AVIGNON UNIVERSITÉ

# Rapport Urbanisation

Étudiants
**Sébastien Rocca**
**Guillaume Bonenfant**

**19 juin 2023**

**Master Informatique**
**ILSEN**
**UE** Conception logicielle
**UCE** Urbanisation

**Responsables**
Mickael Rouvier
Mathias Quillot

UFR
SCIENCES
TECHNOLOGIES
SANTÉ

AVIGNON UNIVERSITÉ
CERI

CENTRE
D'**ENSEIGNEMENT**
ET DE **RECHERCHE**
EN **INFORMATIQUE**
ceri.univ-avignon.fr

# Sommaire

# 1 Introduction

## 1.1 Le projet

### 1.1.1 Description

Dans le cadre de l'UE Urbanisation du Master 1 au CERI, il nous a été demandé de concevoir et de programmer une application de reconnaissance de bouteille de vin par la photo afin de trouver des informations pertinentes sur la bouteille telle qu'une description, l'année de la cuvée, le cépage ainsi que les divers avis et notes des utilisateurs ayant goûté au produit.
Cette application devait donc aussi laisser l'utilisateur interagir pour pouvoir laisser des avis et également pouvoir laisser des administrateurs supprimer des messages si ces derniers ne sont par exemple pas pertinent ou bien encore ajouter, modifier, supprimer des vins.

### 1.1.2 Contraintes

Comme tout projet contient des contraintes, celui-ci ne fait pas exception. Nous les développeurs allons devoir prendre en compte une montée en charge de l'application. En effet, cette dernière connaîtra un fort succès rapidement et elle devra donc pouvoir supporter un fort afflux d'utilisateurs. Rien n'est plus ennuyant qu'une application inutilisable dès sa sortie !

Une autre contrainte qui n'est pas énoncée dans le sujet, mais qui est également présente dans tous les projets : la contrainte de coût ! Comme nous n'avons pas de budget pour ce projet, son développement doit rester gratuit !

## 1.2 Nos débuts

Nous avons donc commencé par concevoir l'application et exprimer tous nos besoins de façon à éclaircir notre projet, ses différentes utilisations, comment nous allions la programmer, quelle infrastructure nous allions avoir pour respecter tous les critères de demandés. La conception s'appuie sur différents axes :

- l'architecture fonctionnelle : Comportement de l'application et définition d'un scénario type avec des diagrammes Usecase et Séquence
- l'architecture logicielle : choix des différents logiciels pour programmer l'application, dans notre cas un langage pour le client mobile, un langage backend pour l'api, un SGBD et un logiciel pour analyser les photos
- l'architecture technique : définition des blocs applicatifs et de leurs interactions

# 2 Conception

## 2.1 Architecture Fonctionnelle

L'architecture fonctionnelle permet de définir les composants fonctionnels métiers des activités métiers, tout en gardant une couche d'abstraction quant à l'implémentation concrète de ces composants et la conception détaillée du logiciel.

Concrètement, cela signifie que l'on doit décrire les blocs de l'application et leurs interactions sans parler programmation et logiciel pur (code, librairies, protocoles, architecture, choix de base de données...).

### 2.1.1 Scénario

Nous avons imaginé 2 cas d'utilisation de l'application finale : un utilisateur classique et un administrateur.

**L'utilisateur classique** : l'utilisateur va dans un magasin et trouve une bouteille de vin. Il veut savoir ce que contient ce vin. Il ouvre donc l'application, prend en photo l'étiquette de la bouteille et l'application lui montre en retour les informations de la bouteille (cuvée, avis, note...).
Ensuite, l'utilisateur peut alors se connecter (s'il ne l'est pas encore) ou s'inscrire, et écrire un avis ou mettre une note.

**L'administrateur** : l'administrateur veut supprimer les commentaires offensants de l'application. Il ouvre donc l'application, il se connecte avec son compte administrateur et supprime les messages concernés. Puis en regardant les différents vins, il trouve des incohérences, il modifie donc les informations (description, cuvée, nom...) des bouteilles en question.

### 2.1.2 Le diagramme UseCase

Ce diagramme des cas d'utilisation [1] représente de façon simple les actions que peut exécuter un utilisateur classique ou bien un administrateur.

L'utilisateur peut :

- Scanner une bouteille en la prenant en photo. Cette action va ensuite appeler une fonction de traitement de l'image dans le but de retrouver le vin qui correspond dans la base de données.
- Laisser un avis à propos d'un vin. Cette action va d'abord demander à l'utilisateur de se connecter s'il ne l'est pas déjà.

L'administrateur peut :

- Ajouter, modifier ou supprimer un commentaire ou un vin. Cette action va d'abord demander à l'administrateur de se connecter s'il ne l'est pas déjà et vérifier que ce compte est bien un compte administrateur.

### 2.1.3 Le diagramme de séquence

Le diagramme de séquence [2] montre dans quel ordre les fonctionnalités de l'application s'exécutent pour réaliser une action.

Dans le cas de l'utilisateur, il commence par lancer l'application, puis il peut prendre une photo d'un vin. L'application va analyser la photo et afficher les informations de ce dernier. Puis s'il souhaite laisser un commentaire, l'utilisateur va alors d'abord devoir se connecter, écrire et envoyer son commentaire et l'application affichera enfin un message de confirmation du bon envoie de l'avis.

Pour l'administrateur, il commence par lancer l'application, puis se connecte et l'application vérifie en même temps si ce compte est un compte administrateur. Si oui, il aura alors des droits supplémentaires. L'administrateur pourra alors choisir d'ajouter, modifier ou supprimer un commentaire ou un vin

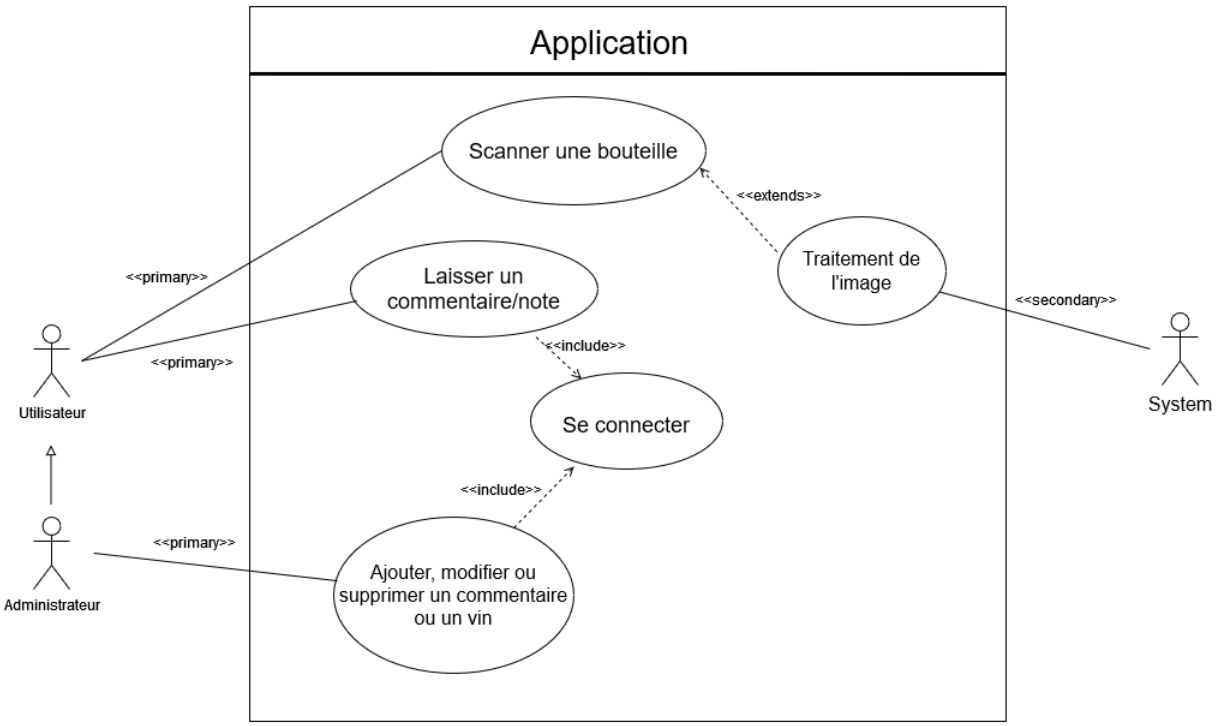


**Figure 1.** Diagramme UseCase

## 2.2 Architecture Logicielle

Ici, nous devons définir des blocs applicatifs (logiciels, langages, composants...) de l'application et leurs échanges, tout en étant cohérent avec l'architecture fonctionnelle [2.1] définie auparavant.

### 2.2.1 Blocs Applicatifs

Nous avons donc comme composants :

- Un client applicatif (l'application mobile)
- Un serveur backend pour l'application, accessible sous la forme d'une API
- Une base de donnée
- Un logiciel de reconnaissance ou d'analyse d'image ou d'écriture

### 2.2.2 Langage pour l'application mobile

Pour le choix du langage, nous avons commencé par établir nos critères pour trouver le langage idéal pour ce projet. Ces critères sont :

- Être cross-plateformes, c'est-à-dire pouvant être utilisé pour programmer pour Android et iOS en même temps.
- Avoir une prise en main relativement facile pour pouvoir être développée rapidement.
- fluidité lors de l'exécution de l'application
- Avoir accès aux API natives (photo principalement)
- Être compatible avec de la reconnaissance d'image

Puis nous avons regardé tous les langages utilisables pour ce type de tâche et nous nous sommes retrouvés avec 4 principaux langages : `Xamarin`, React Native, `Ionic` et `Flutter`.

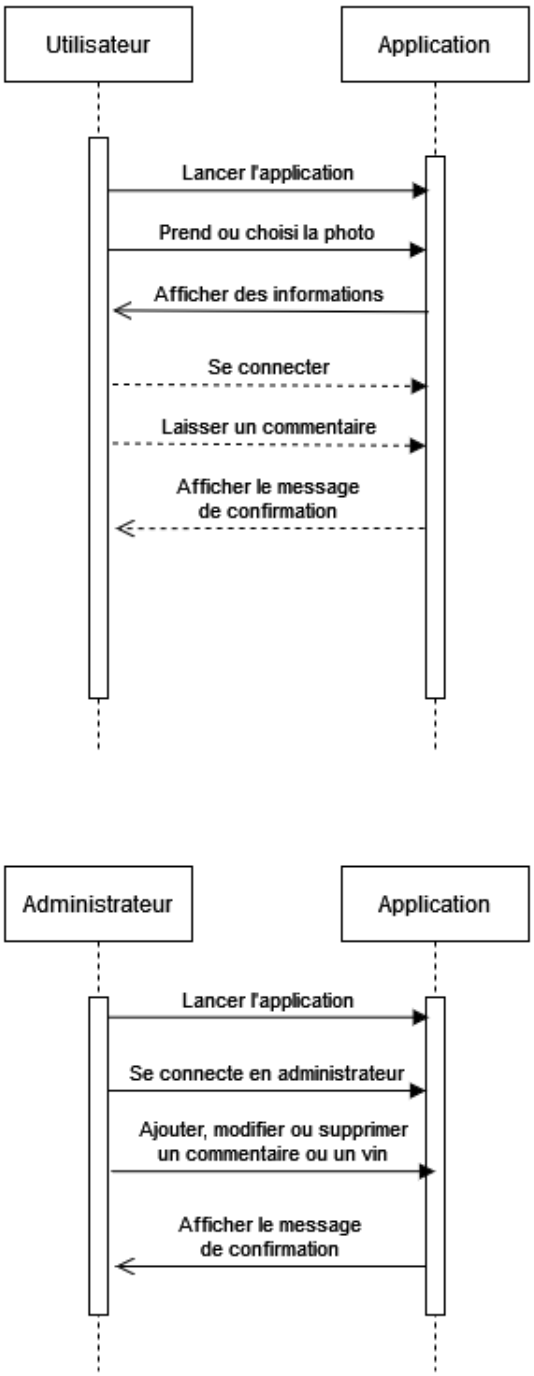


**Figure 2.** Diagramme de séquence

Nous avons ensuite fait un tableau 3 avec les différents critères qui nous paraissaient importants : la prise en main et le langage utilisé, la fluidité du langage, la communauté et les cas d'utilisation de ces langages. Nous avons fait ce tableau à l'aide de ces sources [8], [10], [1], [13], [11].

Pour notre choix final de langage, nous avons décidé d'éliminer `Flutter` et `Xamarin` pour leur prise en main plus longue que les deux autres langages et nous avons enfin départagé React Native et `Ionic` grâce aux cas d'utilisations : `Ionic` est fait pour des applications

| Langage | Prise en main/langage utilisé | Performances | Communauté | Cas d'utilisations |
|---|---|---|---|---|
| **Xamarin** | Moyenne : C# | Bonne | Forte | Applications simples |
| **React Native** | Rapide : Javascript | Très bonne | Forte | Tout type d'applications |
| **Flutter** | Plus longue : Dart | Excellente | Moyenne | Tout type d'applications |
| **Ionic** | Rapide : HTML, CSS, Typescript, Javascript | Bonne | Forte | Applications simples |

**Figure 3.** Comparaison des langages utilisables pour ce projet

simples alors que React Native est fait pour tout type d'applications. Notre application étant relativement simple, nous avons donc conclu que le langage parfait pour ce projet serait `Ionic`.

### 2.2.3 Langage API

Dans notre cahier des charges, il est inscrit que notre application sera fortement utilisée dès sa sortie, potentiellement par des milliers d'utilisateurs.

Nous devons donc utiliser une technologie côté serveur qui sera compatible avec cette montée en charge.

**Pourquoi utiliser une API (Application Programming Interface) ?**

Une API permet de :

- Donner une flexibilité à la conception pour de nouvelles applications sur d'autres supports
- Ne pas avoir à stocker les données sur l'appareil et ainsi éviter de devoir mettre à jour l'application client à chaque nouvelle information ajoutée (commentaire, note, bouteille...).
- Éviter de faire des appels de base de données directes depuis l'application (pour une question de sécurité).

Comment va fonctionner notre API :
Notre API REST possède donc un côté serveur et un côté client. On envoie une requête `HTTP` au serveur de l'API avec en arguments des paramètres (texte, fichier, image...), qui nous retourne les informations souhaitées au format `JSON`.

On a un vaste choix de langage pour développer l'API [1] :
Nous avons donc choisi **Node.js**, qui présente les caractéristiques souhaitées en termes de rapidité, de fonctionnalités et de documentation, sa large base communautaire [3] nous assure de trouver de la documentation aisément.
De plus, `Node.js` inclus également un serveur web au même titre qu'`Apache`, ce qui demande finalement moins de configuration pour des performances supérieures en termes de rapidité de réponse [2].

| Language | Type | Caractéristiques | Documentation | Popularité (github) |
|---|---|---|---|---|
| **NodeJS** | Language | asynchrone, rapide (compilation JIT) | Très bonne | 83.3k étoiles |
| **Flask** | Micro Framework Python | Simple, minimaliste, extensible | Très bonne | 57.2k étoiles |
| **Django** | Framework Python | Complet en fonctionnalité | Très bonne | 60,8k étoiles |
| **Symfony** | Framework PHP | Complet en fonctionnalité | Très bonne | 26k étoiles |

**Table 1.** Tableau comparatif des différents langages utilisables pour une API

### 2.2.4 SGBD

Afin de pouvoir enregistrer les informations concernant les vins ainsi que les avis/notes des utilisateurs et leurs comptes, il nous faut un moyen efficace et conventionnel pour stocker ces informations, nous devons donc choisir un SGBD (Système de Gestion de Base de Données).

Il y a différents Systèmes de Gestion de Bases de Données disponibles sur le marché, Parmi les solutions les plus performantes, les plus connues sont :

- `PostgreSQL` : système de gestion de base de données de classe professionnelle open source, utilisable pour de la haute disponibilité, possède `Pgplsql`, un langage de programmation procédural pour gérer les bases de données et mettre en place des automatismes (calqué sur PL/SQL d'Oracle). Compatible avec la plupart des langages (C, C++, Java, Python, PHP, Node.js...)
- `MySQL` : pour des bases de données relationnelles, Open Source, hautes performances, également compatible avec de nombreux langages (C, C++, Java, Python, PHP, Node.js...)
- `MariaDB` : Fork de MySQL, Open source, haute performance
- `Oracle` : utilisé en entreprise, très fiable, optimisé pour les hautes performances, migration facilité vers et depuis d'autres bases de données existantes, mais propriétaire et payant en fonction de l'utilisation.

Source : [9].

Toutes ces solutions proposent des performances et fonctionnalités similaires : nombreux langages compatibles, multiplateforme (Windows, Linux, parfois macOS).

Pour ce choix, nous n'avons aucune contrainte spécifique, et vu que nous souhaitons avoir une solution peu couteuse et "future-proof", nous avons donc choisi `PostgreSQL` car c'est une solution fiable, connue, documentée et performante.

### 2.2.5 Logiciel d'analyse de photo

Pour la partie de l'analyse de l'étiquette de la bouteille, nous avons choisi d'analyser uniquement le code barre de l'étiquette, plus précisément les numéros en dessous [4] car la police d'écriture des étiquettes sur les bouteilles de vin varient beaucoup en fonction du vigneron rendant l'analyse de l'étiquette elle-même difficile. Analyser le code barre de cette dernière rend donc la tâche d'analyse plus facile.

Pour ce faire, nous devons donc utiliser un logiciel spécialisé efficace.
Il y a différents types de logiciels, comme de la reconnaissance d'image pure avec la comparaison entre 2 images ou bien de la reconnaissance d'écriture (OCR).

Voici une liste des solutions existantes [2] :
Toutes étant très utilisées et bien documentées, les différences majeures sont leurs fonctions : un logiciel de reconnaissance de caractère (OCR), une boîte à outil spécialisé


**Figure 4.** Exemple de code barre avec des numéros en dessous

| Nom | Type | Caractéristiques | Popularité (github) |
|---|---|---|---|
| **Tesseract** | OCR(écriture) | Très bonne reconnaissance, hors ligne | 42.8k étoiles |
| **Pytorch** | Toolkit IA | Très complet, complexe a utiliser | 52.4k étoiles |
| **OpenCV** | Toolkit CV | Très complet, complexe a mettre en oeuvre | 58.2k étoiles |
| **TensorFlow** | Toolkit IA | Complet, mise en oeuvre complexe | 161k étoiles |

**Table 2.** Tableau des différentes solutions de lecture de l'étiquette

dans la reconnaissance et analyse d'image/vidéo, ou encore des boîtes à outil pour IA plus généralistes. L'autre différence est la facilité de mise en œuvre, par exemple Tesseract est plus simple à utiliser pour de la reconnaissance d'écriture (car spécialisé) qu'`OpenCV`, `TensorFlow` ou `Pytorch` [6] [5], qu'il faut configurer voir entrainer pour avoir des résultats corrects.

Dans notre cas, le plus simple est donc d'utiliser `Tesseract`, qui n'a pas besoin d'entrainement, pour reconnaitre les chiffres inscrits en dessous du code barre de la bouteille.

Tesseract est disponible sous `JavaScript` et `NodeJS` avec `Tesseract.js` [12], ce qui facilite grandement le déploiement sur la plateforme, tout en restant hors ligne.

### 2.2.6 Gestion de la montée en charge

Simuler la montée en charge est nécessaire pour savoir si le service proposé sera continu et ce même si de nombreuses personnes l'utilisent simultanément.
Il faut donc définir plusieurs étapes : simuler la montée en charge, mesurer les performances de l'application puis la gérer dans une situation réelle.

**Simulation de la montée en charge :**

Il existe diverses solutions pour simuler la montée en charge et mesurer les performances [7], parmi elles se trouvent `Apache JMeter` et `Artillery`, 2 solutions open sources et éprouvées, toutes deux utilisables sur des API utilisant le protocole `HTTP`.

`JMeter` est utilisable pour des applications web, c'est un logiciel populaire, portable, qui possède une grande communauté (et donc bien documenté), et personnalisable avec ses plugins, mais a pour inconvénient d'être complexe à utiliser.

`Artillery` quant à lui est similaire. Les modèles de test sont écrits en `YAML` (un format de représentation et de structuration de données comme `XML` ou `JSON`), il est facile à prendre en main, permet de tester rapidement et supporte les principales technologies utilisées sur les sites web et applications (`Websocket`, `HTTP`...), son seul défaut est que contrairement à `JMeter`, qu'il n'est utilisable qu'en mode ligne de commande.

Nous nous baserons donc sur `Artillery` pour effectuer nos tests de montée en charge, car nous n'avons pas besoins d'une solution modulaire et plus complexe comme le propose **JMeter** pour une API proposant peu de services pour l'instant.

**Gestion de la montée en charge :**

Pour la gestion de la montée en charge, nous comptons utiliser `Docker` comme solution de conteneurisation pour notre serveur web principal (qui dessert notre API) développé en `Node.js`, ainsi que notre serveur `PostgreSQL`, ce qui permet de déployer les logiciels facilement sous la forme d'instances conteneurisées.

Nous comptons également utiliser `Kubernetes` pour un déploiement des instances conteneurisées sur plusieurs serveurs une gestion automatisée des différents conteneurs en cas de pannes ou de mise à l'échelle en fonction de l'afflux des utilisateurs sur la plateforme, de plus, nous souhaitons utiliser le logiciel `Rancher`, qui ajoute une interface graphique web à `Kubernetes` pour une administration et un suivi simplifié, rendant le tout plus efficace.

## 2.3 Architecture Technique

### 2.3.1 Architecture des composants

Nous avons 4 composants [5] :

- **L'API :** codée en `Node.JS`, qui gère les requêtes envoyées par l'appareil de l'utilisateur et renvoi les informations demandées.
- **Le Client applicatif :** L'application sur le mobile de l'utilisateur, avec laquelle il va pouvoir identifier les vins et voir les informations à leur sujet (description, avis, note...).
- **Le traitement de l'image :** Le logiciel avec lequel nous traitons l'image envoyée par l'utilisateur afin d'en tirer les informations nécessaires (le code du vin, dans notre cas).
- **Le serveur de base donnée :** stocke les informations sur les vins, les utilisateurs ainsi que les avis/note des différents utilisateurs.

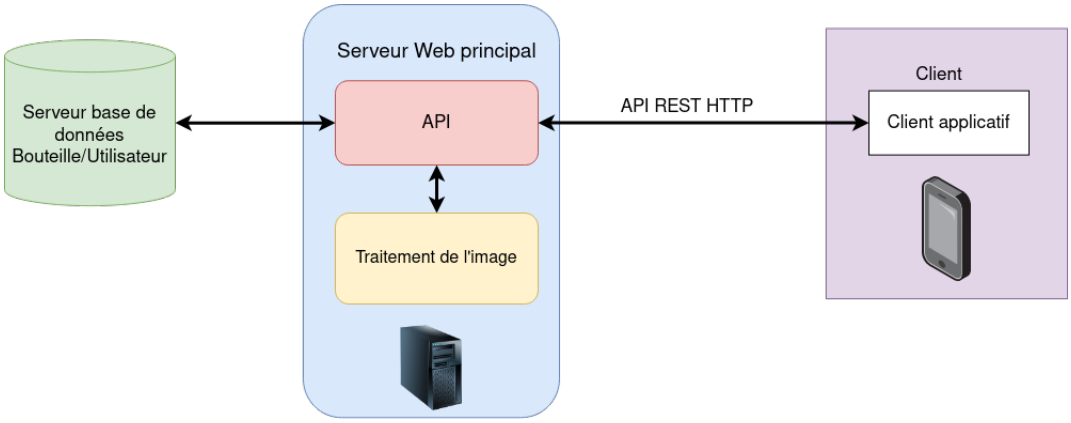


**Figure 5.** Diagramme de l'architecture des composants

### 2.3.2 Cartographie & matrice des flux

Nous avons également réalisé une cartographie et une matrice des flux (voir [6] et [3]) pour montrer comment les composants de l'application communiquent entre eux.

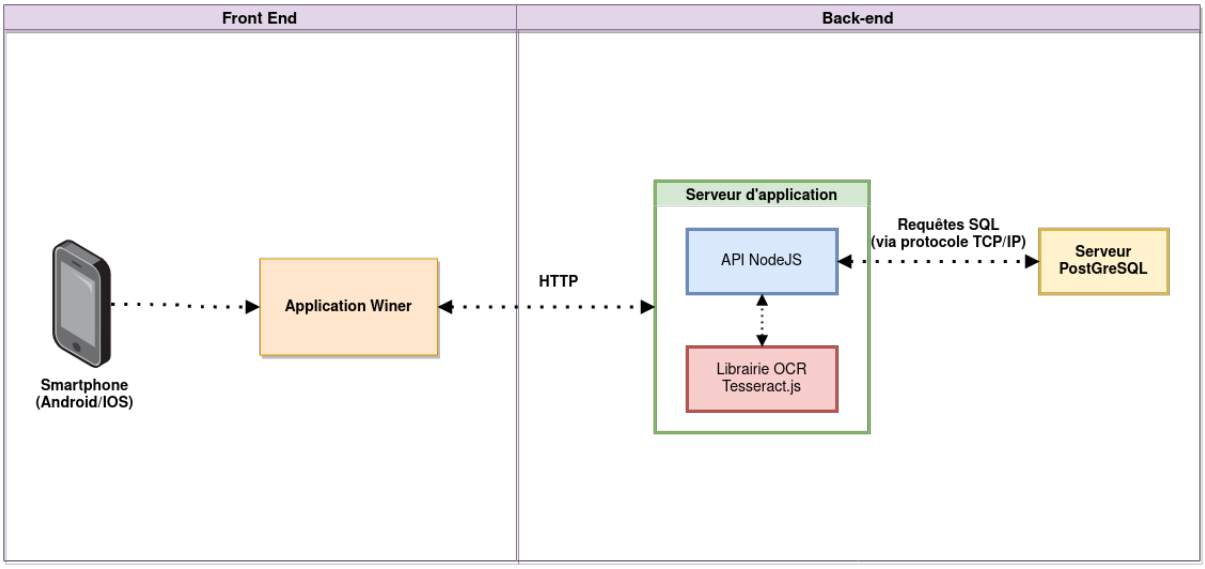


**Figure 6.** Cartographie des flux

| # | Depuis | Vers | Bilatéral ? | Support | Format | Transport | Description |
|---|---|---|---|---|---|---|---|
| 1 | Application | Serveur d'application | oui | Internet | HTML | HTTP | Communication client/serveur |
| 2 | Serveur d'application | Serveur base de données | oui | LAN | SQL | TCP/IP | Communication serveur/base de données |

**Table 3.** Matrice des flux de l'application

La communication entre le serveur d'application et le serveur de base de données PostgreSQL se fait via TCP/IP [4].

### 2.3.3 Schéma de la base de données

Ce diagramme [7] présente l'architecture de notre base de données.

Il y a 3 tables :

- la table `vin`, contenant les informations sur les vins, les colonnes sont :
  - `id` : l'id du vin (clé, unique)
  - `nom_vin` : nom du vin (VARCHAR : chaine de caractère)
  - `description` : description du vin (chaine de caractère)
  - `domaine` : domaine de production du vin (chaine de caractère)
  - `cepage` : cépage utilisé pour produire le vin (chaine de caractère)

- la table `avis`, contenant les avis des utilisateurs, les colonnes sont :
  - `identifiant` : l'identifiant (pseudonyme) de l'utilisateur qui a posté le commentaire (clé)
  - `note` : note attribuée par l'utilisateur (nombre entier de 1 à 5)
  - `vin` : id du vin
  - `avis` : avis sur le vin (chaine de caractère)

- la table `utilisateur`, contenant les informations sur les utilisateurs, les colonnes sont :

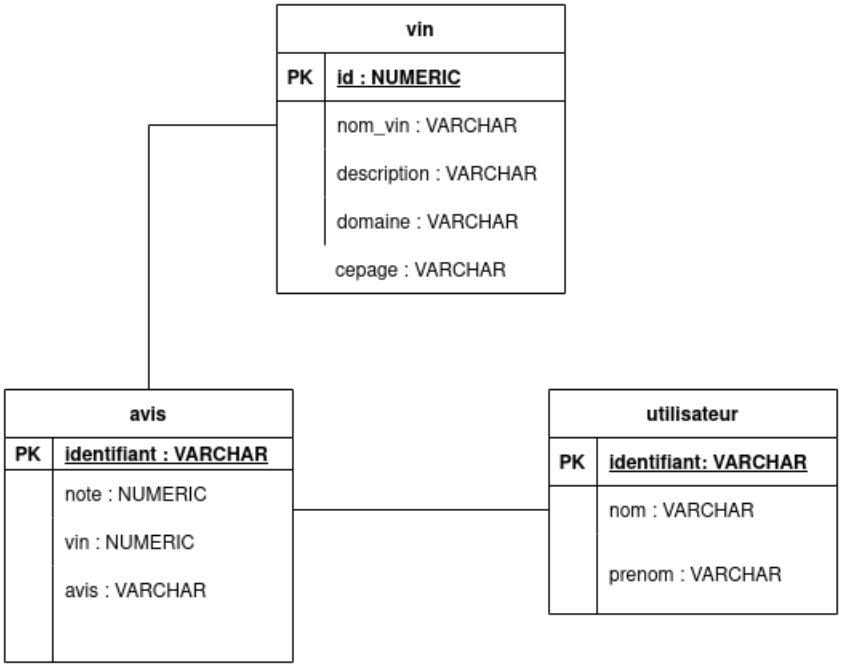


**Figure 7.** Diagramme de la base de données

- `identifant` : identifiant de l'utilisateur (clé, unique, chaine de caractère)
- `nom` : nom de l'utilisateur (chaine de caractère)
- `prenom` : prénom de l'utilisateur (chaine de caractère)

## 2.4 Interface Graphique

Enfin, avant de commencer le développement de l'application nous avons fait des schémas pour imaginer à quoi l'on voudrait qu'elle ressemble. La première image sera notre page d'accueil, la deuxième notre page de connexion, la troisième représente l'affichage d'un vin, la quatrième l'affichage des avis pour un vin et la cinquième l'affichage d'un profil utilisateur.


**Figure 8.** Schémas de l'interface graphique

# 3 Travail réalisé

Dans cette partie nous allons voir ce qui a donc effectivement été réalisé par rapport aux attentes.

## 3.1 Visuel

Tout d'abord, notre application a été développée pour Android et iOS grâce à `Ionic` mais n'a pas été déployée. Sur ces différentes pages, la navigation se fait simplement : il y a toujours une barre en bas de l'écran permettant de retourner en arrière ou bien de se connecter si on se trouve sur la page d'accueil. Quelques modifications ont été apportées par rapport aux schémas pour soucis d'esthétique mais tous les affichages sont présents.

### 3.1.1 Page d'accueil

lorsque le client lance l'application, il arrive sur la page d'accueil [9] qui lui permet directement de prendre une photo ou bien d'aller sur la page pour se connecter s'il ne l'est pas encore, aller voir son profil sinon.

**Figure 9.** Page d'accueil

Une fois la photo prise, on demande à l'utilisateur s'il veut bien analyser cette photo et si oui, on traite alors l'image et affiche le vin qui convient (données récupérées en brut, car il n'y a pas de communication avec le back-end).

### 3.1.2 Page d'affichage

Après l'analyse de l'image, nous sommes redirigés sur une page affichant les différentes informations du vin [10] :

- Le nom du vin
- Son image
- Le nombre d'étoiles (la note) qu'il a reçu.
- Sa description
- Son embouteillage
- Son cépage
- Le château où il a été fait
- Les avis des utilisateurs sur celui-ci
- Un bouton pour aller laisser un commentaire


**Figure 10.** Affichage des informations d'un vin

### 3.1.3 Laisser un commentaire

Sur la page pour laisser un commentaire [11], il est possible de mettre une note au vin en appuyant simplement sur le nombre d'étoiles souhaitées et d'écrire son commentaire si la personne veut en rajouter un, puis à appuyer sur le bouton "Enregistrez votre avis" (Cette fonctionnalité ne rajoute aucun avis, il n'y a pas de communication avec le back-end).

### 3.1.4 Page de connexion

La page de connexion [12] quant à elle demande à l'utilisateur un identifiant et un mot de passe pour se connecter. Actuellement, le mot de passe n'est pas vérifié, l'utilisateur se connecte directement avec l'identifiant rentré et possède des données écrites en brut qui sont stockées en local, ce qui permettra un affichage du profil.

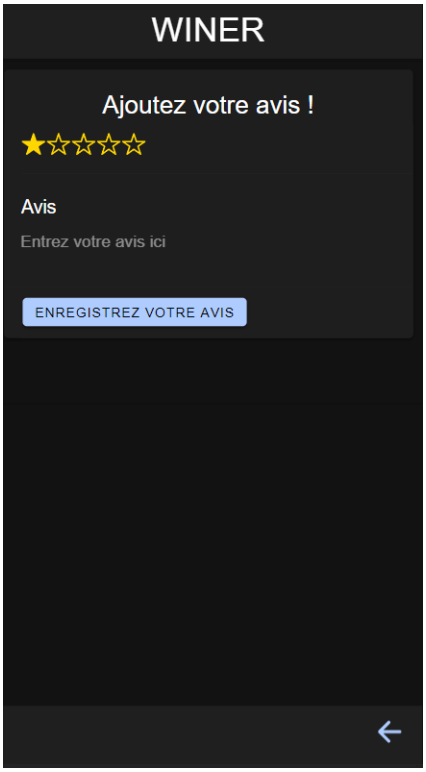


**Figure 11.** Page pour ajouter un avis

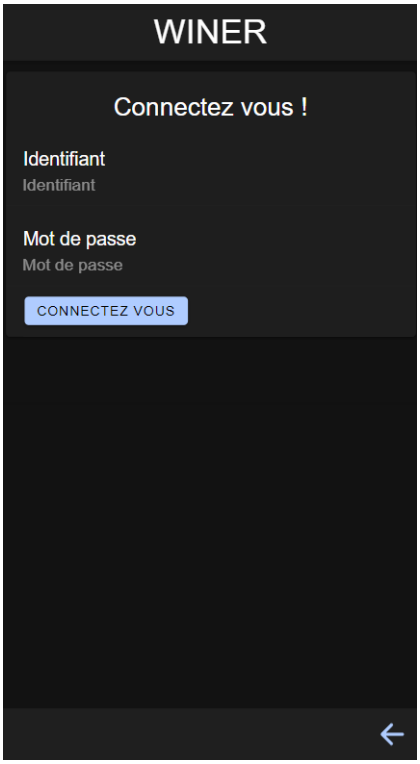


**Figure 12.** Page de connexion

### 3.1.5 Page de profil utilisateur

Enfin, sur la page d'affichage du profil utilisateur [13], il est possible de voir :

- Le nom de l'utilisateur

- Son vin préféré
- Son vin détesté
- Ses avis laissés
- Son historique de vins analysés
- Un bouton pour se déconnecter

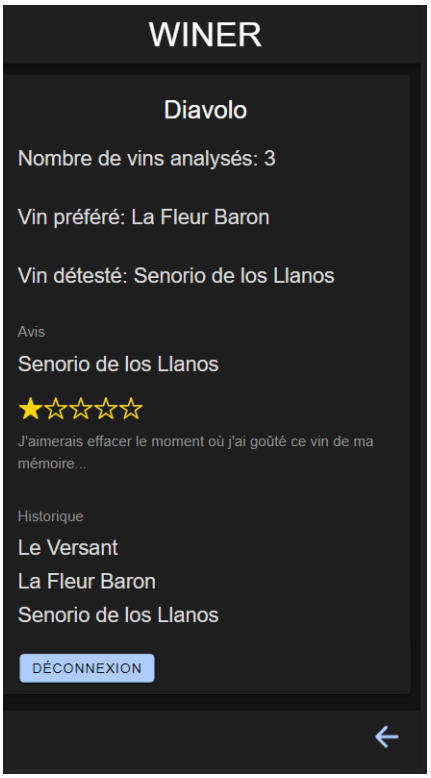


**Figure 13.** Page du profil utilisateur

Ces informations ne sont pas personnalisées par rapport à l'utilisateur hormis l'identifiant, ce sont des données écrites en brut stockées en local (recevoir une requête du back-end pour initialiser les données serait suffisant pour que cela soit fonctionnel).

## 3.2 Backend

### 3.2.1 API

L'api a été réalisé en `Node.js` comme prévu initialement, elle propose différents endpoints :

- `/analyse` : permet d'analyser une image de code barre d'étiquette envoyée par l'application, répond par les informations d'une bouteille, renvoi un fichier `JSON` contenant :
  - `nom_vin` : nom du vin
  - `description` : description du vin
  - `id` : id unique du vin
  - `cepage` : cépage du vin
  - `domaine` : domaine de production du vin

- `/avis` : En spécifiant l'id d'une bouteille, renvoi un fichier `JSON` contenant une liste d'objet `JSON` contenants :
  - `pseudo` : pseudonyme de l'utilisateur
  - `note` : sa note
  - `avis` : son avis

- `/utilisateur` : pour récupérer les informations correspondantes à l'ID d'un utilisateur, retourne un `JSON` contenant :
  - `nom` : nom de l'utilisateur
  - `prenom` : prénom de l'utilisateur
  - `pseudo` : l'identifiant de l'utilisateur
  - `avis` : une liste de ses avis sous forme d'objets `JSON`, chacun contenant :
    - `nom` : nom du vin
    - `note` : note du vin
    - `avis` : le commentaire de l'utilisateur
    - `cepage` : le cépage du vin
    - `domaine` : domaine de production du vin

#### 3.2.2 Reconnaissance d'image

Pour la reconnaissance de l'image, nous avons utilisé Tesseract avec son implémentation `JavaScript Tesseract.js` sur le backend en `NodeJS`, comme initialement prévu.

#### 3.2.3 Base de données

Pour la base de données, nous avons utilisé un conteneur Docker de `PostgreSQL` pour simplifier l'installation du logiciel, j'ai ensuite créé les tables décrites dans [2.3.3].

## 4 Conclusion

### 4.1 Pistes d'amélioration

Pour améliorer notre projet, nous aurions aimé finir la connexion entre l'application et l'API en back-end ce qui aurait rendu l'application totalement fonctionnelle (sans le côté administrateur que nous avons totalement oublié de réaliser, que ce soit côté front-end tout comme back-end ! Cela peut par exemple être dû à un manque d'organisation au niveau des tâches à réaliser)

#### 4.1.1 Coté application mobile

Pour améliorer l'application mobile, il aurait été possible de faire une Single Page Application (SPA) dans le but d'améliorer les performances de l'application (ne pas avoir à rechercher les pages à chaque fois). Il aurait également été possible grâce à l'idée précédente, de ne plus stocker certaines données en local (comme les vins que l'on analyse) car cela ne serait plus nécessaire.

#### 4.1.2 Côté test & montée en charge

Pour la partie, simulation, mesure et gestion de la montée en charge, nous aurions aimé mettre en place les solutions proposées avec l'utilisation de conteneurs docker géré par Kubernetes et rancher pour la partie API et Analyse d'image.

### 4.2 Le mot de la fin

#### 4.2.1 Sébastien

Ce projet était bien, j'ai appris à concevoir un projet depuis rien et utiliser de nouveaux outils tel que `Postman` pour simuler une requête `HTTP` sur l'API et renvoyer un fichier `JSON` contenant les informations.

Nos problèmes majeurs étaient le manque de communication et de clarté entre nous sur la fin du projet, c'est pourquoi nous n'avons pas pu combiner nos parties (application mobile et API) pour finaliser.

### 4.2.2 Guillaume

Globalement, ce projet était plaisant à réaliser, c'est la première fois que nous avions à voir la conception d'un projet de A à Z et j'ai appris beaucoup de choses par exemple les différents langages utilisables pour une application qui se veut cross-plateforme (Android et iOS).

M'étant occupé du développement visuel de l'application sous `Ionic`, j'ai également beaucoup appris à propos de ce langage spécifiquement et de la facilité qu'il y a à développer une application pour téléphone (tâche qui me paraissait complexe jusque-là).

En revanche, ayant commencé le développement de l'application tardivement, réunir le front-end et le back-end ne fut pas une tâche facile et a créé du stress dans notre groupe, ce qui était évidemment moins plaisant (d'autant plus que cela n'est finalement pas fonctionnel !). Il y a également eu un manque de communication à partir de ce moment-là pour s'ordonner. Ce que l'autre faisait n'était pas clair et cela a fini par rendre le projet pas totalement abouti.

## Références


[1] admin. *React Native vs Ionic vs Xamarin vs NativeScript : Comparing All Frameworks*. openwavecomp.com. 2021. url : https://www.openwavecomp.com/blog/react-native-vs-ionic-vs-xamarin-vs-nativescript-comparing-all-frameworks/.

[2] emilioSp. *NodeJS vs Apache performance battle*. dev.to. 2020. url : https://dev.to/emiliosp/nodejs-vs-apache-performance-battle-for-the-conquest-of-my-5c4n.

[3] OpenJS Foundation. *NodeJS*. OpenJS Foundation. 2021. url : https://github.com/nodejs/node.

[4] PostgreSQL Global Development Group. *Chapter 48. Frontend/Backend Protocol*. PostgreSQL Global Development Group. 2021. url : https://www.postgresql.org/docs/9.3/protocol.html.

[5] Vladyslav Holubiev. *OpenCV vs Tesseract OCR*. stackshare.io. 2019. url : https://stackshare.io/stackups/opencv-vs-tesseract-ocr.

[6] *How do I choose between Tesseract and OpenCV ?* StackOverflow. 2014. url : https://stackoverflow.com/a/11489853.

[7] Dickson Mwendia. *Top 6 Tools for API & Load Testing*. 2020. url : https://medium.com/@Dickson_Mwendia/top-6-tools-for-api-load-testing-7ff51d1ac1e8.

[8] Matt Netkow. *Ionic vs Flutter : Best Platform for Hybrid App Development*. Ionic. url : https://ionic.io/resources/articles/ionic-vs-flutter-comparison-guide.

[9] Richard Peterson. *13 BEST Free Database Software (SQL Databases List) in 2021*. guru99. 2021. url : https://www.guru99.com/free-database-software.html.

[10] Krunal Shah. *Ionic vs Flutter : How to make the right choice*. Third Rock Techkno. 2021. url : https://www.thirdrocktechkno.com/blog/ionic-vs-flutter-how-to-make-the-right-choice/#:~:text=Since%5C%20Ionic%5C%20is%5C%20built%5C%20on,not%5C%20be%5C%20your%5C%20ideal%5C%20choice.

[11] Promatics Technologies. *Here's who would win if Xamarin, Flutter and React Native Fight Out in 2021 ?* Promatics Technologies. 2020. url : https://promatics.medium.com/heres-who-would-win-if-xamarin-flutter-and-react-native-fight-out-in-2021-8fa6e22fdfbb.

[12] *TESSERACT.js*. Naphta. 2021. url : https://tesseract.projectnaptha.com/.

[13] . *React Native Vs. Xamarin Vs. Ionic Vs. Flutter:Which Is Best For Cross-Platform Mobile App Development ?* segmentfault. 2019. url : https://segmentfault.com/a/1190000018139911.