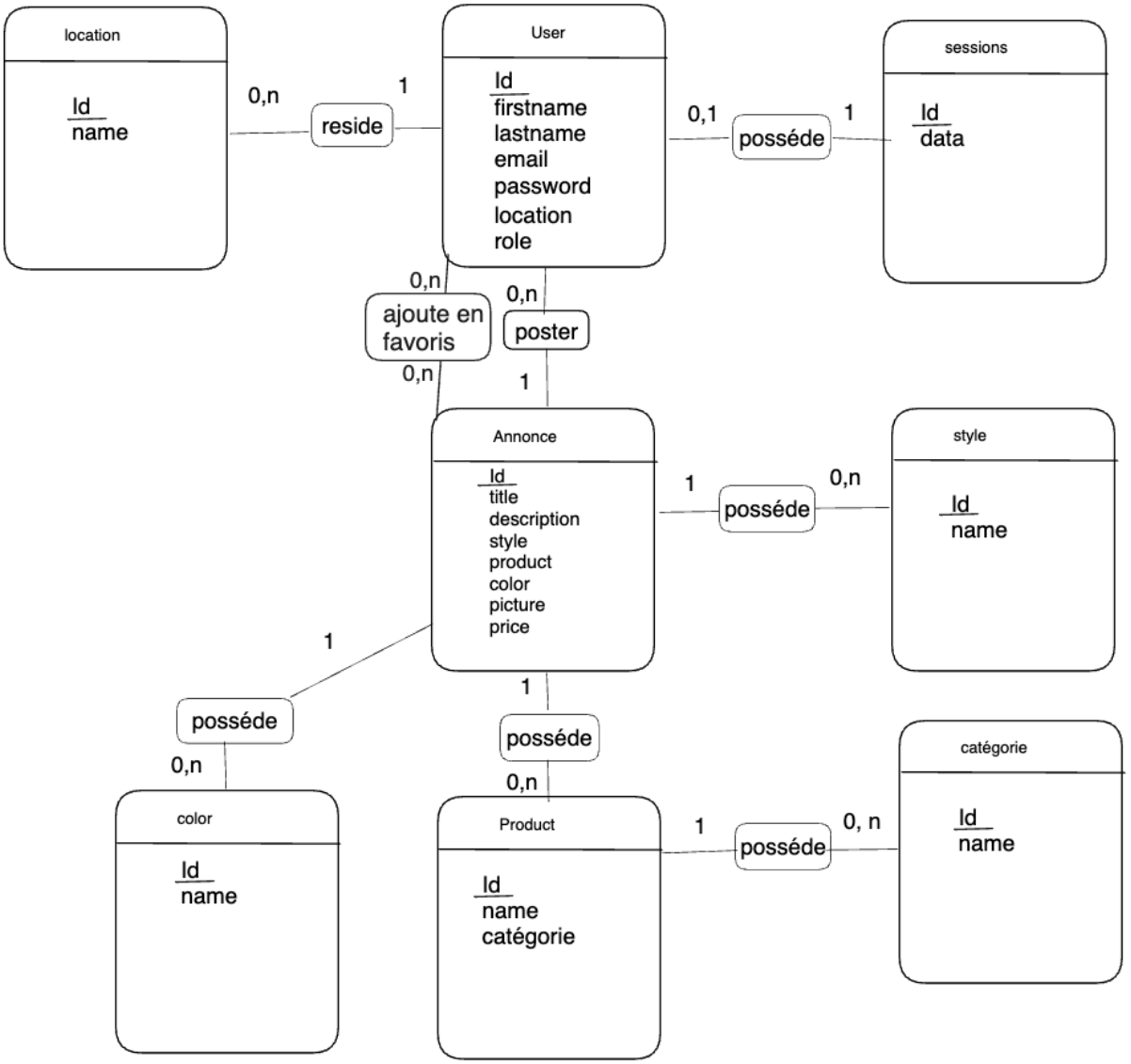
- Un utilisateur peut poster une annonce.
- Un utilisateur peut se connecter grâce à un id de session.
- Une annonce doit être postée par un utilisateur.
- Une annonce doit contenir un produit.
- Un produit doit avoir une catégorie.
- Une annonce doit avoir une couleur.
- Une annonce doit avoir un style.
- Un utilisateur doit être rattaché à un lieu.

À partir de ces quelques règles, nous avons pu définir 8 entités, leurs propriétés (la liste de données de l’entité), les relations qui expliquent et précisent comment les entités sont reliées entre elles et leurs cardinalités.

Lorsque l'on conçoit une base de données avec le MCD de Merise, on obtient un schéma avec des entités et des associations. Pour préciser au mieux les associations, on utilise des cardinalités. Quand par exemple on veut définir combien de fois une entité peut appartenir à une association. Ce sont des caractères (0,1, n) qui fonctionnent par couple et qui sont présents de chaque côté d'une association

Exemple: Un utilisateur peut poster 0 ou plusieurs annonces, et une annonce ne peut être posté que par un seul utilisateur. La cardinalité est donc de n,1.

Voici le MCD de l'application Olim:



On peut voir ci-dessus la cardinalité entre **User** et **Annonce**. Un utilisateur peut poster 0 ou un nombre infini d'annonces, et une annonce doit être postée par un seul utilisateur minimum et maximum (elle ne peut pas être orpheline, ni être postée par deux utilisateurs). La relation dans le sens Annonce > User est de 1,1 (remplacé dans le schéma par 1) et dans le sens User > Annonce est de 0,n . Nous allons donc retenir seulement les cardinalités maximales, ce qui nous donne **1,n**. Nous avons là une relation many-to-one.

Les types de relations

Il existe trois types de relations:

- **many-to-one** (1,n): C'est notre exemple du dessus. Une annonce peut être postée par **un seul** utilisateur, mais un utilisateur peut poster **plusieurs** annonces.
- **one-to-one** (1-1): C'est une relation unique entre deux entités. Par exemple, un citoyen français à **une seule** carte vitale, et une carte vitale peut appartenir à **un seul** citoyen.
- **many-to-many** (n,m): C'est la relation la plus difficile. Une entité peut interagir avec plusieurs éléments d'une autre entité, et vice versa. Par exemple, pour implémenter la feature qui permet d'ajouter une annonce en favoris. Un utilisateur pourrait avoir **plusieurs** favoris, et une annonce pourrait être dans les favoris de **plusieurs** utilisateurs. On aurait donc une cardinalité n,n mais on écrit n,m.

6.1.2 MLD

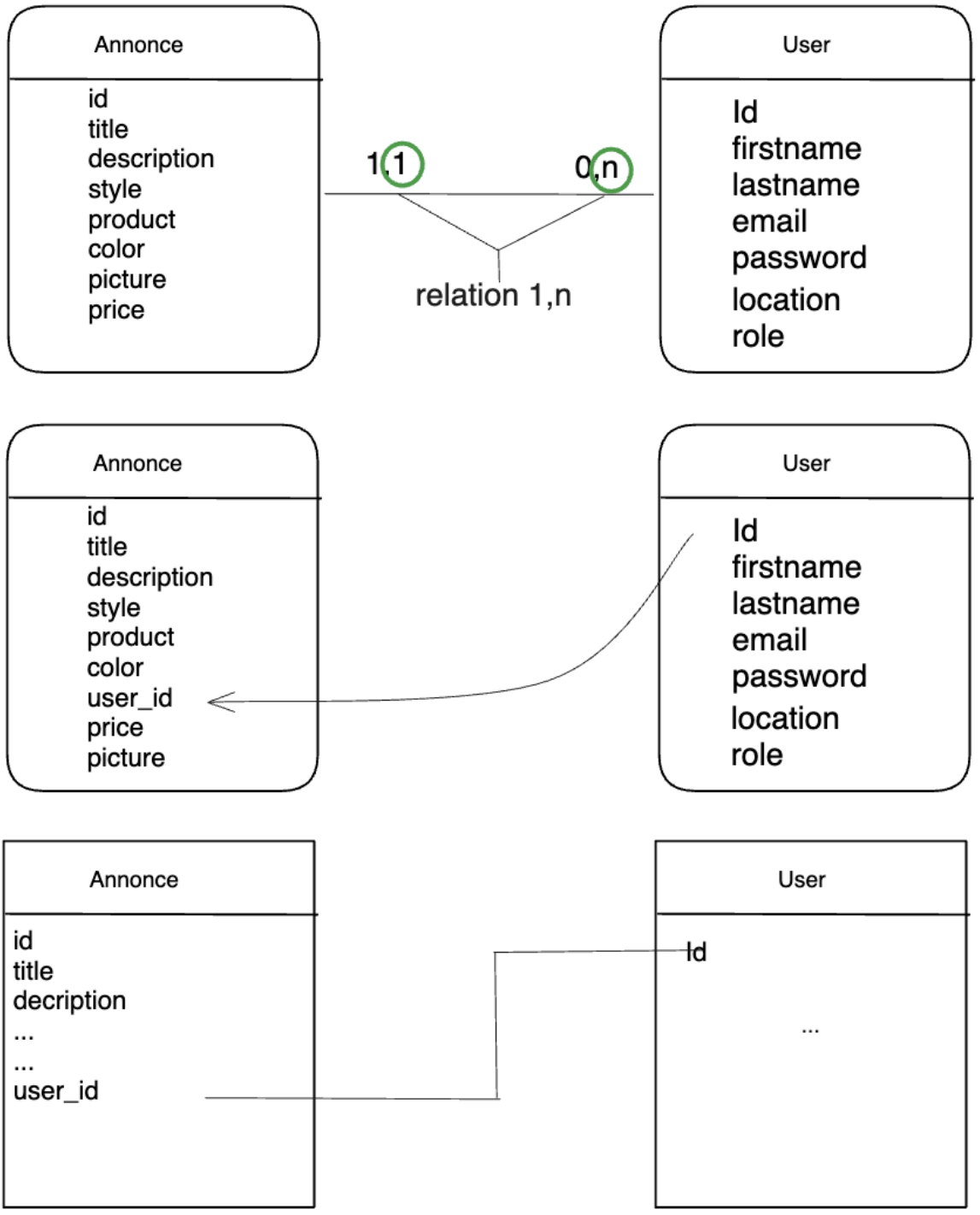
En utilisant le MCD comme point de départ, nous progressons vers le Modèle Logique de Données, une étape plus concrète dans la définition de notre base de données. Dans cette phase, chaque entité du MCD se transforme en une table, où l'identifiant de chaque entité devient la clé primaire de la table correspondante. Conformément à notre schéma, on peut voir qu'il n'est pas nécessaire de créer des tables de jonction, des relations directes entre les tables sont établies grâce à l'utilisation des clés étrangères.

"*Concrètement, le MLD permet de connaître le nombre de tables ainsi que leurs contraintes (liaisons entre tables) à mettre en œuvre dans une base de données relationnelle.*" base-de-donnes.com

Reprenons nos entités Annonce et User. Le **1** de la relation 1,n est du côté **Annonce** tandis que le **n** est coté **User**. Ce sens est très important, car c'est

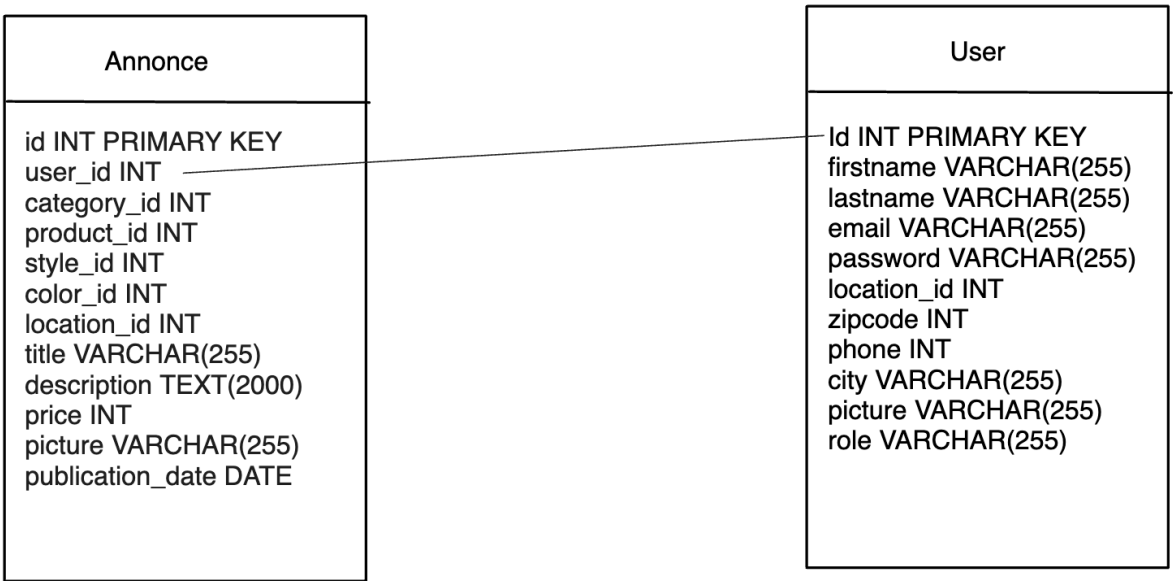
maintenant dans la table Annonce que nous allons venir ajouter un nouveau champ, qui aura pour but de relier les deux tables.

On appellera ce champ **‘user_id’**. Pour chaque tuple, le but de ce champ sera de prendre la valeur de la **clé primaire** de la table User, correspondant au User qui poste l’annonce.



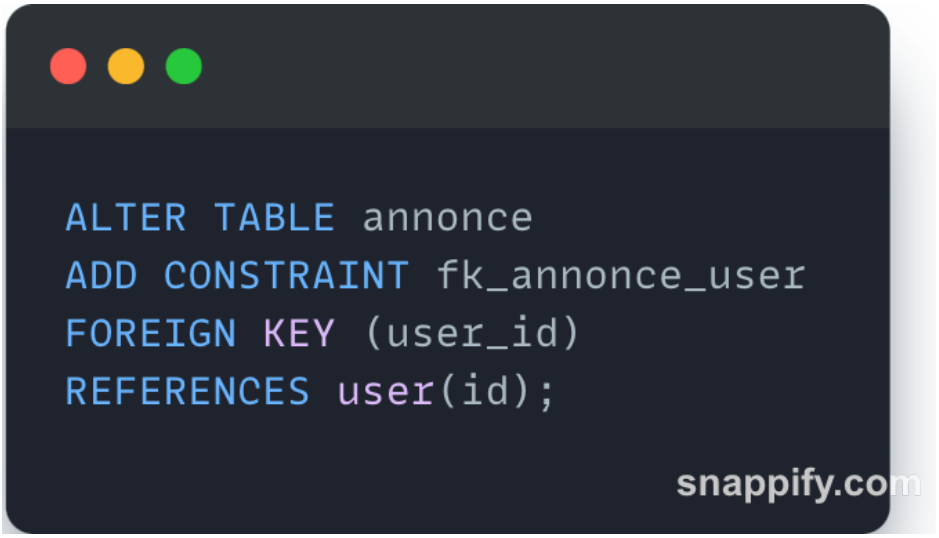
6.1.3 MPD

Ensuite nous ajoutons le type de données attendues pour chaque tuple, ce qui nous donne le MPD:



Clé étrangère

La clé étrangère sert à empêcher une insertion dans une table ou suppression dans une table (l'id d'un utilisateur dans la table Annonce par exemple), si la valeur de user_id n'existe pas. Elle est donc **indispensable** pour conserver des données saines et cohérentes. Par exemple, si on tente de supprimer un utilisateur alors que son Id est présent dans une annonce, les données de la table annonces seront incohérentes, car aucun utilisateur n'y sera rattaché.



6.1.4 Migrations

Pour créer les tables et leurs relations entre elles nous nous sommes partagé le travail et avons chacuns créer des tables et leurs contraintes, “à la main” dans un souci d’apprentissage. Nous avons tout réuni, et organisé pour respecter un ordre de création.

Nous avons découvert plus tard l'outil MySQL Workbench qui propose une interface pour créer les tables et les remplir par la suite.

Voici un extrait du script avec la création de la table Annonce, récupéré grâce à la commande **mysqldump -u root -p olimFinal > olim.sql**.

```
CREATE TABLE `annonce` (
  `id` int NOT NULL AUTO_INCREMENT,
  `user_id` int NOT NULL,
  `category_id` int NOT NULL,
  `product_id` int NOT NULL,
  `color_id` int DEFAULT NULL,
  `location_id` int NOT NULL,
  `style_id` int DEFAULT NULL,
  `title` varchar(255) NOT NULL,
  `description` varchar(8000) NOT NULL,
  `photo` varchar(255) DEFAULT NULL,
  `price` int NOT NULL,
  `publication_date` date DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_annonce_category` (`category_id`),
  KEY `fk_product_annonce_id` (`product_id`),
  KEY `fk_annonce_color` (`color_id`),
  KEY `fk_annonce_location` (`location_id`),
  KEY `fk_annonce_style` (`style_id`),
  KEY `fk_orders_user` (`user_id`),
  CONSTRAINT `fk_annonce_category` FOREIGN KEY (`category_id`) REFERENCES `category` (`id`),
  CONSTRAINT `fk_annonce_color` FOREIGN KEY (`color_id`) REFERENCES `color` (`id`),
  CONSTRAINT `fk_annonce_location` FOREIGN KEY (`location_id`) REFERENCES `location` (`id`),
  CONSTRAINT `fk_annonce_style` FOREIGN KEY (`style_id`) REFERENCES `style` (`id`),
  CONSTRAINT `fk_annonce_user` FOREIGN KEY (`user_id`) REFERENCES `user` (`id`) ON DELETE CASCADE,
  CONSTRAINT `fk_orders_user` FOREIGN KEY (`user_id`) REFERENCES `user` (`id`) ON DELETE CASCADE,
  CONSTRAINT `fk_product_annonce_id` FOREIGN KEY (`product_id`) REFERENCES `product` (`id`)
)
```

6.2 Configuration de la base de données

Afin de connecter notre projet node à la base de données, nous devons suivre plusieurs étapes.

Creation

Lors du processus de développement, chaque développeur à créer la base de données en local. Ensuite, nous avons partagé le script et chacun l'a lancé sur sa machine. (Plus tard dans le processus, nous avons "hébergé" une base de données commune via free.db)

Connexion

La prochaine étape consiste à connecter la base de données à notre serveur. Pour cela nous avons utilisé le module mysql2, qui nous facilite la tâche. Nous avons dû renseigner nos informations de connexion mysql (stocker dans un .env)

```
const dbConfig = {
 host: process.env.DB_HOST,
 port: process.env.DB_PORT,
 user: process.env.DB_USER,
 password: process.env.DB_PASSWORD,
 database: process.env.DB_NAME,
};
```

et ensuite

```
const mysql = require("mysql2/promise");
const dbConfig = require("./dbConfig");

const database = mysql.createPool(dbConfig);

database
 .getConnection()
 .then(() => {console.log('Olim database connected')})
 .catch((err) => {
   console.error(err);
 });

module.exports = { database };
```

6.3 Les requêtes SQL

Les instructions SQL, ou requêtes SQL, sont des commandes valides interprétées par les Systèmes de Gestion de Bases de Données (SGBD). Elles nous permettent de créer, modifier, ou supprimer des tables, ainsi que d'effectuer des opérations

sur les données stockées à l'intérieur de ces tables. Dans notre projet elles sont situées dans le modèle, la couche d'accès aux données.

Insérer des données

La fonction createUser() située dans la couche d'accès aux données, nous permet d'insérer des données dans la table User grâce à la requête 'INSERT INTO'.

Elle insérera les données passées en paramètre, à savoir firstname et lastname, dans les colonnes qui sont précisées dans la requête.

JS userModel.js

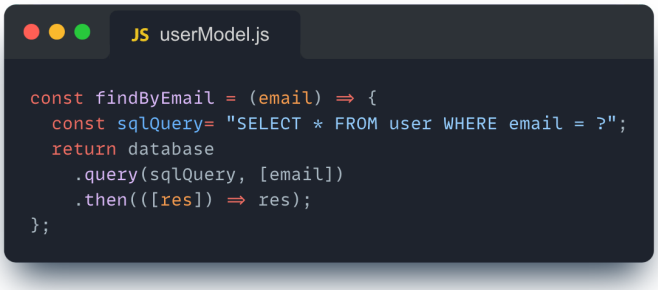
```
const createUser = (firstname, lastname) => {
  const querySql = "INSERT INTO user (firstname, lastname) VALUES (?,?);
  return database.query(querySql, [firsname,lastname]).then(([res]) => res);
};
```

Écrire tous les champs et les valeurs peut s'avérer rébarbatif. Grâce à MySQL, nous avons utilisé la syntaxe "SET". Le payload est un objet dont chaque clé est le nom d'une colonne, et la valeur associée sera insérée dans cette colonne. Le bénéfice le plus évident de cette syntaxe est qu'elle place les noms de colonnes et les valeurs d'insertion très proches l'un de l'autre. Cela élimine complètement la possibilité de fournir les noms de colonnes et les valeurs dans un ordre incorrect.

```
const createUser = (payload) => {
  const querySql = "INSERT INTO user set ?";
  return database.query(querySql, [payload]).then(([res]) => res);
};
```

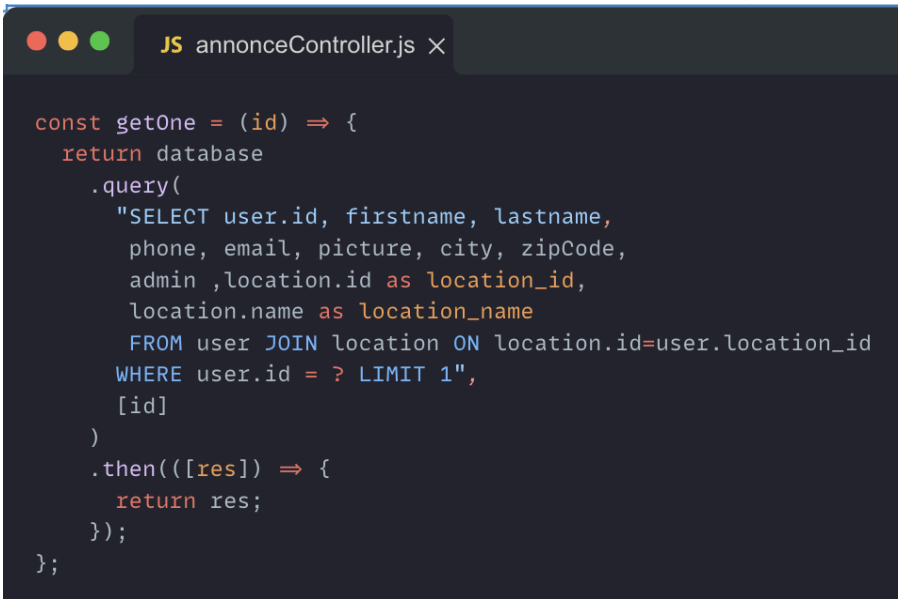
Récupérer des données

La fonction findByEmail() nous permet de récupérer les informations d'un utilisateur grâce à son email.



Agrégation de données

Lorsque l'on souhaite récupérer des données d'une autre table en fonction d'une clé d'une première table, il faut utiliser 'JOIN' en SQL. Nous avons créé des alias avec AS, afin d'éviter des erreurs d'incohérence.



7. Réalisations significatives

7.1 Authentification avec un JWT

L'authentification et l'autorisation combinent argon2 pour le hachage des mots de passe et JSON Web Token (JWT) pour la gestion des sessions utilisateur.

Voici comment fonctionne concrètement ce processus d'authentification :

Lorsqu'un utilisateur souhaite se connecter, il envoie son identifiant unique, généralement son adresse e-mail, ainsi que son mot de passe en clair au serveur. Le serveur utilise argon2 pour hasher le mot de passe reçu et le compare au mot de passe haché stocké dans la base de données. Si les deux correspondent, cela signifie que l'utilisateur a fourni les bonnes informations.

Une fois l'authentification réussie, le serveur génère un JSON Web Token (JWT) contenant des informations sur l'utilisateur, telles que son identifiant unique et ses autorisations. Ce JWT est ensuite renvoyé au client. Le client reçoit le JWT et le stocke en local, dans les cookies dans notre cas.

Lorsque le client souhaite accéder à une ressource protégée sur le serveur, il inclut le JWT dans les en-têtes de sa requête HTTP. Le serveur vérifie alors la validité du JWT et extrait les informations sur l'utilisateur pour déterminer s'il est autorisé à accéder à la ressource demandée.

En utilisant cette approche, l'utilisation du mot de passe de l'utilisateur est limitée au moment de la connexion initiale. Une fois que le JWT est généré avec succès, il est utilisé pour authentifier et vérifier les permissions de l'utilisateur lors de chaque requête, ce qui réduit le nombre d'interactions avec la base de données et améliore les performances globales du système.

Le JWT est composé de trois parties:

- **Header :** La première partie d'un JWT est l'en-tête, qui contient des métadonnées sur le type de token et l'algorithme de hachage utilisé pour signer le jeton. Par exemple, l'en-tête peut spécifier que le token est un JWT (typiquement "JWT") et qu'il est signé avec l'algorithme HMAC SHA256.

- **Payload :** La deuxième partie du JWT est la charge utile (payload), qui contient les informations que le jeton représente. Ces informations peuvent inclure des données sur l'utilisateur (ne jamais inclure des données sensibles, le payload est seulement encodé en base64url donc facilement lisible.), ainsi que d'autres métadonnées utiles pour l'application. La charge utile est encodée en base64url pour la rendre compacte et facilement transmissible sur le réseau.

- **Signature :** La troisième partie du JWT est la signature, qui est utilisée pour vérifier l'intégrité des données du jeton et garantir qu'elles n'ont pas été modifiées lors de la transmission. La signature est calculée en utilisant l'algorithme de hachage spécifié dans l'en-tête et une clé secrète connue uniquement par le serveur. La signature garantit que le JWT n'a pas été

altéré ou falsifié par des tiers lors de son transfert entre le client et le serveur.

sign()

```
const token = jwt.sign({
                id,
                firstname,
                picture,
              },
              process.env.JWT_AUTH_SECRET,
              { expiresIn: "1h" }
            );
            res.cookie("token", token, {
              httpOnly: true,
              secure: true,
              sameSite: true,
              sign: true,
            });
```

Verify()

```
const authChecker = {
  isUserLoggedIn: async (req, res, next) => {
    const { token } = req.cookies;
    if (token) {
      try {
        const user = jwt.verify(token, process.env.JWT_AUTH_SECRET);
        req.user = user;
        next();
      } catch (err) {
        res.clearCookie("token");
        next(err);
      }
    } else {
      res
        .status(401)
        .json({ message: "you need to be authenticated to acess this page" });
    }
  },
};
```

API Rest

Les normes REST dans l'API de notre projet suivent les principes de l'architecture RESTful (Representational State Transfer), qui sont des conventions et des bonnes pratiques à respecter plutôt qu'une technologie spécifique. En ce qui concerne la définition des chemins, une règle fondamentale stipule qu'il doit y avoir un "path" définissant chaque ressource. Ainsi, il existe autant de chemins que de ressources demandées. Les URL doivent suivre une logique sémantique spécifique pour permettre une compréhension immédiate du résultat de la route en lisant simplement l'URL. Par exemple, pour récupérer les annonces, le terme "annonce" est inclus dans le chemin:

```
// ALL ANNONCE
app.get('/annonce',(req,res) => {
  // ...
})

// ONCE ANNONCE
app.get('/annonce/:id',(req,res) => {
  // ...
})
```

7.2 CRUD des annonces

L'interface de programmation d'application (API), est le moyen par lequel la partie client communique avec la base de données via des requêtes HTTP. Notre API a été élaborée avec Express, et chaque route suit une structure spécifique.

Voici un exemple de requête qui renvoie ‘Hello World!’ au client.

```
const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000)
```

app.**METHOD(PATH,HANDLER)**

app est une instance de classe d'Express

METHOD est une méthode de requête HTTP, il en existe plusieurs, mais les plus utilisées sont get, post, put, delete.

PATH est le chemin de la route d'API sur lequel le ou les middleware seront invoqués.

HANDLER est la fonction (middleware) à exécuter lorsque la route est appelée. Il peut aussi s'agir de plusieurs middleware, séparés par des virgules, des tableaux de middleware ou tout en même temps. Le middleware ci-dessus contient les paramètres req, res pour Requête, Response.

req est un objet HTTP envoyé par le client. Cet objet contient des informations comme l'entête, le body (corps) de la requête avec des données saisies par un utilisateur, des paramètres passés par l'url, en général l'id de l'item que l'on veut modifier ou supprimer, des queries, les cookies...

res est aussi un objet HTTP, il est envoyé par le serveur vers le client. Il est renvoyé à la suite d'une requête HTTP. Il peut contenir des informations comme une liste d'utilisateurs, renvoyé grâce à une des nombreuses méthodes de res, res.send('hello world!'). Il peut aussi renvoyer des codes réponse. (200, 300, 400)

next est une fonction utilisée pour passer le contrôle de la requête au prochain middleware dans la pile de middlewares. Chaque middleware peut effectuer des opérations sur la requête, la réponse, ou passer la main au middleware suivant.

Router

Pour plus de visibilité et dans un soucis de clean code, nous avons utilisé le router d'express afin de créer plusieurs sous-groupes de route. Le router d'express permet d'appeler un fichier en deuxième paramètre de la méthode use(). De cette manière, nous avons créé un fichier index.js dans le dossier **router**. Dans ce dossier, nous avons aussi créé trois fichiers, pour nos routes. Un fichier *annonceRouter.js*, *userRouter.js* et *utilsRouter.js*.

JS index.js

```
const express = require("express");

const annonceRouter = require("./annonceRouter");
const userRouter = require("./userRouter");
const utilsRouter = require("./utilsRouter");

const router = express.Router();

router.use("/annonce", annonceRouter);
router.use("/user", userRouter);
router.use("/utils", utilsRouter);

module.exports = router;
```

JS app.js

```
app.use("/api", router);
```

GET

Si par exemple on envoie une requête .get sur la route “api/annonce/${id}”, la requête arrive sur ‘/api/annonce/:id’, elle utilisera annonceRouter, passera par le middleware `isUserLoggedIn` puis exécutera la méthode getOneAnnonce de l’objet annonceController.

JS index.js

```
router.use("/annonce", annonceRouter);
```

JS annonceRouter.js

```
annonceRouter.get(
  "/:id",
  authChecker.isUserLoggedIn,
  annonceController.getOneAnnonce
);
```

JS annonceController.js

```
getOneAnnonce: (req, res, next) => {
    const { id } = req.params;
    annonceModel
      .findOne(id)
      .then((annonce) => {
        if (!annonce) {
          res.status(404).send("Sorry, this adds does not exists");
        } else {
          res.send(annonce);
        }
      })
      .catch((err) => {
        next(err);
      });
  },
```

JS annonceModel.js

```
const findOne = async (id) => {
  const [res] = await database.query(
    "SELECT annonce.*, category.name as category, style.name as style,
     color.name as color, product.name as product, user.firstname as userName,
     user.picture as userPicture, user.city as userCity,
     user.email as userEmail, user.phone as userPhone
     FROM annonce
     JOIN user ON user_id = user.id
     JOIN color ON color_id = color.id
     JOIN style ON style_id = style.id
     JOIN product ON product_id= product.id
     JOIN category ON annonce.category_id = category.id
     WHERE annonce.id = ?",
    [id]
  );
  return res[0];
};
```

POST

url api: '/api/annonce/

JS annonceRouter.js

```
annonceRouter.post(
  "/",
  authChecker.isUserLoggedIn,
  upload.single("photo"),
  annonceController.createAnnonce
);
```

JS annonceController.js

```
createAnnonce: (req, res, next) => {
    const user_id = req.session.passport.user;
    const photo = req?.file?.filename;
    const payload = { user_id, ...req.body, photo };

    annonceModel
      .createOne(payload)
      .then((result) => {
        res.status(201).send(`${result.insertId}`);
      })
      .catch((err) => next(err));
  },
```

```
const createOne = async (payload) => {
  const [res] = await database.query(`INSERT INTO annonce SET ?`, [payload]);
  return res;
};
```

PUT

url api: '/api/annonce/${id}

JS annonceRouter.js

```
annonceRouter.put(
  "/:id",
  authChecker.isUserLoggedIn,
  annonceController.updateAnnonce
);
```

JS annonceController.js

```
updateAnnonce: (req, res) => {
    const { id } = req.params;
    const payload = req.body;
    annonceModel
      .updateAnnonce(payload, id)
      .then((result) => res.status(200).send(`annonce ${id} updated`))
      .catch((err) => {
        console.error(err);
        res.sendStatus(500);
      });
  },
```

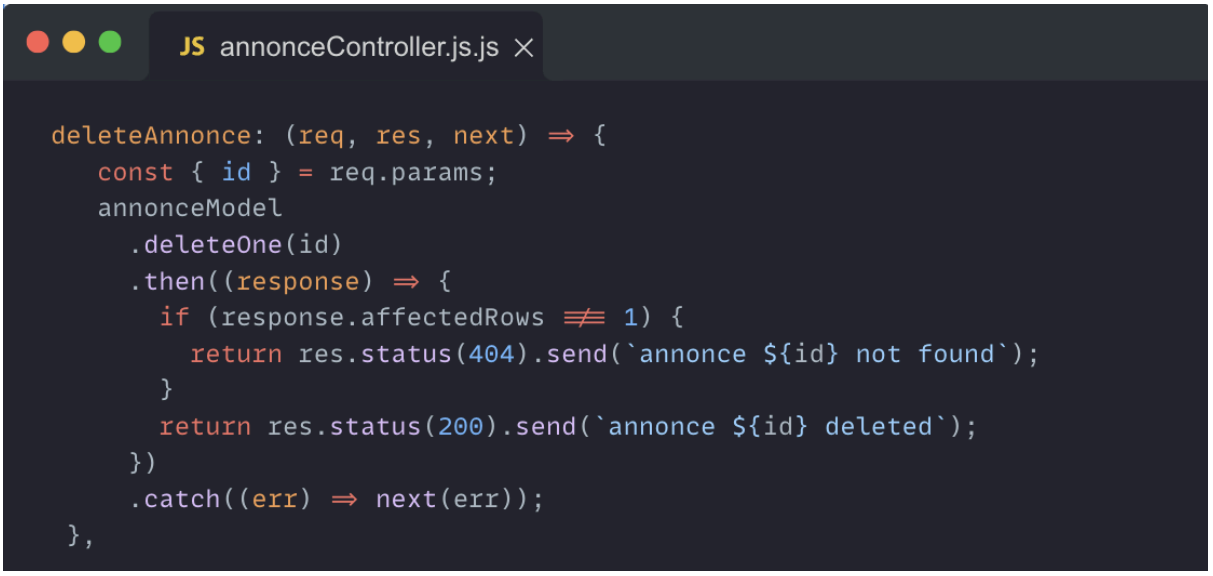
```
const updateAnnonce = async (payload, id) => {
  const [res] = await database.query("UPDATE annonce SET ? WHERE id = ?", [
    payload,
    id,
  ]);
  return res;
};
```

DELETE

url api: '/api/annonce/${id}

JS annonceRouter.js

```
annonceRouter.delete(
  "/:id",
  authChecker.isUserLoggedIn,
  annonceController.deleteFavoriteAnnonceByAnnonceId,
  annonceController.deleteAnnonce
);
```




```
const deleteOne = async (id) => {
  const [res] = await database.query("DELETE FROM annonce WHERE id = ?", [id]);
  return res;
};
```

7.3 Poster une annonce

Données en entrées, données attendues, données obtenues.

Côté client

Lorsque l'utilisateur entre ces données dans les champs de textes, la fonction

handleOnChange(e), présente dans l'attribut onChange est appelée pour mettre à jour l'état de l'objet Annonce avec les nouvelles paires de clé valeurs. e.target.name fait référence au nom de l'input qui est la clé, et e.target.value fait référence à la valeur de l'input, dans le cas des Select, l'id est ajouté à l'objet. Le state *setSelectedState* sert à afficher les produits en relation à la catégorie choisie par l'utilisateur, les produits de la catégorie choisie sont alors fétché via la méthode *getAllProductByCategoryId*(id).

Au clic sur le bouton Submit, la fonction *postAdd*() est appelée, et la requête http est alors envoyée. Cette fonction crée un objet FormData pour faciliter l'envoie de données via une requête http. La méthode .append() ajoute une clé et une valeur à l'objet. Et ensuite, la requête est appelée avec la méthode POST.

AdForm.jsx

```
const [annonce, setAnnonce] = useState({
    title: null,
    category: null,
    product: null,
    color: null,
    style: null,
    description: null,
    price: null,
  });

const handleOnChange = (e) => {
    if (e.target.name === "category") {
      getAllProductByCategoryId(e.target.value)
    }
    setAnnonce({
      ...annonce,
      [e.target.name]: e.target.value,
    });
  };
```

InputSelect.jsx

```jsx
import React from "react";

function InputSelect({
  disabled,
  name,
  placeholder,
  onChange,
  formValue,
  options,
}) {
  const handleChange = (e) => {
    onChange(e, e.target.name, e.target.value);
  };
  return (
    <div className="input-wrap">
      <select
        className="newAd__selectCategory"
        onChange={handleChange}
        name={name}
        required
        disabled={disabled}
      >
        <option value={formValue}>{placeholder}</option>
        {options &&
          options.map((option) => (
            <option key={option.id} value={option.id}>
              {option.name}
            </option>
          ))}
      </select>
    </div>
  );
}

export default InputSelect;
```

```
<form
        encType="multipart/form-data"
        className="newAd__form"
        onSubmit={postAdd}
        action=""
      >
        <div className="newAd__left">
          <div className="newAd__titleAd">
            <Input
              type="text"
              id="adTitle"
              name="title"
              placeholder="Choose a catchy title *"
              onChange={handleOnChange}
            />
          </div>

          // etc ...
          <div className="newAd__publish">
            <Button className="newAd__btn" action="" type="submit" label="publish">
              <img
                src="src/assets/icons/publish_icon.png"
                alt="publish button"
                className="newAd__publishImg"
              />
            </Button>
          </div>
        </div>
      </form>
```

Données en entrées

Pour cet exemple, l'utilisateur à posté une nouvelle annonce qui contient pour:

- Titre: 'Canapé'
- Catégorie: 'Furniture' qui a pour valeur l'id 1
- Produit: 'Sofa' qui a pour valeur l'id 7
- Couleur: 'Bleu' qui a pour valeur l'id 1
- Style: 'Classique' qui a pour valeur l'id 1
- Description: "Canapé très bon état"
- Prix: 200

À ce stade, la requête a été envoyée au serveur avec comme données en entrée un objet :

```
{
  category_id: '1',
  product_id: '7',
  color_id: '1',
  style_id: '1',
  title: 'Canapé',
  description: 'Canapé neuf',
  price: '200',
  location_id: '10'
}
```

Données attendues

Lorsqu'un utilisateur poste un annonce, on souhaite stocker cette annonce dans la base de données, on attend donc une nouvelle ligne dans la table annonce. Si on se réfère à notre MLD , notre base de données attends ceci:

```
{
  user_id: 142,
  category_id: '1',
  product_id: '7',
  color_id: '1',
  style_id: '1',
  title: 'Canapé',
  description: 'Canapé neuf',
  price: '200',
  location_id: '10',
  picture 'photo.jpg',
}
```

Côté serveur

Du côté du serveur, la requête post sur la route '/api/annonce/', exécute la méthode

createAnnonce() du annonceController et lui transmet l'objet request. Le controller exécute la logique métier avant de faire appel à la couche d'accès au données.

C'est notamment ici que l'on va récupérer l'id de l'utilisateur via la session. On va créer un objet pour l'envoyer au model.

JS annonceController.js

```
createAnnonce: (req, res, next) => {
    const user_id = req.session.passport.user;
    const photo = req?.file?.filename;
    const payload = { user_id, ...req.body, photo };

    annonceModel
      .createOne(payload)
      .then((result) => res.status(201).send(`${result.insertId}`))
      .catch((err) => next(err));
  },
```

C'est le model qui va créer l'annonce en base de données en utilisant l'objet envoyé depuis le controller.

JS annonceModel.js

```
const createOne = (payload) => {
  return database
    .query(`INSERT INTO annonce SET ?`, [payload])
    .then(([res]) => res);
};
```

L'objet retourné par le model au controller contient l'id de l'annonce qui a été créé.

Le controller va renvoyer au client: l'id créé ainsi que le code HTTP 201 grâce aux méthodes *res.status(201).send(result.insertId*).

En cas d'erreur, le *catch*() renvoie l'erreur.

Données obtenues

Côté client, si l'on console.log la réponse, pour les besoin de ce dossier, nous voyons bien l'id créé qui s'affiche. L'utilisateur est alors redirigé vers l'annonce qu'il vient de créer.

7.4 Ajouter un fichier

Sur l'application Olim, lorsqu'un utilisateur ajoute une annonce, il est invité à joindre une photo de l'objet qu'il met en vente.

Côté client

Pour gérer l'ajout de fichier nous récupérons le fichier via un input de type file et avec un attribut accept à image/*, qui signifie que le type de fichier doit être une image. Au *onChange* de l'input, on exécute la fonction *handleFile*, qui transforme le fichier reçu par l'input et met à jour l'état file pour modifier l'UI.

```
const handleFile = (e) => {
   if (!e.target.files || e.target.files.length === 0) {
     setFile(undefined);
     return;
   }

   setFile({
     fileName: e.target.files[0].name,
     file: e.target.files[0],
   });
 };

// ... //

<input
   className="newAd__btn"
   style={{ display: "none" }}
   type="file"
   name="photo"
   id="file"
   required
   accept="image/*"
   onChange={handleFile}
```

Le fichier est ensuite ajouté au corps de la requête POST pour être envoyé au serveur.

Côté serveur

Pour gérer les fichiers côté serveur, nous avons utilisé multerJs. Multer est un middleware express qui gère les requêtes avec des données de type multipart. (multipart/form-data).

Tout d'abord, il nous faut configurer multer, en lui indiquant où stocker les fichiers reçus, et aussi pour renommer les fichier:

```
const storage = multer.diskStorage({
  destination(req, file, cb) {
    cb(null, "../frontend/public/uploads"); // chemin du dossier où enregistrer les images
  },
  filename: (req, file, callBack) => {
    const ext = file.mimetype.split("/")[1]; // on réccupére ici l'extention.
    const fileName = file.originalname.substring(
      0,
      file.originalname.lastIndexOf(".") // ici le nom du fichier.
    );
    callBack(null, `${fileName}-${Date.now()}.${ext}`); // et on reconstruie le nom avec la date.
  },
});

const upload = multer({ storage });
```

Et il nous reste plus qu'à l'utiliser dans la route en s'assurant de passer la clé de l'objet envoyé depuis le client, ici 'photo':

```
annonceRouter.post(
  "/",
  authChecker.isUserLoggedIn,
  upload.single("photo"),
  annonceController.createAnnonce
);
```

Le fichier se trouvant dans req.file, et non dans req.body, nous devons recréer un payload avant de l'envoyer vers la couche qui va stocker notre annonce en base de données:

```
createAnnonce: (req, res, next) => {
    const user_id = req.session.passport.user;
    const photo = req?.file?.filename;
    const payload = { user_id, ...req.body, photo };

    annonceModel
      .createOne(payload)
      .then((result) => {
        res.status(201).send(`${result.insertId}`);
      })
      .catch((err) => next(err));
  },
```

7.5 Filtrer des annonces

En Express.js, une query fait référence aux paramètres d'une requête HTTP qui sont généralement inclus dans l'URL après le point d'interrogation, tels que ?nom='toto'. Ces données peuvent être récupérées côté serveur via req.query et utilisées pour personnaliser la logique de la route. Les queries express sont envoyées dans l'url côté client, et récupérées côté serveur grâce à **req.query** qui transforme '?nom="toto" en {nom: 'toto'}.

Pour les annonces, l'utilisateur à la possibilité de filtrer les annonces, en fonction de plusieurs propriétés. (Couleur, style, prix...). Nous avons passé les filtres de l'utilisateur dans une query.

Côté client, nous construisions la query au fur et à mesure que l'utilisateur coche les items.

Voici la construction de la query côté client:

```
useEffect(() => {
  if (filter) {
    const query = [];
    for (const key in filter) {
      if (filter[key]) {
        query.push(`${[key]}=${filter[key]}`);
      }
    }

    const temp = [];
    query.map((filtre, index) => {
      if (!query.includes("?") && index === 0) {
        temp.push(`?${filtre}`);
      } else {
        temp.push(`&${filtre}`);
      }
      return temp;
    });
    setQueryFilter(temp.join(""));
  }
}, [filter]);
```

La requête est ensuite envoyé:

```
api.get(`${url}/annonce/${queryFilter}`)
```

Puis est récupéré côté serveur grâce à **req.query**.

```
findAll: (req, res, next) => {
    const where = [];
    if (req.query.product_id) {
      where.push({
        column: "annonce.product_id",
        value: req.query.product_id,
        operator: "=",
      });
    }

    if (req.query.category_id) {
      where.push({
        column: "annonce.category_id",
        value: req.query.category_id,
        operator: "=",
      });
    }
    // ... etc
     annonceModel
      .findAll(where)
      .then((result) => res.send(result))
      .catch((err) => next(err));
    }
```

Dans cet exemple, on construit un tableau où **where** est destiné à être utilisé comme filtre lors de la recherche d'annonces. Pour chaque paramètre de requête possible (product_id, category_id), il vérifie s'il est présent dans la requête. Si c'est le cas, une condition est ajoutée au tableau **where** spécifiant la colonne de la base de données (column), la valeur à rechercher (value), et l'opérateur à utiliser (operator). Le tableau where est ensuite passé en paramètre de la fonction *findAll*(). Pour être ensuite transformé en requête sql:

JS annonceModel.js

```
const findAll = (where) => {
  const initialSql = `SELECT annonce.*, user.firstname as userName, user.city as userCity,
    user.email as userEmail, category.name as category, product.name as product,
    location.name as region,
    color.name as color, style.name as style FROM annonce
    JOIN category ON category.id=category_id
    JOIN product ON product.id=product_id
    JOIN location ON location.id=location_id
    JOIN color ON color.id=color_id
    JOIN style ON style.id=style_id
    JOIN user ON user.id=user_id`;

  if (where.length > 0) {
    const whereClause = where
      .map(
        ({ column, operator }, index) =>
          `${index === 0 ? "WHERE" : "AND"} ${column} ${operator} ?`
      )
      .join(" ");

    const values = where.map(({ value }) => value);

     console.log(`${initialSql} ${whereClause}`, values) // Ce Console.log est présent dans
                                                         //ci-dessous.
    return database
      .query(`${initialSql} ${whereClause}`, values)
      .then(([res]) => res);
  }
```

Output du console.log:

```
SELECT annonce.*, user.firstname as userName, user.city as userCity, user.email as userEmail,
       category.name as category, product.name as product, location.name as region,
       color.name as color, style.name as style FROM annonce
     JOIN category ON category.id=category_id
     JOIN product ON product.id=product_id
     JOIN location ON location.id=location_id
     JOIN color ON color.id=color_id
     JOIN style ON style.id=style_id
     JOIN user ON user.id=user_id WHERE annonce.product_id = ?
       AND annonce.location_id = ? AND annonce.color_id = ?
       AND annonce.style_id = ? AND annonce.price <= ? [ '8', '10', '1', '5', '420' ]
```

7.6 Tests

Afin de garantir la stabilité et la maintenance du projet, j'ai décidé d'écrire une série de tests. Dans le projet nous retrouvons:

- Les tests **End-to-End** évaluent le comportement d'un système dans son ensemble, simulant le parcours utilisateur complet, souvent à travers l'interface graphique.
- Les tests d'**intégrations** évaluent les interfaces de programmation d'application (API) pour garantir qu'elles fonctionnent correctement en termes de communication, de données et de résultats, généralement sans l'interface utilisateur.
- Les tests **unitaires** ciblent une unité spécifique de code, comme une fonction ou une méthode, pour s'assurer qu'elle produit les résultats attendus dans des conditions isolées.
- Les tests de **composant** teste si un composant se rend avec les données attendues, et que tous les composants testés fonctionnent bien entre eux.

Voici quelques exemples:

Test End-to-End

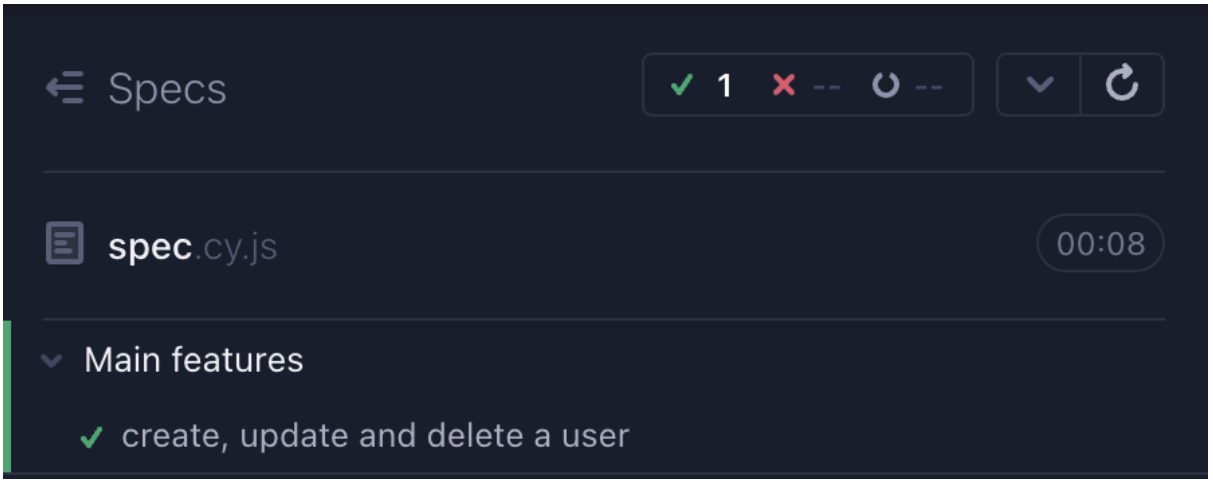
Ce test e2e, écrit avec Cypress, reproduit comme un ‘robot’, les actions d’un utilisateur en interagissant avec l’interface de l’application. Il test le chemin suivant:

Création de compte > login > modification du compte > suppression du compte.

JS spec.cy.js

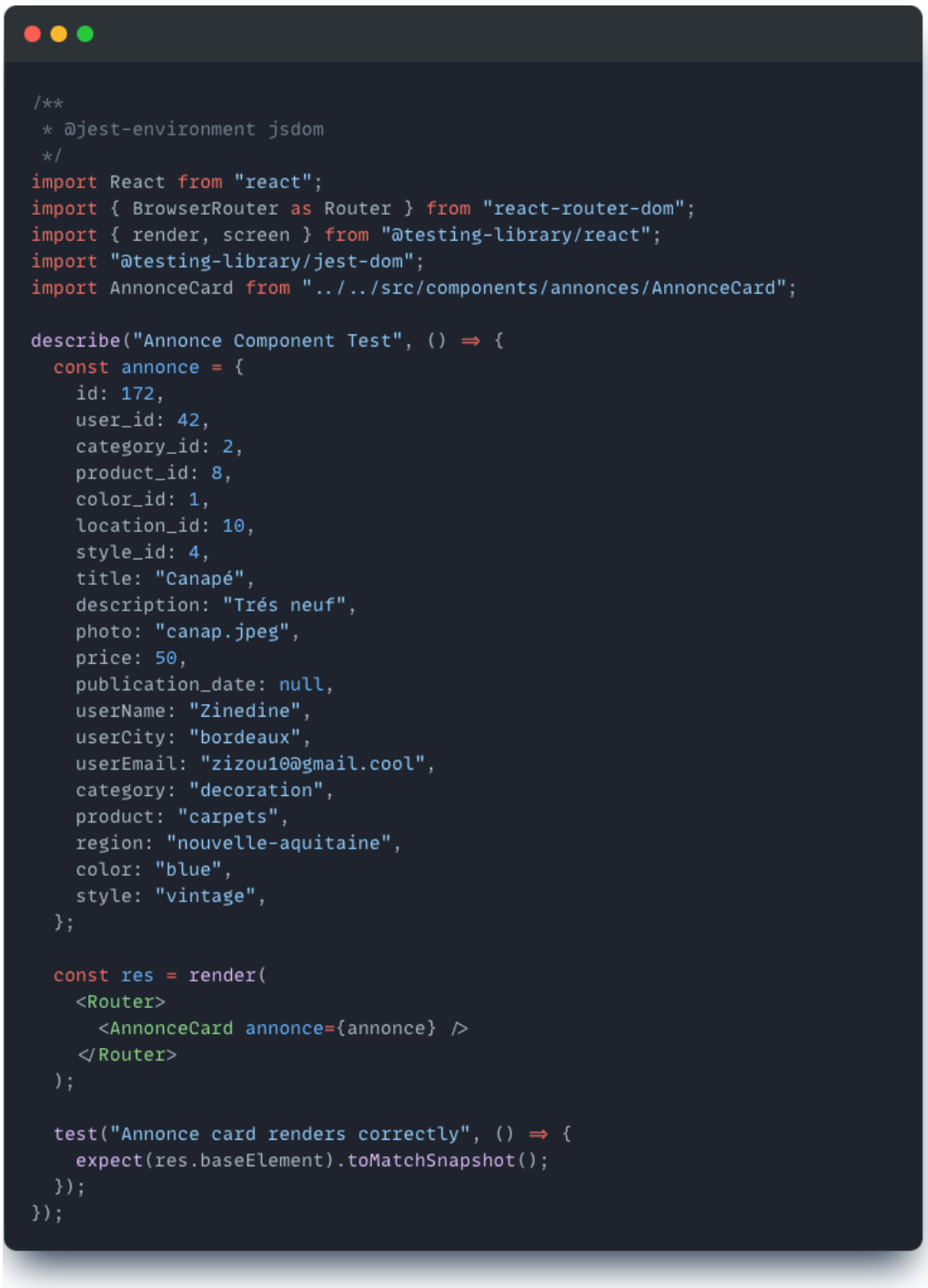
```
describe('Main features', () => {
  it('behave like a user and uses the main features', () => {
      cy.visit('/')
      cy.get('[data-cy="account-icon"').click()

      cy.get('[data-cy="login-link"').click()

      cy.url().should("include",'/login');

      cy.get('[data-cy="signin-link"').click()


      cy.url().should("include",'/signin');

      cy.get('[data-cy="submit"]').should('be.disabled')

      cy.get('[data-cy="firstname"').type(Cypress.env("userData").firstname);
      cy.get('[data-cy="firstname"').should('have.value', Cypress.env("userData").firstname)
      cy.get('[data-cy="lastname"]').type(Cypress.env("userData").lastname);
      cy.get('[data-cy="lastname"]').should('have.value', Cypress.env("userData").lastname)
      cy.get('[data-cy="password"]').type(Cypress.env("userData").password);
      cy.get('[data-cy="password"]').should('have.value', Cypress.env("userData").password)
      cy.get('[data-cy="confirm-password"]').type(Cypress.env("userData").confirmPassword);
      cy.get('[data-cy="confirm-password"]').should('have.value', Cypress.env("userData").confirmPassword)
      cy.get('[data-cy="phone"').type(Cypress.env("userData").phone);
      cy.get('[data-cy="phone"').should('have.value', Cypress.env("userData").phone)
      cy.get('[data-cy="email"').type(Cypress.env("userData").email);
      cy.get('[data-cy="email"').should('have.value', Cypress.env("userData").email)
      cy.get('[data-cy="city"]').type(Cypress.env("userData").city);
      cy.get('[data-cy="city"]').should('have.value', Cypress.env("userData").city)
      cy.get('[data-cy="zipCode"]').type(Cypress.env("userData").zipCode);
      cy.get('[data-cy="zipCode"]').should('have.value', Cypress.env("userData").zipCode)
      cy.get('[data-cy="region"]').select(Cypress.env("userData").location_id);
      cy.get('[data-cy="region"]').should('have.value', Cypress.env("userData").location_id)
      cy.get('[data-cy="submit"]').click()

      cy.url().should('include', '/login')

      cy.get('[data-cy="email"').type(Cypress.env("userData").email);
      cy.get('[data-cy="email"').should('have.value', Cypress.env("userData").email)
      cy.get('[data-cy="password"]').type(Cypress.env("userData").password);
      cy.get('[data-cy="password"]').should('have.value', Cypress.env("userData").password)

      cy.get('[data-cy="submit"]').click();

      cy.wait(1000);

      cy.url().should("eq", Cypress.env("homeUrl"));

      cy.getCookie("session_id").should("exist");

      cy.window().then((win) => {
        const storedData = JSON.parse(localStorage.getItem("account"));
        expect(storedData).to.have.property("state");
        expect(storedData.state).to.have.property("account");
        expect(storedData.state.account).to.have.property("id");
        expect(storedData.state.account).to.have.property("picture");
      });

      cy.get('[data-cy="account-icon"').click()
      cy.get('[data-cy="user-profile"]').click();

      cy.url().should("include", "/userprofile");


      cy.get('[data-cy="edit-icon"]').click();

      cy.get('[data-cy="firstname-input"]').type('updated');
      cy.get('[data-cy="firstname-input"]').should('have.value', 'updated')

      cy.get('[data-cy="submit"]').click();

      cy.get('[data-cy="firstname"]').contains('updated');

      cy.get('[data-cy="delete-icon"]').click();

      cy.window().then((win) =>
        cy.stub(win, 'confirm').as('confirm').returns(true),
        )
      })
})
```

Output:



Tests de composant

Les tests de composant ont été écrits avec Testing Library React et Jest. Le test suivant permet de tester le rendu du composant **AnnonceCard**. Il utilise la méthode *toMatchSnapshot*() et compare les données du rendu courant avec le 'snapshot' enregistré précédemment. Si les deux sont identiques alors le test réussit, sinon il échoue. *baseElement* représente l'élément racine du DOM.



```
/**
 * @jest-environment jsdom
 */
import React from "react";
import { BrowserRouter as Router } from "react-router-dom";
import { render, screen } from "@testing-library/react";
import "@testing-library/jest-dom";
import AnnonceCard from "../../src/components/annonces/AnnonceCard";

describe("Annonce Component Test", () => {
  const annonce = {
    id: 172,
    user_id: 42,
    category_id: 2,
    product_id: 8,
    color_id: 1,
    location_id: 10,
    style_id: 4,
    title: "Canapé",
    description: "Trés neuf",
    photo: "canap.jpeg",
    price: 50,
    publication_date: null,
    userName: "Zinedine",
    userCity: "bordeaux",
    userEmail: "zizou10@gmail.cool",
    category: "decoration",
    product: "carpets",
    region: "nouvelle-aquitaine",
    color: "blue",
    style: "vintage",
  };

  const res = render(
    <Router>
      <AnnonceCard annonce={annonce} />
    </Router>
  );

  test("Annonce card renders correctly", () => {
    expect(res.baseElement).toMatchSnapshot();
  });
});
```

Test unitaire

Le test unitaire qui suit, test si un mot de passe correspond au format requis, à savoir : une majuscule, une minuscule, un chiffre, un caractère spécial et une

longueur d’au moins 8 caractères.

JS passwordUtils.test.js

```
import {
  isValidPassword,
  arePasswordsIdentical,
} from "../../src/utils/paswords";

describe("test the password functions", () => {
  it("test the function isValidPassword", () => {
    expect(isValidPassword("ValidPass123!")).toBe(true);
    expect(isValidPassword("WeakPass")).toBe(false);
  });

  it("test the function arePasswordsIdentical", () => {
    expect(arePasswordsIdentical("password123", "password123")).toBe(true);
    expect(arePasswordsIdentical("password1", "password2")).toBe(false);
  });
});
```

Output:

```
> template-fullstack@1.0.0 test
> npx jest --config=jest.config.js --detectOpenHandles

 PASS  frontend/tests/components/AnnnonceCard.test.jsx
 PASS  frontend/tests/integrations/api.test.js
 PASS  frontend/tests/components/login.test.jsx
 PASS  tests/units/passwordUtils.test.js

Test Suites: 4 passed, 4 total
Tests:       7 passed, 7 total
Snapshots:   1 passed, 1 total
Time:        2.083 s
Ran all test suites.
```

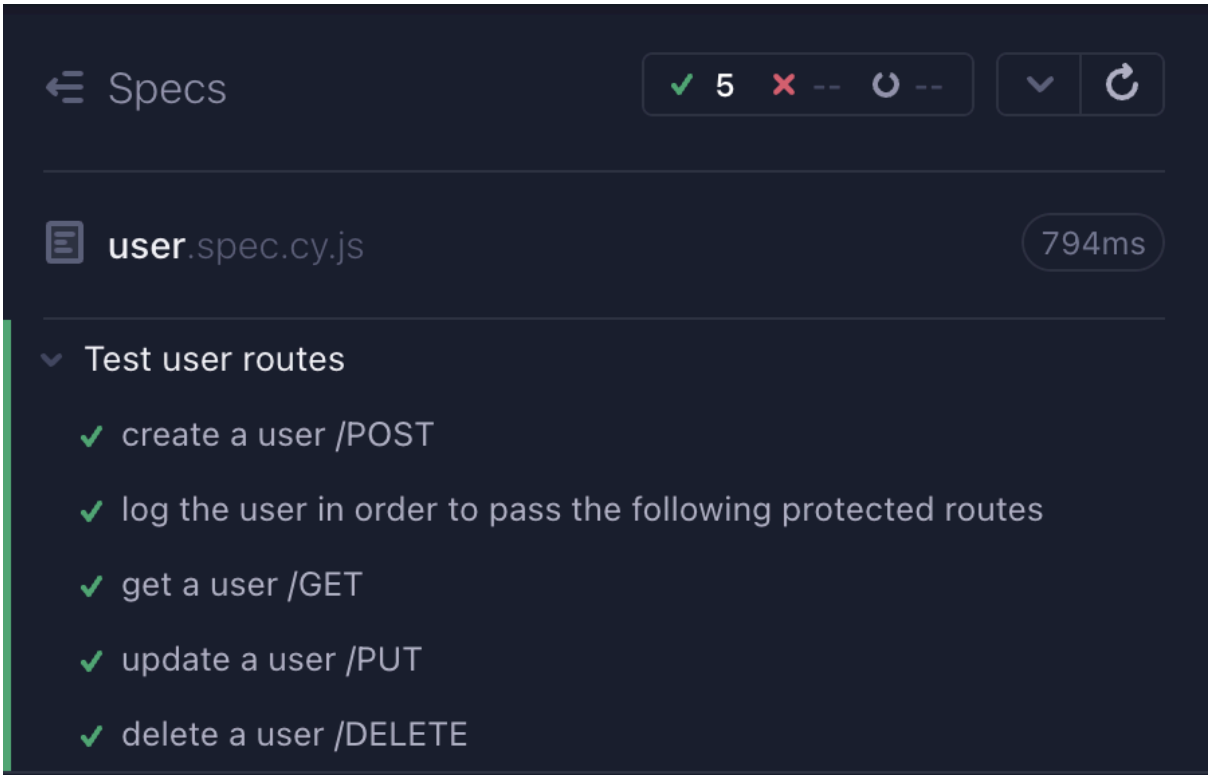
Test d'intégration

Les tests d’intégrations garantissent que l'API fonctionne correctement, respecte les spécifications et fournit les résultats attendus. Cela contribue à garantir la qualité globale de l’app. Ils permettent également de détecter les problèmes dès le début du processus de développement, ce qui facilite la résolution des problèmes avant qu'ils ne deviennent plus complexes et coûteux à corriger.

login.test.jsx

```
describe('Test user routes',()=> {
    let currentUserId;
    let cookie;
    it('create a user /POST', ()=> {
        cy.request('POST', 'http://localhost:8008/api/user/',{
            firstname: "testUser",
            lastname: "test",
            password: "Toto123!!",
            phone: "0606060606",
            email: "tata@gmail.com",
            city: "Bordeaux",
            zipCode: 33000,
            location_id: 10
        }).then((response) => {
              expect(response.status).to.eq(201);
              expect(response.body).to.have.property('id')
              expect(response.body).to.have.property('firstname')
              expect(response.body).to.have.property('email')
              currentUserId  = response.body.id
            });
    })
    it('log the user in order to pass the following protected routes', ()=> {
        cy.request('POST', 'http://localhost:8008/api/user/login', {
            'email': 'tata@gmail.com',
            password: "Toto123!!",
        }).then((response) => {
            expect(response.status).to.eq(200)
            cy.log(response.headers['set-cookie'][0])
            const cookieString = response.headers['set-cookie'][0]
            const sessionIdMatch = cookieString.match(/([^;]+)/);
            cookie = sessionIdMatch ? sessionIdMatch[1] : null;
        })
    })
    it('get a user /GET', () => {
        cy.request({
            method: 'GET',
            url: `http://localhost:8008/api/user/${currentUserId}`,
            headers: {
              Cookie: cookie,
            },
          }).then((response) => {
              expect(response.status).to.eq(200);
              expect(response.body).to.have.property('id')
              expect(response.body).to.have.property('firstname')
              expect(response.body).to.have.property('email')
            });
    })

    it('update a user /PUT', () => {
        const json = JSON.stringify(
    {
        firstname: "mbappé",
        lastname: "test",
        city: "Bordeaux",
        zipCode: 33000,
        location_id: 10,
        phone: "0606060606",
        email: "tata@gmail.com",
        picture: null
    })
    cy.request({
        method: 'PUT',
        url: `http://localhost:8008/api/user/update/${currentUserId}`,
        headers: {
            Cookie: cookie,
            'Content-Type': 'application/json',
        },
        body:{ json : json}
      }).then((response) => {
            expect(response.status).to.eq(200);
            expect(response.body).to.have.property('id')
            expect(response.body.id).to.eq(currentUserId)
            expect(response.body).to.have.property('picture')
        })

    })

    it('delete a user /DELETE', () => {
        cy.request({
            method: 'DELETE',
            url: `http://localhost:8008/api/user/${currentUserId}`,
            headers: {
              Cookie: cookie,
            },
          }).then((response) => {
            expect(response.status).to.eq(204);
        })

    })
})
```

Output:



7.7 Validation des données

Un aspect crucial de la sécurité de notre application concerne la validation des données. Dans toute application, il est impératif de vérifier les données en entrée afin d'éviter l'enregistrement de données non valides dans la base. En effet, les données insérées peuvent potentiellement présenter des risques en transmettant du code malveillant à un utilisateur. La validation des données du côté client ne garantit aucune sécurité, car un attaquant peut contourner l'interface front-end en utilisant des outils tels que cURL ou Postman pour effectuer des requêtes.

Côté client

Par exemple, lorsqu'un utilisateur crée un compte, nous validons la conformité du formulaire côté front-end, via les attributs html 'required', et les types d'input. **Comme cité plus haut React échappe automatiquement tous les éléments <script></script>**. Dans l'exemple ci-dessous, pour des raisons de précision, j'ai choisi de vous montrer le bouton de soumission du formulaire.

Le formulaire pourra être soumis **seulement** si le formulaire est totalement rempli.

```
const dataValidation = (data) => {
  setIsFormValid(
     Object.values(data).every(
      (value) => value !== null && value.trim() !== "" && value !== undefined
    )
  )
}

<Button
  type={button.role}
  disabled={!isFormValid}
  data-cy='submit'
  className='button'
  label={button.label}
/>
```

Côté serveur

Valider les données côté client ne suffit pas à contrer d'éventuelles attaques, et pour garantir une application fonctionnel, il faut systématiquement valider les données reçus côté serveur avant de les enregistrer en base de données.En effet, avec des outils comme Postman, des utilisateurs malveillant pourraient alors faire des requêtes sans passer par l'interface utilisateur. Pour pallier cela, j'ai utilisé le middleware Express-validator. Express Validator est un middleware de validation de données pour Express. Il simplifie le processus de validation des données entrant dans les requêtes HTTP. On peut définir des règles de validation pour les paramètres de requête, les paramètres de chemin, les paramètres de corps (body), etc.

Dans l'exemple ci dessous, je vérifie que les données du formulaire sont conformes à ce que la base de données attend. la méthode escape() permet d'échapper les caractères de type '</>" & ... '.

```
const { check, validationResult } = require("express-validator");

const validateUserData = [
  check("email").isEmail().trim().escape().normalizeEmail(),
  check("firstname").isString().trim().escape(),
  check("lastname").isString().trim().escape(),
  check("city").isString().trim().escape(),
  check("zipCode").isNumeric().trim().escape(),
  check("location_id").isNumeric().trim().escape(),
  check("phone").isNumeric().trim().escape(),
  check("password").isString().trim().escape(),
  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      const error = errors.array()[0];
      return res.status(400).json(`Invalid value for ${error.param}`);
    }
    return next();
  },
];

module.exports = validateUserData;
```

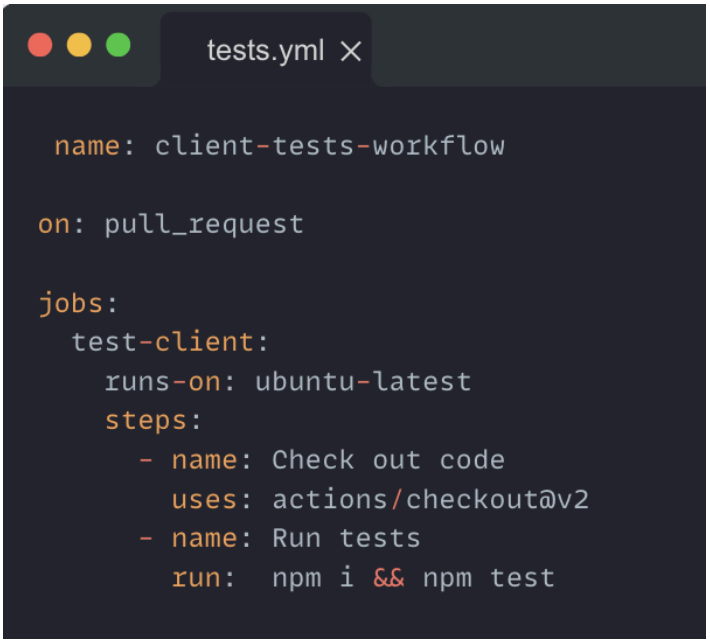
Et je vérifie également que les champs sont bien remplis:

```
const dataValidation = (data) => {
  return Object.values(data).every(
    (value) => value !== null && value !== "" && value !== undefined
  );
};

const isFormFilled = (req, res, next) => {
  if (dataValidation(req.body)) {
    return next();
  }
  return res.status(403).json("form non-valid");
};

module.exports = isFormFilled;
```

Automatisation des tests

La solution de CI/CD, GitHub actions, est intégrée aux repositories GitHub et permet de déclencher des workflows en réponse à des événements tels que les

push sur une branche. Le déploiement d'un pipeline Actions peut être configuré manuellement en éditant deux fichiers YAML associés à un référentiel ou en choisissant une recommandation suggérée par GitHub. Dans ce projet, j'ai utilisé cette solution pour automatiser le lancement des tests avant chaque merge sur la branche develop.



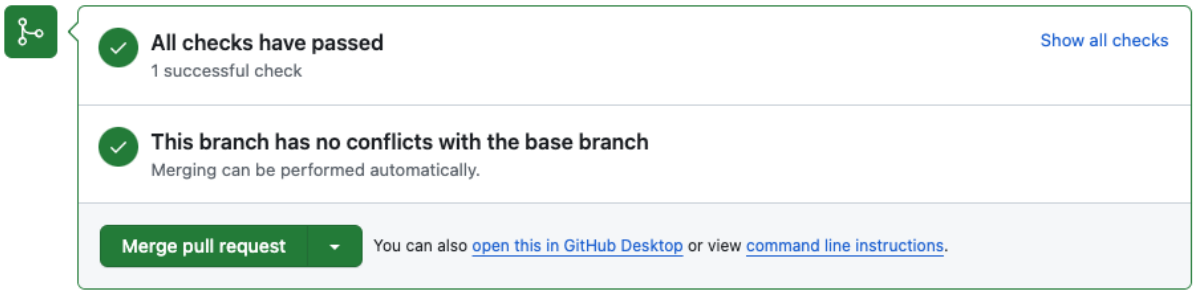
Github actions OLIM

Sur Olim, j'ai ajouté une règle qui nous empêche de merger notre code sur la branche develop si les tests échoue. Pour cela j'ai créé un dossier .github / workflows avec un fichier tests.yml? Puis j'ai créé une règle qui autorise le merge sur la branche develop seulement si les test passent.

(settings > Branches > add rule > Require status checks to pass before merging)

tests.yml

```yaml
name: client-tests-workflow

on: pull_request

jobs:
  test-client:
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v2
      - name: Run tests
        run:  npm i && npm test
```



Sécurité

La sécurité d'une application revêt une importance cruciale pour garantir l'intégrité de l'ensemble des données qui y circulent. Tout au long du processus de développement, les développeurs doivent veiller à mettre en place des bonnes pratiques de sécurité afin de prévenir les principales vulnérabilités telles que les injections SQL, les attaques XSS, le vol de cookies, etc.

React

React,est une bibliothèque qui intègre des mécanismes de protection. En effet comme vu précédemment, React fonctionne au moyen de composants, tous regroupés au sein d'une balise DOM appelée "root". React exerce un contrôle sur le rendu de chaque élément du DOM et échappe automatiquement tous les éléments <script></script> rencontrés à l'intérieur des balises telles que , , et autres. Cela constitue une protection naturelle contre les injections de code, bien que cette mesure ne soit pas universellement applicable.

Si dans un champ de saisie, un utilisateur malveillant venait à saisir un texte comme: <script> alert(‘Coucou’) </script>, il apparaîtrait dans le DOM dans une balise sous forme de texte, les balises script échappées. Ce qui rendrait ceci:

<script>alert(‘coucou’)<script&gt

Cependant, bien que les balises soient sécurisées, tous les attributs au sein de ces balises ne bénéficient pas de la même protection. Par exemple, l'attribut href de la balise de lien <a> ou encore l'attribut onerror de la balise <img /> peuvent être exploités pour exécuter du code JavaScript. Il faut, pour remédier à cette faille, valider les données côté serveur avant de les enregistrer dans la base de données.

Dans le contexte des liens, ajouter l'attribut rel='noopener'empêche l'accès à l'objet window.opener. Cela réduit le risque d'exploitation par des attaques de type phishing.

Injections SQL

Une injection SQL est une attaque où un utilisateur malveillant insère des instructions SQL non sécurisées dans une requête, exploitant ainsi les vulnérabilités du système. L'utilisation ci-dessous de la variable [id] dans une requête préparée crée un espace réservé sécurisé, évitant ainsi les injections SQL

en empêchant l'exécution directe de code SQL non vérifié dans le champ d'identification. Cela renforce la sécurité en dissociant les données des instructions SQL.

Exemple:

7.8 Développer une application mobile.

Comme mentionné plus tôt, pour la partie application mobile j'ai utilisé React Native, compte tenu de sa similarité avec React, le fameux **Learn once write everywhere**.

Pour m'aider dans le développement, j'ai utilisé l'outil Expo qui permet de créer un environnement de développement en faisant tourner une appli react native directement sur mon téléphone.

L'application mobile est une copie conforme en termes d'utilisation, on peut y effectuer les mêmes actions que sur la version web.

Pour gérer la navigation en React Native j'ai utilisé la bibliothèque React Navigation, et son module Bottom Tab Navigator. Bottom Tab Navigator nous permet de créer une barre d'onglet en bas de l'écran afin de naviguer entre différents écrans/fonctionnalités.

Pour cela il faut installer le module, puis créer des écrans, ici nous retrouvons Favorite.js, NewAdd.js et Profile.js. Ensuite nous encapsulons nos écrans dans Navigation Container, qui sert de point d'entrée principal pour la configuration de la navigation. Puis dans le composant Tab.Navigator.

Dans nos Tab, nous devons configurer le “screen” (composant) à afficher via l’attribut “component”, et aussi le nom que nous voulons afficher. Ici seulement des icônes. Voici un exemple:

```
import React from 'react';
import { StyleSheet } from 'react-native';
import NewAdd from './screens/NewAdd';
import Favorite from './screens/Favorite';
import UserProfile from './screens/UserProfile';
import FavoriteIcon from './assets/FavoriteIcon'
import PlusIcon from './assets/PlusIcon'
import ProfileIcon from './assets/ProfileIcon'
import { NavigationContainer } from "@react-navigation/native";
import { createBottomTabNavigator } from "@react-navigation/bottom-tabs";
const Tab = createBottomTabNavigator();


const BottomBar = () => {
  return (
    <NavigationContainer>
      <Tab.Navigator
      screenOptions={({}) => ({
          tabBarActiveTintColor: "blue",
          tabBarInactiveTintColor: "gray",
        })}
      >
        <Tab.Screen  options={{
            tabBarIcon: () => (
              <FavoriteIcon />
            ),
          }}
          name="Favorite"
          component={Favorite}
           />
        <Tab.Screen
         name="NewAdd"
         options={{
            tabBarIcon: () => (
              <PlusIcon />
            ),
         }} component={NewAdd} />
        <Tab.Screen
         name="Profile"
         options={{
            tabBarIcon: () => (
              <ProfileIcon />
            ),
          }} component={UserProfile} />
      </Tab.Navigator>
    </NavigationContainer>
  );
};

export default App;
```

8. Design

Une étape du développement d'une application web consiste à désigner les interfaces utilisateurs. C'est grâce à cette interface que l'utilisateur va pouvoir effectuer des actions, telles qu'ajouter une annonce, se connecter...

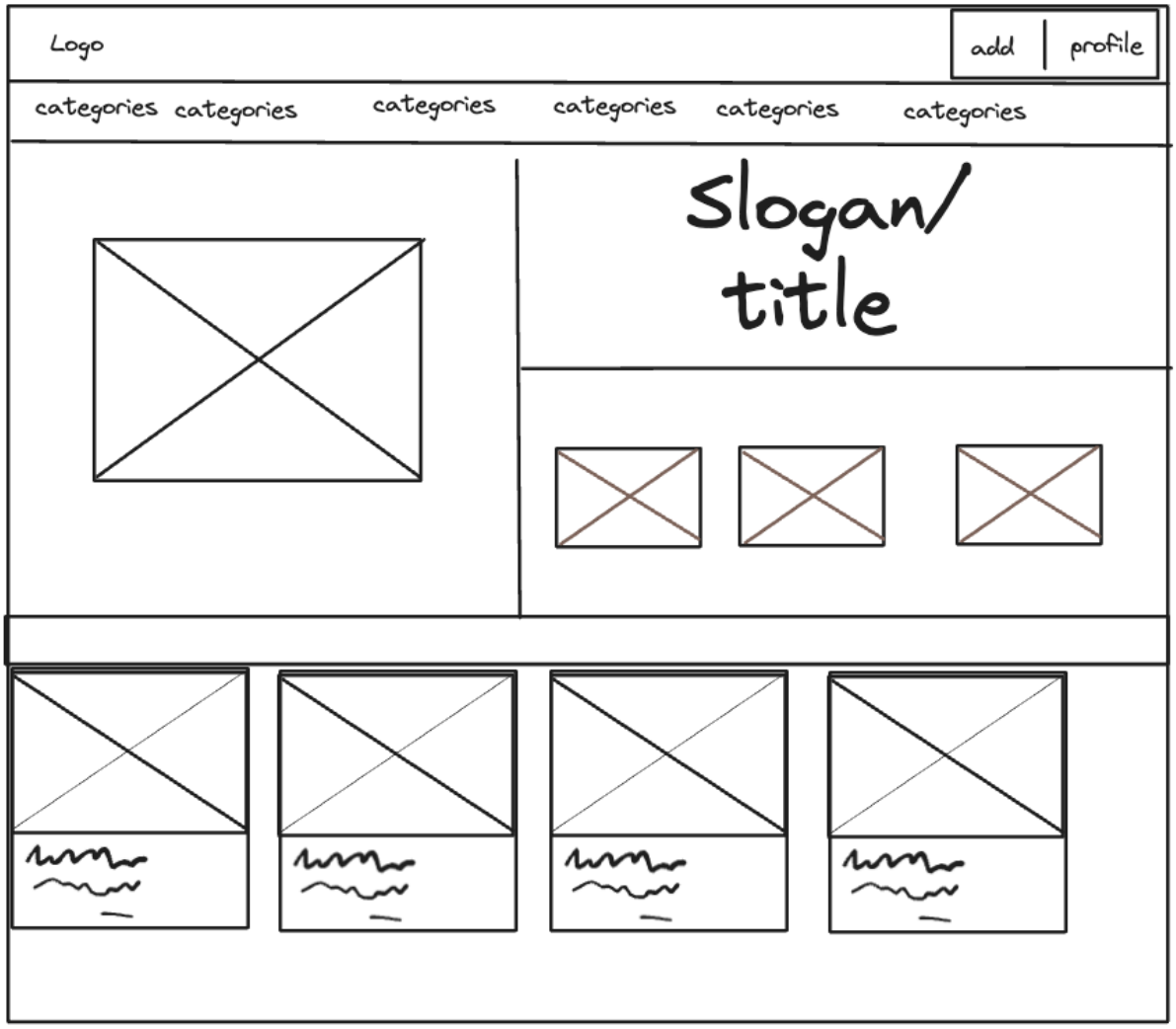
Lors du design de l'interface, nous sommes passé par trois étapes:

- **Wireframe:** C'est le schéma qui nous permet de visualiser la structure et les fonctionnalités de notre application. Il est en général très peu fidèle au design final et est évolutif.
- **Charte graphique:** C'est le document qui va nous permettre d'établir les règles globales en termes de design de l'application. Les couleurs, les polices... En bref, l'identité de la marque ou du produit.
- **Maquette:** C'est à ce moment-là que nous avons défini la mise en page définitive de notre application.

Dans la plupart des cas, une étape de prototypage a aussi lieu. Avant l'étape de la maquette finale. Il est moins fidèle au design final mais contient déjà la plupart des éléments finaux et permet au client d'effectuer des changements avant la livraison de la maquette finale.

8.1 Wireframe

Pour le wireframe, nous avons utilisé l'outil en ligne excalidraw. Le but était de discuter du placement des différents éléments, en imaginant un parcours utilisateurs. Nous avons gardé un œil sur les user stories lors de cette étape. Voici un exemple du wireframe de notre page d'accueil:



8.2 Charte graphique

Lors de cette étape, nous avons discuté des couleurs à utiliser, ainsi que des polices, en gardant à l'esprit d'éventuels soucis d'accessibilités.

#E4F5FF

#00BA63

#FF7F37

#FEBEE8

8.3 Maquette

Pour la maquette nous avons utilisé Figma. Nous avons désigné tous les éléments graphiques, rassemblés par composants, puis réutilisés là où nous le souhaitions.

Voici un exemple de composant du menu:

Voici la maquette de la page d'accueil d'Olim:

Partie 2

1. Collaboration à la gestion de projet

Présentation du projet Nuclear-Blast Amaranthe

Le projet Nuclear-Blast Amaranthe est un projet réalisé dans mon entreprise d'accueil lors de mon alternance. Le projet consiste en une petite expérience afin de faire la promotion du nouvel album du groupe de metal Amaranthe. L'utilisateur sélectionne un 'mood' parmi quatre propositions. A l'issue du choix du mood, nous invitons les utilisateurs à choisir ces 3 musiques préférées. Ensuite nous lui fabriquons une playlist qui 'match' avec son mood et son choix de musiques. L'utilisateur peut alors se connecter à son compte spotify et la playlist est automatiquement sauvegardée sur son compte.

Pour ce projet j'ai travaillé avec la cheffe de projet, ainsi que le directeur artistique de l'agence. Dans un premier temps la cheffe de projet nous a expliqué les attentes et les envies du client, nous devions créer une expérience qui permette de générer une playlist personnalisée à chaque utilisateur. Nous nous sommes vite rendu compte qu'une expérience qui pourrait générer des "streams" sur Spotify serait un plus pour les artistes. J'ai fait quelques rapides recherches afin de m'assurer de la faisabilité de l'idée.

Nous avons définis un chemin utilisateur comme ceci:

- L'utilisateur doit choisir un 'mood' parmi quatre choix.
- L'utilisateur doit renseigner ces 3 titres favoris. (parmis la discographie du groupe)
- L'utilisateur peut arranger la playlist générée comme il le souhaite.
- L'utilisateur doit se connecter à Spotify pour enregistrer la playlist.
- L'utilisateur peut partager sa playlist sur les réseaux sociaux.

Une fois notre idée validée, nous l'avons proposée au client sous forme de wireframe.

Le directeur artistique à travaillé sur les maquettes et j'ai commencé à développer la partie Spotify de l'application.

Une fois les maquettes validées par le client j’ai pu commencer à développer l’interface.

Chaque matin, avec la cheffe de projet, nous faisons le point sur l'avancée du projet, les tâches du jour, et chaque vendredi nous livrions nos avancées au client afin d’avoir des retours le plus rapidement possible. Grâce à ces retours, cela nous a permis de travailler plus en profondeur sur l'algorithme de génération de playlist.

2. Architecture CI / CD

Sur le projet Nuclear-Blast, j’ai utilisé l’architecture déjà existante et j’ai complété cela en configurant une CI (intégration continue) et une CD (Déploiement continu) afin d’automatiser le processus.

Environnements de staging / prod

Dans mon entreprise, nous disposons de deux serveurs, un pour les stagings et un pour les productions. Ils sont tous deux hébergés sur Digital Ocean.

Chaque serveur utilise un reverse proxy (traefik) qui tourne et qui répartit les requêtes sur les différents serveurs. Ainsi, chaque URL qui commence par ‘staging’ sera redirigée vers le serveur de staging.

Dockerisation de l’app

Afin de faciliter le développement et le déploiement, nous avons dockerisé notre application. Pour ce faire j’ai créé un Dockerfile, puis configuré un docker-compose-staging.yml et docker-compose-production.yml

docker-compose-staging.yml

```yaml
version: "3.5"

networks:
  traefik:
    external: true

services:
  app:
    image: worker.lostmechanics.cool/nuclearblast-amaranthe-staging
    build:
      context: .
      args:
        - APP_URL=https://staging.nuclear-blast-amaranthe.lostmechanics.cool

        - SPOTIFY_CLIENT_ID=a90e09763f4d4999ac42eeccdc04035f
        - SPOTIFY_CLIENT_SECRET=${SPOTIFY_CLIENT_SECRET}

        - ENSO_ENTRY=https://staging.enso.lostmechanics.cool/api/
        - ENSO_CLIENT_KEY=fcd78761b8cd26382cc1
        - ENSO_PROJECT_SLUG=amaranthe
        - ENSO_CLIENT_SECRET=${STAGING_ENSO_CLIENT_SECRET}

        - ANALYTICS_SITE_ID=14
    networks:
      - traefik
    deploy:
      labels:
        - traefik.enable=true
        - traefik.http.routers.nuclearblast-amaranthe.rule=Host(`staging.nuclear-blast-amaranthe.lostmechanics.cool`)
        - traefik.http.routers.nuclearblast-amaranthe.tls=true
        - traefik.http.routers.nuclearblast-amaranthe.tls.certresolver=letsEncrypt
        - traefik.http.services.nuclearblast-amaranthe.loadbalancer.server.port=80
```

Les secrets sont configurés depuis Gitlab (Settings > CI / CD > Variables)

Déploiement continu

Pour le déploiement continu, nous utilisons Gitlab CI. Il permet d'automatiser le processus de construction, de test et de déploiement à partir d'un répositotie GitLab. Aussi, on peut définir des pipelines d'intégration continue dans des fichiers de configuration YAML. Ces pipelines peuvent inclure des étapes telles que la compilation du code, l'exécution de tests automatisés et le déploiement vers les environnements. Ainsi, comme nous le voyons ci-dessous, un push sur la branche develop déploie notre application directement sur la staging.

docker-compose-staging.yml

```yaml
stages:
  - deploy

deploy_staging:
  stage: deploy
  tags:
    - worker # Runner Gitlab à utiliser
  script:
    - docker-compose -f "docker-compose-staging.yml" build
    - docker-compose -f "docker-compose-staging.yml" push

    - scp "docker-compose-staging.yml" root@staging.lostmechanics.cool:/var/apps/services/nuclearblast-amaranthe-staging.yml
    - ssh root@staging.lostmechanics.cool "docker stack deploy --prune --with-registry-auth --resolve-image=always --compose-file /var/apps/services/nuclearblast-amaranthe-staging.yml nuclearblast-amaranthe"
  only:
    - develop
```

Conclusion

Ces deux projets ont été très enrichissants. Le premier m’a permis de découvrir et de monter en compétences sur plusieurs concepts qui me paraissaient abstraits. De découvrir le travail en équipe et de monter une app de A à Z. Des améliorations restent néanmoins à apporter.

Les projets réalisés en entreprise m’ont permis de prendre connaissance de mes axes d’améliorations, de m’adapter à la façon de travailler des autres développeurs, qui ont plus d'expérience. Ils m’ont confirmé qu’il n’y a pas une seule manière de faire les choses, que rien n’est figé dans le développement web, que l’on apprend sans arrêt. Que certaines méthodes ne s'appliquent pas à tous les projets. Que c’est ok de chercher une réponse pendant plusieurs heures.

Annexes

Diagramme de cas d’utilisation:

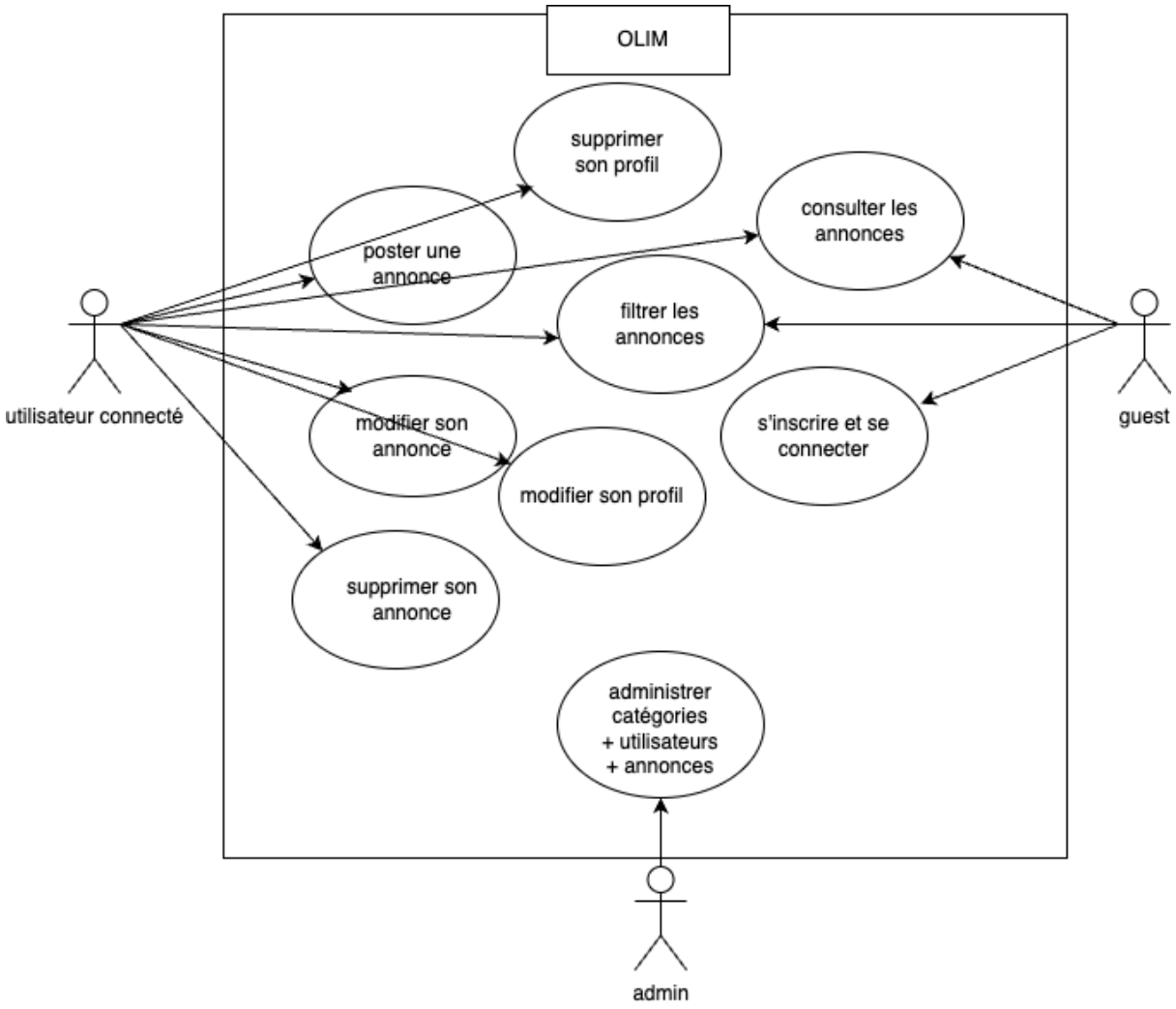
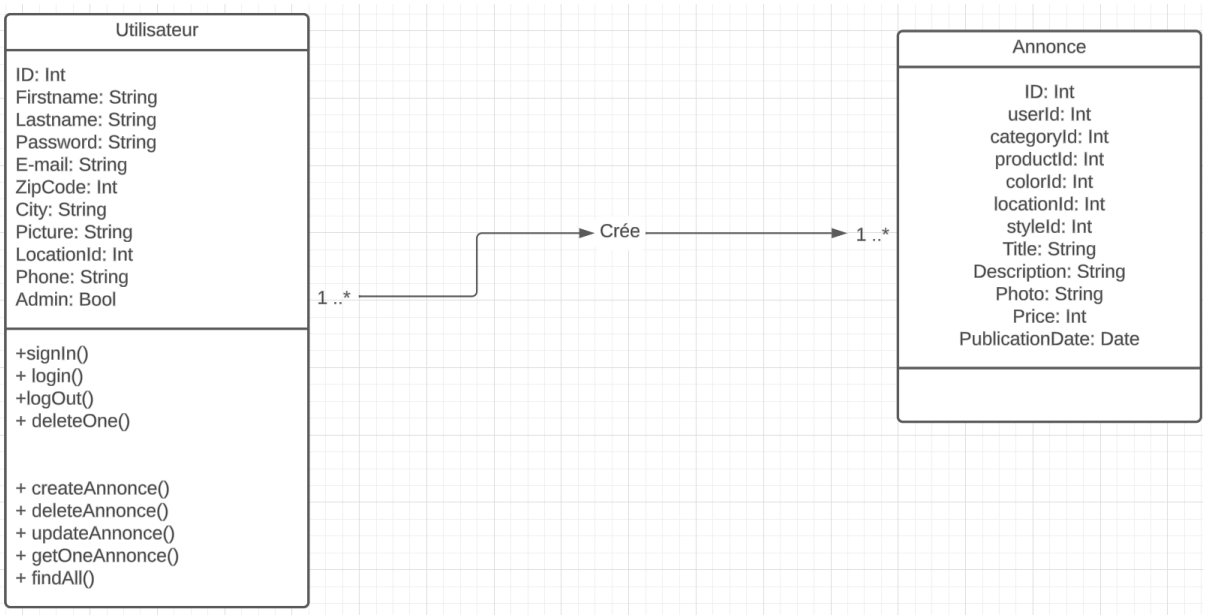


Diagramme de classe:



Github actions OLIM

Sur Olim, j'ai ajouté une règle qui nous empêche de merger notre code sur la branche develop si les tests échoue. Pour cela j'ai créé un dossier .github / workflows avec un fichier tests.yml? Puis j'ai créé une règle qui autorise le merge sur la branche develop seulement si les test passent.

(settings > Branches > add rule > Require status checks to pass before merging)

tests.yml

```yaml
name: client-tests-workflow

on: pull_request

jobs:
  test-client:
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v2
      - name: Run tests
        run:  npm i && npm test
```