

Dossier de projet professionnel



Coté pote

Titre professionnel : Concepteur développeur d'application

Nom : Ziane

Prénom : Nadir

En Formation à La Plateforme_, 8 rue d'Hozier 13002 Marseille,

1. INTRODUCTION	4
1.1. PRÉSENTATION PERSONNELLE	4
1.2. RÉSUMÉ DU PROJET « Côté pote»	5
1.3. RÉSUMÉ EN ANGLAIS DU PROJET « Côté pote»	6
1.4. COMPÉTENCES COUVERTES PAR LE PROJET	7
2. CAHIER DES CHARGES	8
2.1. DÉFINITION DES BESOINS	8
2.1.1. LES OBJECTIFS DE L'APPLICATION	8
2.1.2. LES CIBLES	9
2.1.3. PÉRIMÈTRE DU PROJET	9
2.1.4. FONCTIONNALITÉS	9
2.1.5. CONTRAINTES TECHNIQUES	10
3. CONCEPTION	11
3.1. USER STORY	11
3.2. UX	12
3.2.1. USER EXPERIENCE	12
3.2.2. DÉFINITION DU PARCOURS UTILISATEUR	15
3.3. UI	16
3.3.1. "Côté Pote" & CHARTE GRAPHIQUE	16
3.3.2. MAQUETTE & PROTOTYPE	17
3.4. MODÉLISATION DE LA BASE DE DONNÉES	19
3.4.1. MÉTHODE MERISE	19
3.4.2. MODÈLE CONCEPTUEL DE DONNÉES (MCD)	19
3.4.3. MODÈLE LOGIQUE DE DONNÉES (MLD)	21
3.4.4. MODÈLE PHYSIQUE DE DONNÉES (MPD)	23
3.5. COLLABORATION A LA GESTION DE PROJET	24
3.5.1. OBJECTIFS	24
3.5.2. ORGANISATION	24
3.5.3. VERSIONING & WORKFLOW	26
3.5.1. GESTION DES DÉPENDANCES	27

4. ENVIRONNEMENT TECHNIQUE	29
4.1. TECHNOLOGIES & OUTILS	29
4.2. SPECIFICITES TECHNIQUES : BACKEND	29
4.2.1. API REST	29
4.2.2. SYMFONY / API PLATFORM	30
4.2.3. SYMFONY / API PLATFORM : INSTALLATION & CONFIGURATION	31
4.2.4. SYMFONY / API PLATFORM : CRÉATION D'UNE COLLECTION	35
4.2.5. SYMFONY / API PLATFORM : AJOUT DES RELATIONS	41
4.3. SPECIFICITES TECHNIQUES : FRONTEND	43
4.3.1. REACT NATIVE	43
4.3.2. EXPO	44
4.3.3. REACT NAVIGATION	46
4.3.4. AXIOS : INTERACTION AVEC LA BASE DE DONNÉES	48
5. PRÉSENTATION DU JEU D'ESSAI	50
5.1. PARCOURS UTILISATEUR POUR LA CONNEXION D'UN UTILISATEUR	50
6. EXTRAIT D'UNE RECHERCHE À PARTIR DE SITE	58

1.Introduction

1.1. Présentation personnelle

Je m'appelle Nadir Ziane, je suis actuellement concepteur / développeur web / web mobile à La Plateforme_.

Passionné par le monde de l'informatique depuis très jeune, après un long séjour linguistique de 2 ans en Espagne je me suis résolu à reprendre mes études pour pouvoir devenir développeur.

Ce choix a été mûrement réfléchi, depuis petit étant attiré par plusieurs formes d'arts (dessin, musique, piano, théâtre), j'ai toujours su que je voulais travailler dans le milieu créatif mais lequel, je ne savais pas.

Après avoir cherché comment allier l'utile à l'agréable, j'ai dressé un constat, j'aime l'informatique et la culture web en général, et que j'aime créer des choses :

Je voulais devenir développeur web.

Ainsi, j'ai commencé à me former seul sur des plateformes en ligne avant de commencer une formation à la Coding School au sein de la Plateforme. Après une première année de formation qui m'a permis d'obtenir le titre de « Développeur Web/Web Mobile », j'ai décidé de continuer sur une seconde année en alternance afin d'intégrer un environnement de professionnel et acquérir les compétences nécessaires à l'obtention du titre de « Concepteur/Développeur d'applications ».

1.2. Résumé du projet « Côté Pote »

Pendant nos périodes de formation, du temps a été consacré au développement d'une application mobile, J'ai décidé de partir sur une application de paris entre amis que j'ai appelé Côté Pote.

Côté Pote est une application qui permet à un utilisateur ou un groupe d'utilisateurs de publier des paris de n'importe quel type, que ce soit sportif, musical, jeux-vidéo...

L'objectif est de mettre en place une application permettant à des groupes d'amis ou utilisateur solo sans groupe d'amis d'avoir la possibilité de lancer des paris.

L'utilisateur pourra retrouver chaque paris postée grâce à un feed qui permettra de consulter tous les paris.

Les utilisateurs peuvent à chaque type d'événement lancer des paris

L'application s'inspire à la fois des applications de paris traditionnels et des applications de soirée entre amis.

Je voulais une application ludique, avec les mêmes codes que les applications de soirée entre amis ou app de paris sportifs sur lesquels nous nous rendons souvent, tout en apportant une plus-value.

1.3. Summary of the "Côté Pote" project in English.

Côté Pote is an application that allows a user or a group of users to publish bets of any kind, be it sports, music, video games...

The goal is to create an application where groups of friends or solo users without a group of friends can create bets.

The user will be able to find each bet posted on a feed where all bets can be viewed.

Users can place bets on any type of event (world cup, gaming competitions, etc.).

The application is inspired by both traditional betting applications and party applications.

I wanted a fun application, with the same codes as the party app or sports betting app, but with an added value.

The application is developed in React Native for the front-end and Symfony / API Platform for the back-end.

1.4. Compétences couvertes par le projet

Voici les compétences du référentiel couvertes par le développement réalisé sur ce projet.

- **Activité type 1:** Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité
 - Maquetter une application
 - Développer des composants d'accès aux données
- **Activité type 2:** Concevoir et développer la persistance des données en intégrant les recommandations de sécurité
 - Concevoir une base de données
 - Mettre en place une base de données
 - Développer des composants dans le langage d'une base de données
- **Activité type 3:** Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité
 - 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
 - Préparer et exécuter le déploiement d'une application

2. Cahier des charges

Avant de passer à la réalisation de la maquette ou une quelconque étape de développement, nous avons travaillé avec mon groupe qui était composé de 4 personnes, entre autres, Robin Papazian, Robin Arbona, Pierre Malardier et moi-même, à l'élaboration d'un cahier des charges afin de cadrer le projet, lister les fonctionnalités et se donner des objectifs.

2.1. Définition des besoins

2.1.1. Les objectifs de l'application

- Les objectifs qualitatifs**

L'objectif est de mettre en place une appli permettant à des groupes d'amis ou utilisateur solo sans groupe d'amis d'avoir la possibilité de lancer des paris (défis peut-être par la suite).

Prévue pour être utilisée uniquement sur une app, un travail en amont sera réalisé pour que l'app soit compatible sur tous les types d'OS que ce soit Android ou IOS.

- Les objectifs quantitatifs**

L'application devra être capable de stocker les informations de chaque utilisateur de manière sécurisée.

L'identité visuelle doit être accrochante et dans l'air du temps avec un Flat Design.

La navigation doit être naturelle et intuitive et une attention particulière doit être portée sur l'UI/UX de manière générale pour pouvoir accrocher l'attention d'un public large.

2.1.2. Les cibles

L'application sera ouverte à toute personne souhaitant s'amuser avec ses amis dans le cadre de paris de toute sorte ce qui donne une liberté et permet de toucher un large spectre d'utilisateurs peu importe la provenance sociale ou le type de loisir visée.

La cible comprend toute personne qui cherche à identifier des changements d'humeur, des périodes de bien-être ou d'autres phases émotionnelles passagère...

sur l'application il y aura que des utilisateurs inscrits, ce qui comprend les utilisateurs qui pourront participer à des paris, voir l'historique de tous les paris auxquelles ils ont participés, agrémenter ou modifier leurs profil (image, nom, prénom, date de naissance, num de tel, mail), avoir accès au défis de la semaine, créer des groupe de paris, invité des amis dans le ou les différent groupe de paris auxquels il participe.

2.1.3.Périmètre du projet

L'app sera disponible uniquement en français.

Par ailleurs, il devra être adapté à tous les OS (Android, IOS) pour permettre à tous les utilisateurs d'avoir une expérience d'utilisation optimale.

Celle-ci pourra comprendre l'utilisation de fonctionnalités natives des appareils mobiles, telles que l'appareil photo ou la géolocalisation.

Les données personnelles des utilisateurs seront stockées dans une base de données.

2.1.4.Fonctionnalités

- L'app comprend une page d'accueil à partir de laquelle nous pouvons nous connecter/inscrire.

- Il sera également possible de consulter le fonctionnement de l'app après inscription de l'utilisateur.
-
- L'app comportera un espace utilisateur sécurisé par un mot de passe et hashé dans la base de données (en accord avec la réglementation du RGPD).
- Pour les personnes souhaitant modifier leurs informations, il sera mis à disposition sur l'app une page profil.
- Il y aura la possibilité de modifier, ajouter, supprimer un pari
- Il y aura la possibilité de participer à un pari
- il y aura la possibilité de rechercher un pari en particulier

2.1.5. Contraintes techniques

L'application devant être compatible de manière native avec les systèmes d'exploitation mobile iOS et Android, plusieurs choix techniques ont été fait pour ce choix.

Côté Front, nous avons décidé d'utiliser la librairie Expo React Native. Celle-ci permet de créer des applications mobiles multiplateformes sans avoir à apprendre les langages de chaque système, elle offre un gain de temps et de productivité énorme tout en offrant une certaine ergonomie.

La base de données et l'API qui permettent les requêtes HTTP devront être sécurisées.

Les mots de passe de l'application devront être hashé en base de données et tout utilisateur pourra récupérer ou voir supprimer ses données à tout moment.

3. Conception

Afin de mieux appréhender nos étapes de conception et d'intégration, nous avons fait le choix d'utiliser une approche AGILE, celle-ci nous a paru idéale pour optimiser au mieux notre travail.

3.1. User Story

Avant de réaliser les étapes de conception concernant l'UX et l'UI, nous avons fait de la veille et testé plusieurs applications mobiles abordant le même thème comme winamax, picolo, betclic...

Cette démarche nous a permis de nous mettre à la place d'un utilisateur et de définir les fonctionnalités de l'application.

Les « User Stories » sont utilisées pour imaginer les besoins des utilisateurs, elles permettent de voir les caractéristiques, les fonctions et les exigences de l'application à créer.

Elles permettent d'appréhender la nécessité d'une fonctionnalité sur l'application et aide à faire comprendre l'utilité de la fonctionnalité et sa priorité.

Les user stories ont ainsi permis à notre équipe d'établir une arborescence représentant le parcours utilisateur.

3.2. UX

3.2.1. User Experience

Les premières minutes d'utilisation sont cruciales pour assurer l'évolution et l'utilisation d'une application sur la durée.

L'expérience de l'application doit être agréable intuitive ainsi que tape à l'œil avec un esprit jeune et frais.

L'UX (User Experience) appliquée au mobile consiste en la conception d'applications à même de faire vivre une expérience ludique, drôle et humaine à l'utilisateur dans notre cas.

L'UX a été ainsi mobilisée au début du projet, lors de l'élaboration de la stratégie pour :

- Faciliter l'utilisation de l'application et son fonctionnement
- Contribuer à l'amusement de l'utilisateur dans la durée en faisant appel à sa créativité lors de la création de son pari
- Permettre un usage sur différents appareils et dans des contextes spécifiques (en soirée, en compétition...)

Pour savoir comment optimiser notre application mobile et apporter une expérience plaisante pour l'utilisateur, il a fallu effectuer des recherches et porter notre attention sur les points suivants :

- Limiter les fonctionnalités au minimum puisque le but de l'application est l'amusement sans forcément compliquer la tâche de l'utilisateur.

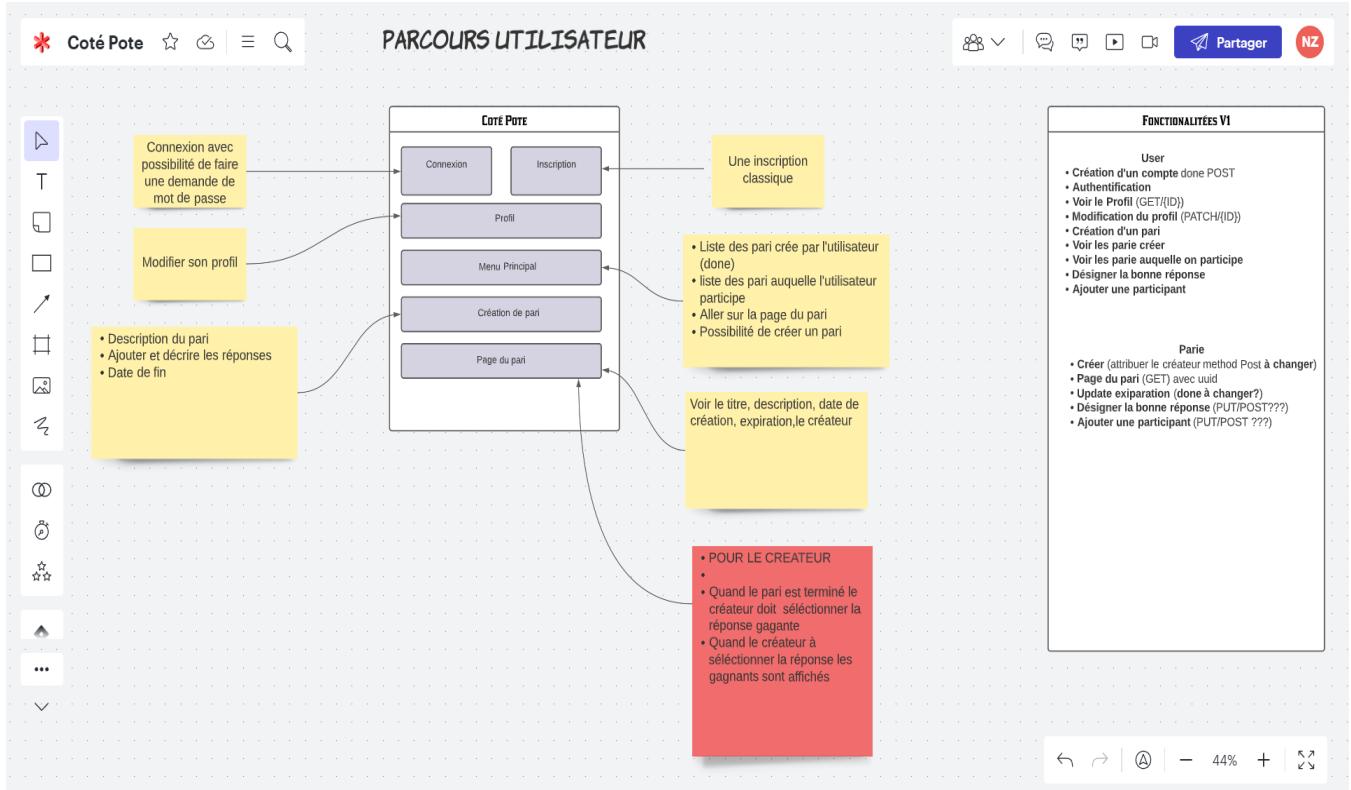
- Simplifier le contenu au maximum, donc proposer une application fluide et rapide est un objectif majeur.
- Guider l'utilisateur par le design Afin d'aider l'utilisateur dans sa prise de décision, le travail autour des couleurs et des formes est important dans le but de hiérarchiser l'information et inciter à l'action est nécessaire. La charte graphique doit être déclinée sur l'ensemble de l'application afin d'être reconnaissable, rassurer l'utilisateur et obtenir un rendu harmonieux.
- Limiter le travail de l'utilisateur de par son format et les situations dans lesquelles il est utilisé, manipuler un smartphone n'est pas toujours une tâche facile.
Il est nécessaire de diminuer le plus possible le nombre d'actions à entreprendre côté utilisateur en limitant les étapes, supprimant un maximum de champs dans les formulaires, utilisant les données déjà fournies par l'utilisateur, affichant les bons claviers au bon moment, utilisant les fonctionnalités natives du téléphone tel que l'appareil photo etc.
- Travailler l'ergonomie, donc ne pas oublier que les utilisateurs ne sont pas toujours dans des conditions optimales de navigation.
Chaque élément doit donc être de taille raisonnable et atteignable à une main.
- Ne pas obliger à une quelconque action, réfléchir à l'usage et à la récurrence de l'envoi des notifications « Push ».
- Prendre en compte la diversité humaine et concevoir des produits et services numériques ouverts à toutes et tous et qui ne nécessitent pas d'adaptation de la part des utilisateurs.

- Des conventions existent pour le mobile et les utilisateurs s'attendent à les retrouver dans l'application comme par exemple une «Tab Bar» pour la navigation, un « Burger Menu » plutôt que menu classique...
- Maximiser les chances de pérennisation de l'application en proposant le bon contenu au bon moment, le degré de personnalisation à implémenter est à estimer suivant les fonctionnalités du projet et des ressources.
- Les stores d'application possèdent leurs propres algorithmes de positionnement. L'ASO ou « AppStore Optimization » est un travail à part entière, rédiger de bonnes descriptions qui tapent à l'œil, sélectionner la bonne catégorie, ajouter des captures d'écran attrayantes, représentatives des fonctionnalités, obtenir de bons avis avec de bonnes notes.

3.2.2. Définition du parcours utilisateur

Pour définir le parcours utilisateur, nous avons identifié les étapes clefs de la navigation, et traduit à partir de celle-ci les possibilités et actions menant aux différentes pages de l'application.

À partir de cette réflexion, nous avons modélisé un schéma représentant l'ensemble du parcours utilisateur et des fonctionnalités disponibles.



3.3. UI

3.3.1. Côté Pote & charte graphique

Pour réaliser la maquette, nous avons d'abord constitué une sélection de designs inspirés d'applications mobiles existantes, entre autres du benchmarking.

Cela nous permet d'avoir plusieurs idées de ce vers quoi l'on pourrait se diriger pour réaliser la charte graphique et le design de l'application.

L'élaboration de la charte graphique, celle-ci reprend :

- Un logo visible sur l'écran d'accueil
- Une liste des polices à importer et utiliser sur l'ensemble de l'application
- Les couleurs principales de l'application
- Des icônes, formes et couleurs de boutons





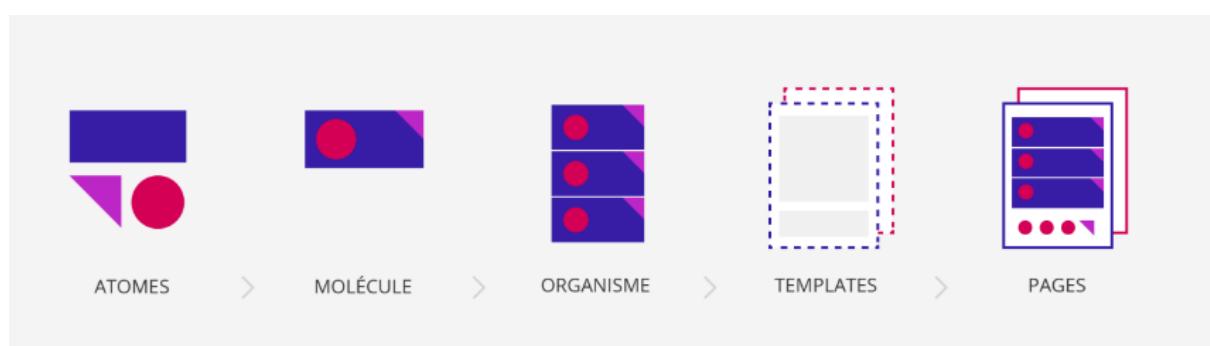
3.3.2.Maquette & Prototype

Pour la création de la maquette, nous avons fait le choix d'adopter une approche de création selon les principes de l'Atomic Design.

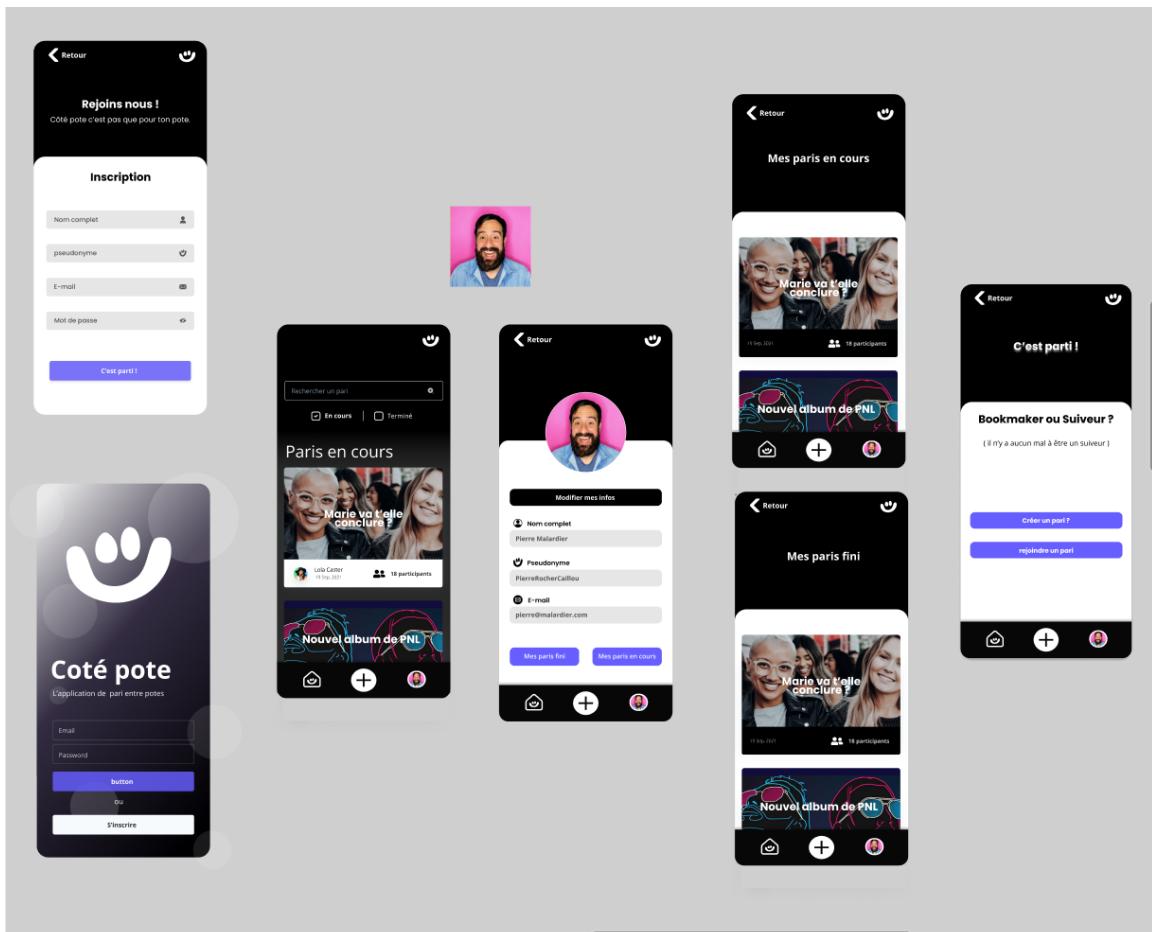
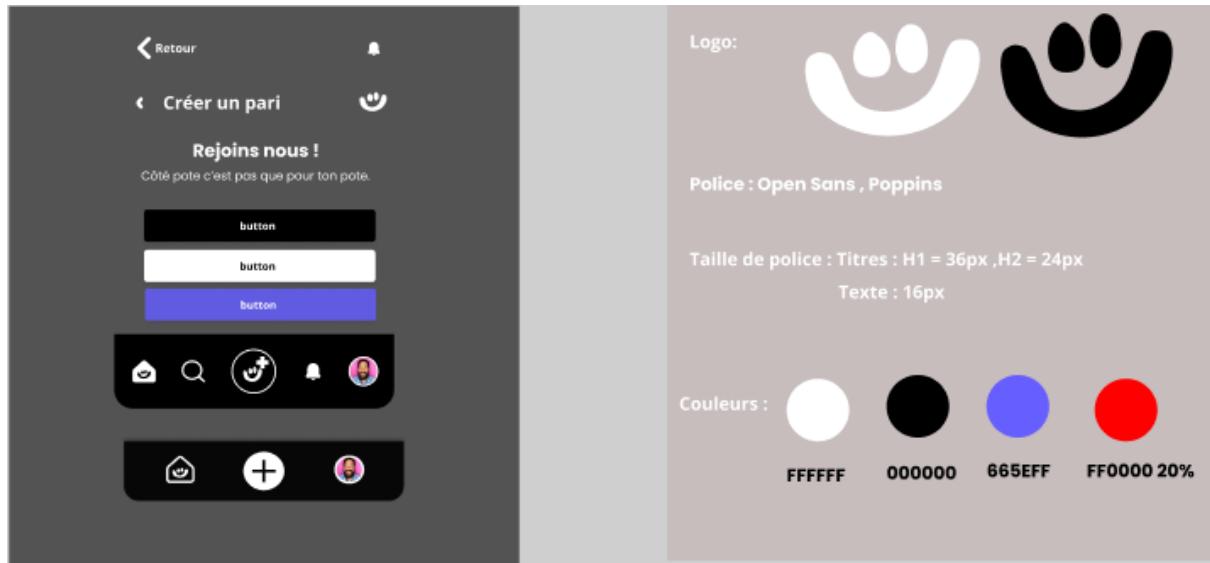
Cette méthodologie de conception d'interfaces graphiques permet de réduire nos composants à de plus petites briques réutilisables, lesquelles peuvent ensuite être assemblées pour constituer des modules de plus grande taille.

Tous ces types de composants ont un but précis, et combiné entre eux, permettent de constituer un ensemble de pages cohérent et facilement maintenable de par la modularité qu'apporte ce type de conception.

C'est un concept que l'on peut également comparer à la programmation orientée objet et qui dans notre cas, est justifié par l'utilisation de la librairie React Native adoptant un design pattern basé sur la création d'application par composant.



Pour reprendre les principes de l'Atomic Design nous avons créé un Kit UI. C'est un ensemble d'éléments graphiques, basés sur la hiérarchie citée précédemment (atomes, molécules, organismes) et qui seront réutilisés pour créer chaque page.



3.4. Modélisation de la base de données

3.4.1. Méthode MERISE

MERISE (Méthode d'Étude et de Réalisation Informatique pour les Systèmes d'Entreprise). Née à la fin des années 1970 en France, elle a pour objectif de définir une démarche permettant la modélisation et la conception de S.I. Parmi les ressources informatiques de ces S.I., figurent en particulier les fichiers de données, bases de données et système de gestion de bases de données (S.G.B.D.).

C'est sur ce dernier point que la méthode MERISE nous a été utile, car c'est en se basant sur ses principes de modélisation que nous avons conçu la base de données de Côté Pote.

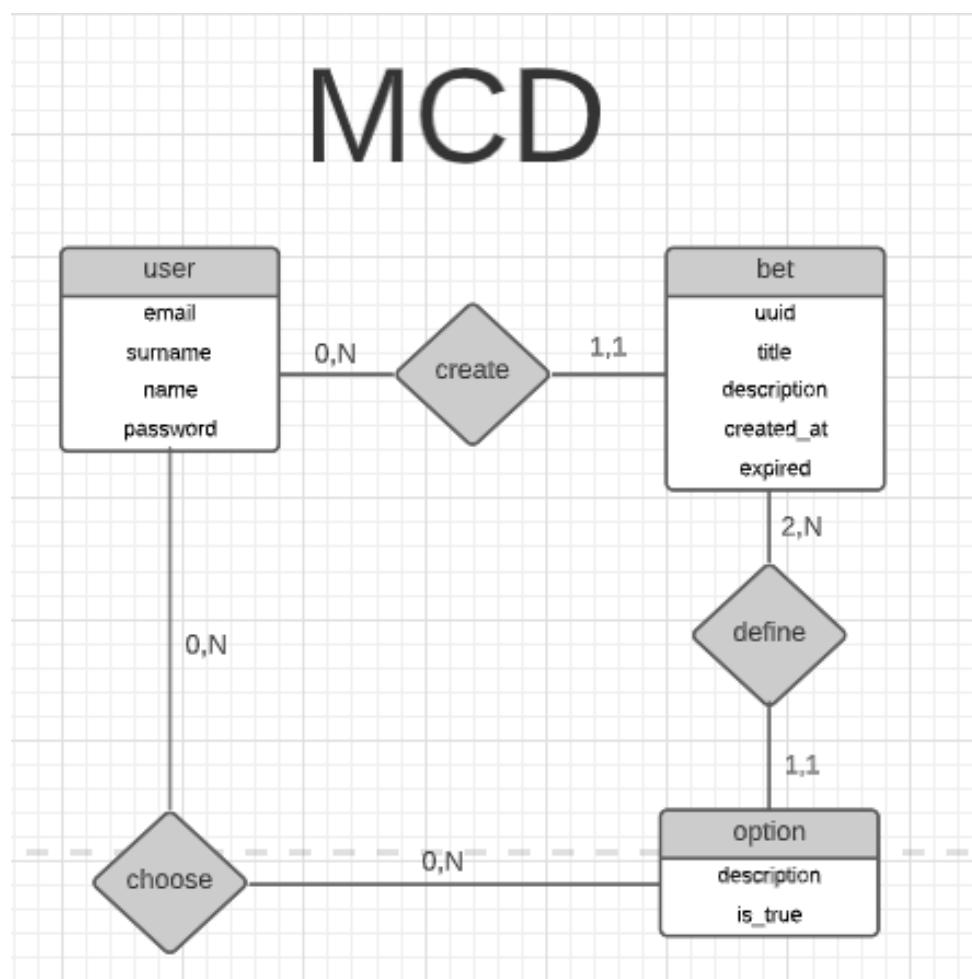
3.4.2. Modèle conceptuel de données (MCD)

c'est un modèle simplifié de la base de données, où les tables sont seulement au stade d'entités (c'est l'équivalent de la table en MCD), entre son second nom de schéma Entité/Association.

D'abord il a fallu imaginer tous les besoins des futurs utilisateurs de Côté Pote. A partir de ces besoins, j'ai été en mesure d'établir les règles de gestion des données à conserver.

Ensuite il faut définir le dictionnaire des données, c'est-à-dire toutes les données élémentaires qui vont être conservées en base de données et définir certaines caractéristiques qui figureront dans le MCD. Parmi ces caractéristiques, on retrouve par exemple la référence d'une donnée et notamment un identifiant unique, sa désignation, son type etc...

Étape finale à sa conception, on a pu à partir des informations précédemment recueillies, créer chaque entité, unique et décrite par un ensemble de propriétés et leur associations permettant de définir les liens et cardinalités entre les entités.



3.4.3.Modèle Logique de Données (MLD)

Le modèle logique de données (MLD), est une étape intermédiaire entre le modèle conceptuel de données et le modèle physique de données.

Le passage d'un MCD en MLD s'effectue selon quelques règles de conversion précise :

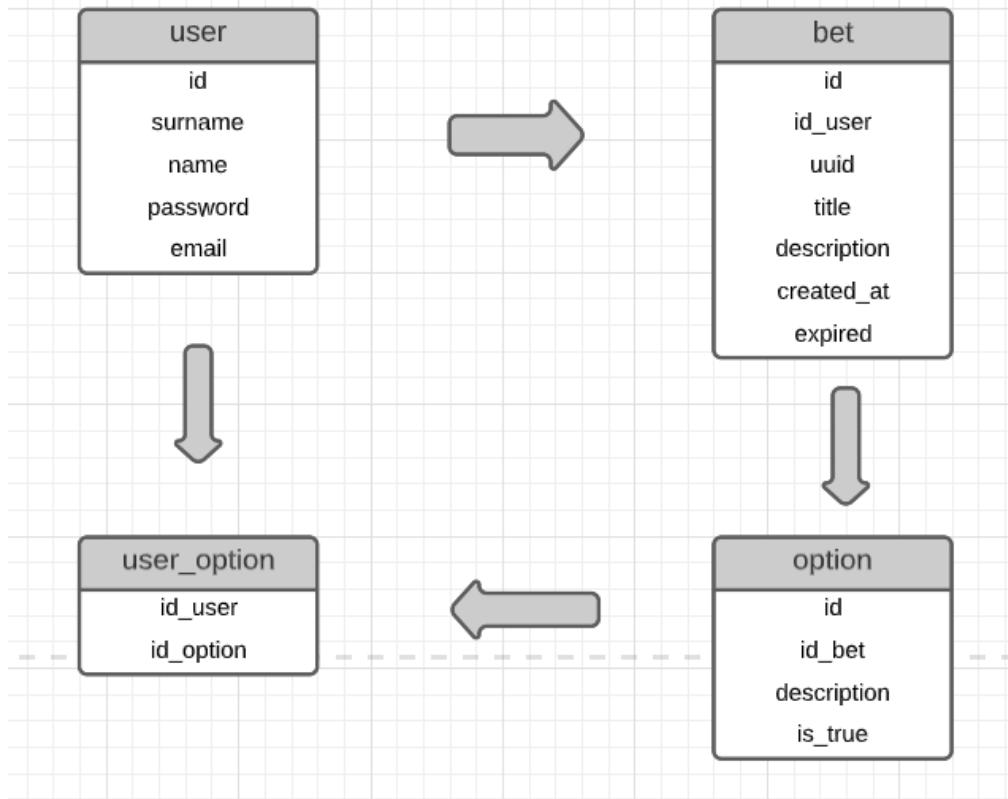
Une entité du MCD devient une table. Dans un SGBD de type relationnel, une table est une structure tabulaire dont chaque ligne correspond aux données d'un objet enregistré et où chaque colonne correspond à une propriété de cet objet.

Ces colonnes font notamment référence aux caractéristiques définies dans le dictionnaire de données du MCD.

Les identifiants respectifs de chaque entité deviennent des clefs primaires et toutes les autres propriétés définies dans le MCD deviennent des attributs. Les clefs primaires permettent d'identifier de façon unique chaque enregistrement dans une table et ne peuvent pas avoir de valeur nulle.

Les cardinalités de type "0:n" / "1:n" sont représentées à travers le référencement de la clef primaire de la table qui la possède, en clef étrangère au sein de la table auquelle elle est liée. Si deux tables liées possèdent une cardinalité de type "0:n / 1:n", la relation sera traduite par la création d'une table de jonction, dont la clef primaire de chaque table deviendra une clef étrangère au sein de la table de jonction.

MLD

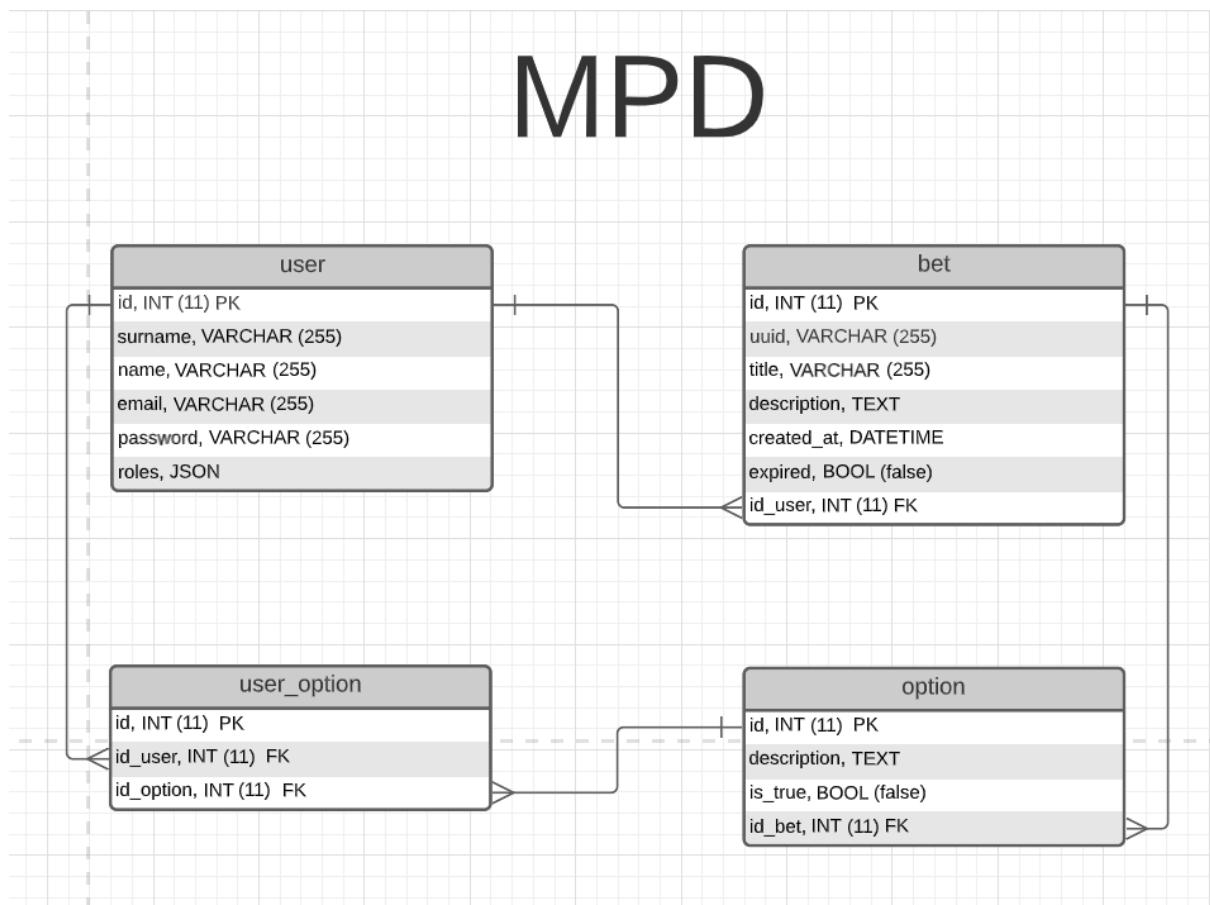


Ici, nous voyons bien que les clés primaires de chaque table associée deviennent des clés étrangères dans chacune des tables qui reçoivent les données.

3.4.4. Modèle Physique de Données (MPD)

Étant presque une formalité, le MPD consiste à l'implémentation des modèles générés précédemment dans le Système de Gestion de Base de Données (SGBD) utilisé.

Dans notre cas, ce sera le SGBD MySQL couplé au moteur de stockage InnoDB qui permet la gestion des contraintes des clefs étrangères en base de données.



3.5. Collaboration à la gestion de projet

3.5.1. Objectifs

Avant de démarrer le développement, il a été nécessaire de définir des objectifs clairs en termes de livrables.

Nous avons décidé de produire une version bêta de l'application. Cette version présente les fonctionnalités principales afin de tester l'essentiel des fonctionnalités auprès d'un panel d'utilisateurs et de prendre des décisions sur l'évolution du projet en fonction de leurs retours.

Nous avons choisi de présenter une application sur laquelle un utilisateur peut s'inscrire, se connecter, ajouter un pari, participer à un pari. La page profil permet de consulter ses informations et les modifier. Une Tab Bar en bas de l'application permet de naviguer de manière fluide entre les différentes interfaces.

Pour atteindre ces objectifs dans un temps limité, c'est-à-dire pendant le rythme de l'alternance, nous avons fait des choix en termes d'organisation, de technologies et d'outils.

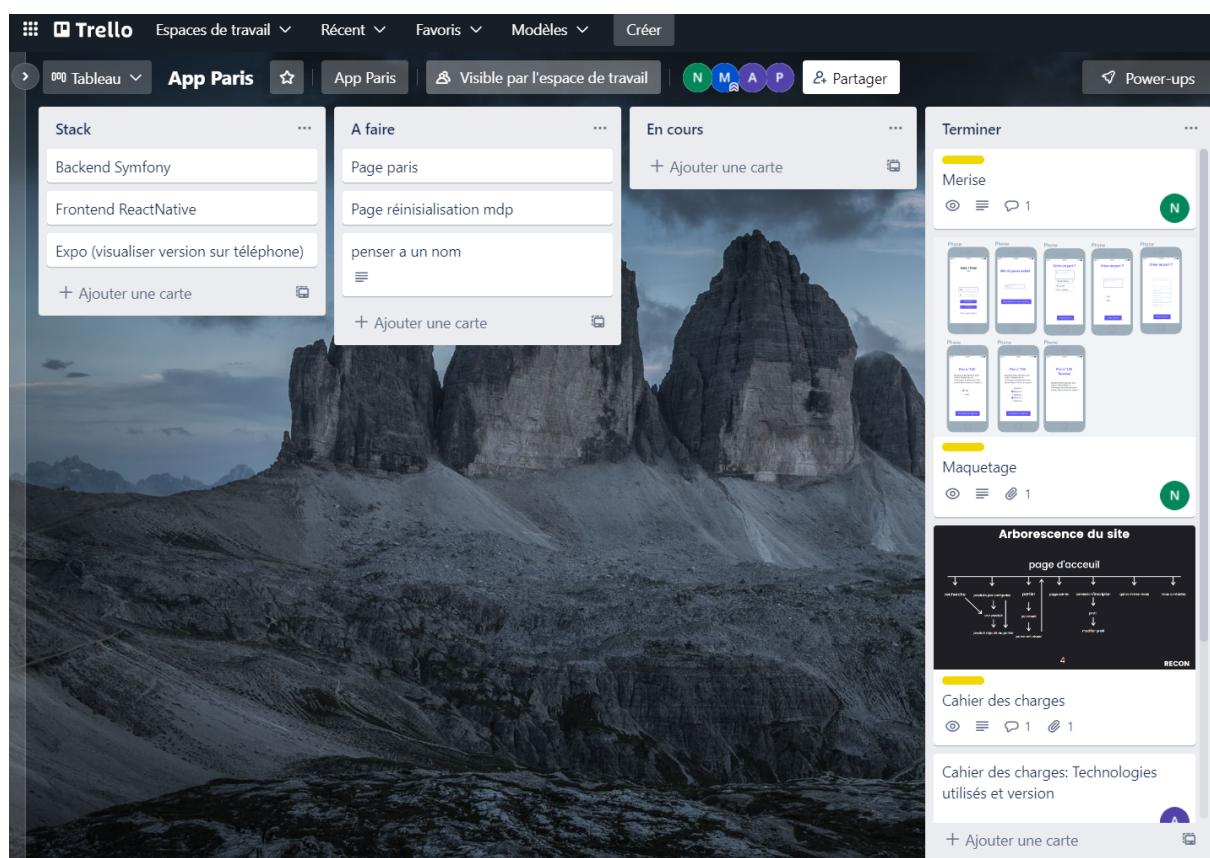
3.5.2. Organisation

En vue d'optimiser notre gestion de projet et le travail d'équipe, nous avons utilisé divers outils de travail collaboratif permettant de répartir les tâches de développement.

Concernant la gestion des tâches, nous avons utilisé l'outil de gestion de projet Trello, il offre la possibilité de lister les tâches à accomplir et suivre leurs avancements par les différents membres de l'équipe. Les utilisateurs ont accès à un tableau sous forme de cartes assignables et mobiles d'une planche à l'autre, traduisant leur avancement.

Pour notre cas, plusieurs contraintes étaient à prendre en compte concernant les délais de réalisation, le rythme de l'alternance ne nous permettant d'accorder que très peu de notre temps au projet, nous avons fait le choix de ne pas se donner de délais impartis concernant la réalisation des tâches.

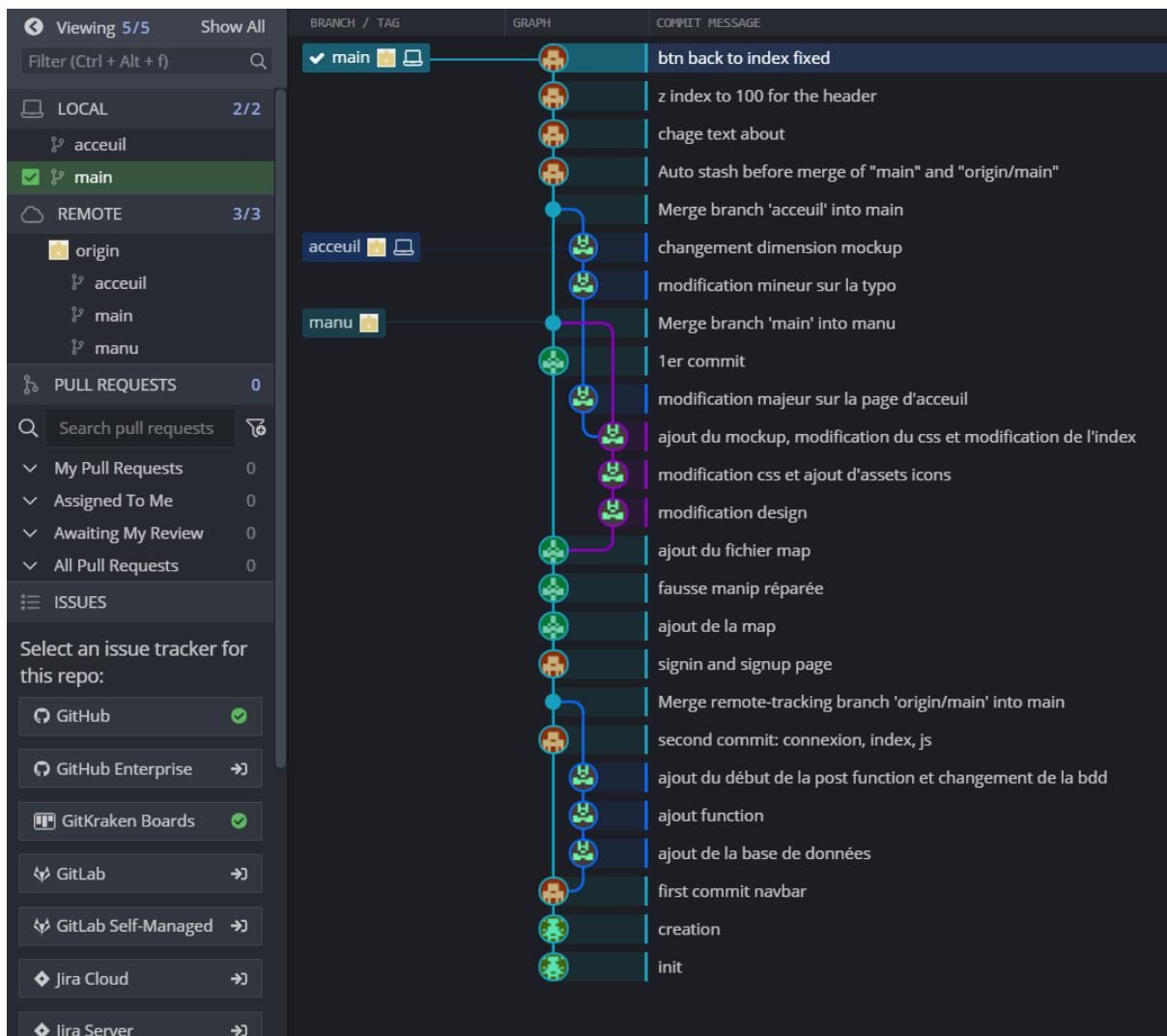
Pour s'adapter à notre rythme, nous avons ainsi tenté de suivre une approche « AGILE » pour collaborer et atteindre nos objectifs de développement.



3.5.3.Versionning & Workflow

Pour gérer les différentes modifications apportées dans le code et sauvegarder l'ensemble du travail, pour ma part, j'ai utilisé le logiciel de versionning Git Kraken, associé à GitHub.

Git Kraken offre une interface graphique de gestion et de visualisation des actions opérées sur Git et permet une vue d'ensemble des différentes branches, commits, etc.



Pour organiser les différentes parties de l'application j'ai créé deux repositories :

- Un repository Backend qui contient la partie API développée avec Symfony / API Platform
- Un repository Frontend qui contient l'interface développée avec la EXPO React Native

Sur les repositories Frontend et Backend, nous avons choisi la méthode de travail GitFlow.

GitFlow permet de poser un cadre en standardisant la création de branches et ainsi modéliser un workflow selon le type de tâche que l'on va faire.

- Branches principales :
 - Main : cette branche représente l'état du projet en production.
- Branches secondaires :
 - feature : chaque nouvelle fonctionnalité sera développée sur une branche feature. La création d'une branche feature se fera toujours depuis la version la plus récente de la branche main et sera préfixée de cette manière

3.5.1. Gestion des dépendances

Parmi les différentes fonctionnalités développées, certaines dépendent de l'utilisation de librairies externes.
Ces librairies permettent l'implémentation de fonctionnalités déjà créées par d'autres développeurs et nous évitent de perdre du temps.

Voici quelques exemples de librairies utilisées dans le projet :

- Axios qui est un Client HTTP simplifiant les requêtes en BDD
- React Navigation qui permet le routing et le partage de données entre plusieurs écrans

Pour gérer ces dépendances, nous utilisons l'outil Yarn. Ce gestionnaire pour Javascript permet à partir d'un fichier au format JSON "package.json" de répertorier tous les modules nécessaires au projet et leurs versions. Grâce à cela, nous pouvons facilement les installer et les maintenir à jour.



The screenshot shows a code editor window with the file "package.json" open. The file contains the configuration for a React Native project named "app-cotepote". It includes details about the project version, main entry point, scripts for running on different platforms, and a comprehensive list of dependencies from various npm packages. The code is color-coded for readability, with syntax highlighting for JSON keys and values.

```
1  {
2    "name": "app-cotepote",
3    "version": "1.0.0",
4    "main": "node_modules/expo/AppEntry.js",
5    // (débogage)
6    "scripts": {
7      "start": "expo start",
8      "android": "expo start --android",
9      "ios": "expo start --ios",
10     "web": "expo start --web",
11     "eject": "expo eject"
12   },
13   "dependencies": {
14     "@expo-google-fonts/dev": "^0.2.2",
15     "@expo-google-fonts/poppins": "^0.2.2",
16     "@react-native-async-storage/async-storage": "^1.15.17",
17     "@react-native-community/masked-view": "^0.1.11",
18     "@react-navigation/bottom-tabs": "^6.0.9",
19     "@react-navigation/drawer": "^6.3.1",
20     "@react-navigation/native": "^6.0.6",
21     "@react-navigation/stack": "^6.0.11",
22     "axios": "^0.25.0",
23     "buffer": "^6.0.3",
24     "expo": "~44.0.0",
25     "expo-app-loading": "~1.3.0",
26     "expo-constants": "~13.0.1",
27     "expo-font": "~10.0.4",
28     "expo-google-fonts": "~0.0.0",
29     "expo-image-picker": "~12.0.1",
30     "expo-secure-store": "~11.1.0",
31     "expo-status-bar": "~1.2.0",
32     "react": "17.0.1",
33     "react-dom": "17.0.1",
34     "react-native": "0.64.3",
35     "react-native-gesture-handler": "~2.1.0",
36     "react-native-keyboard-aware-scroll-view": "^0.9.5",
37     "react-native-reanimated": "2.3.1",
38     "react-native-safe-area-context": "3.3.2",
39     "react-native-screens": "~3.10.1",
40     "react-native-web": "0.17.1",
41     "save": "^2.4.0"
42   },
43   "devDependencies": {}
```

4. Environnement Technique

4.1. Technologies & Outils

4.2. Spécificités techniques : Backend

4.2.1. API REST

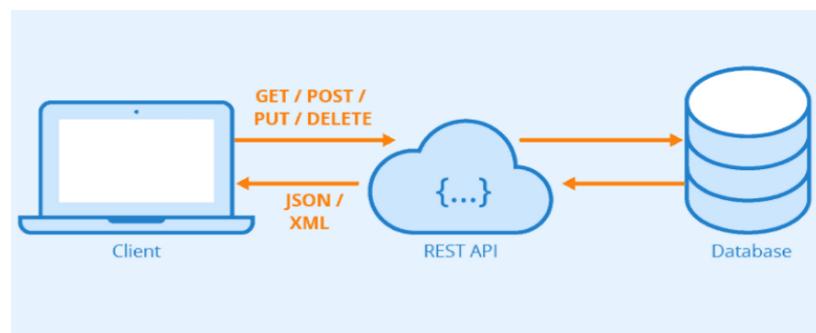
Pour l'application mobile, nous avons fait le choix concernant la gestion des données, de créer API, API Application Programming Interface ».

Pour faire simple, une API est une interface qui permet d'échanger des données, de façon réutilisable et standardisée, entre différents composants d'une application ou entre une application et d'autres développeurs sans dépendre d'un langage en particulier.

Chaque donnée, sont regroupées au sein d'une « Collection » et peuvent être consultées par le client sous différents formats de données comme le JSON.

Plusieurs architectures d'API existent. Parmi elles, on retrouve l'architecture REST, Abréviation de Representational State Transfer, c'est le modèle le plus utilisé pour la création d'API et c'est cela que nous avons choisi.

Elle met à profit l'utilisation des verbes http (GET, POST, PUT, DELETE, etc.) et permet une compréhension facile des différentes actions possibles sur une API.



La Client-serveur séparation, Cette séparation permet au client de s'occuper uniquement de la récupération et de l'affichage de l'information et permet au serveur de se concentrer sur le stockage et la manipulation des données.

Une mise en cache des données est nécessaire et permet de réduire le nombre de fois où un client et un serveur doivent interagir.

Il en résulte une accélération des temps de chargement pour l'utilisateur car pour une même requête exécutée plusieurs fois, le client pourra récupérer tout ou une partie des données directement sur la mémoire cache de sa machine / mobile sans avoir besoin de requêter le serveur.

4.2.2. Symfony / API Platform

Pour mener à bien le développement de notre back-end, nous avons fait de la veille afin de déterminer quelle solution serait la plus pertinente pour mettre en place notre API en fonction de nos contraintes. Notre choix s'est tourné vers l'utilisation de Symfony / API Platform

Tout d'abord, API Platform est un framework web utilisé pour générer des API REST, se basant sur le patron de conception MVC dit Modèle, Vue, Contrôleur.

La partie serveur du framework est écrite en PHP et basée sur le framework Symfony, tandis que la partie client est écrite en JavaScript et TypeScript

Doctrine ORM qui est un système de persistance est automatiquement livré dans la distribution API Platform.

Entre autres, Doctrine ORM est le moyen le plus simple de persister et d'interroger des données dans un projet API Platform grâce au pont fourni avec la distribution.

Doctrine Bridge est optimisé pour la performance et la commodité du développement. Par exemple, lors de l'utilisation de Doctrine, API Platform est capable d'optimiser automatiquement les requêtes SQL générées en ajoutant les JOIN clauses appropriées. Il fournit également de nombreux filtres intégrés puissants.

Doctrine ORM et son pont prennent en charge les SGBD les plus populaires, notamment PostgreSQL, MySQL, MariaDB, SQL Server, Oracle, SQLite et MongoDB ODM.

API Platform a une recette officielle Symfony Flex. Cela signifie que vous pouvez facilement l'installer depuis n'importe quelle application Symfony en utilisant le binaire Symfony :

4.2.3.Symfony / API Platform : Installation & Configuration

Il faut d'abord s'assurer d'avoir l'environnement adéquat sur sa machine, entre autres WAMP Server et Composer pour ma part.

Pour créer un nouveau projet Symfony :

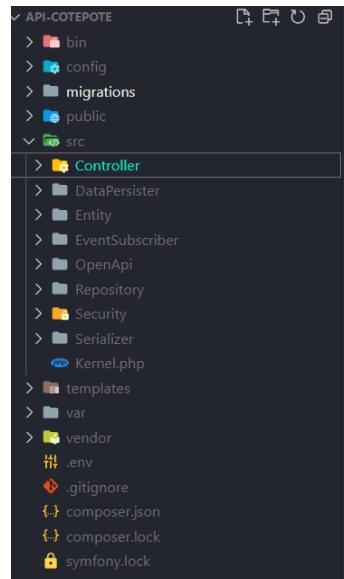
```
symfony new nom-du-projet
```

```
cd nom-du-projet
```

Pour Installer le composant serveur de la Api Platform dans ce squelette :

```
symfony composer require api
```

une fois ces deux commandes lancés, nous nous retrouvons avec l'architecture suivante :



Ce qui est pratique avec ce type de framework, on peut aisément utiliser et télécharger des librairies pour pouvoir s'en servir par la suite sur mon projet, je peut retrouver simplement dans le fichier "composer.json" toutes les librairies et dépendances que j'ai sur mon projet :

```
composer.json > ...
1 ~ {
2   "type": "project",
3   "license": "proprietary",
4   "minimum-stability": "stable",
5   "prefer-stable": true,
6   "require": {
7     "php": ">=8.0",
8     "ext-ctype": "*",
9     "ext-iconv": "*",
10    "api-platform/core": "^2.6",
11    "doctrine/annotations": "^1.0",
12    "doctrine/doctrine-bundle": "^2.5",
13    "doctrine/doctrine-migrations-bundle": "^3.2",
14    "doctrine/orm": "2.11",
15    "gesdinet/jwt-refresh-token-bundle": "*",
16    "lexik/jwt-authentication-bundle": "2.14",
17    "nelmio/cors-bundle": "2.2",
18    "phpdocumentor/reflection-docblock": "5.3",
19    "phpstan/phpdoc-parser": "1.2",
20    "ramsey/uuid": "4.2",
21    "symfony/asset": "5.4.*",
22    "symfony/console": "5.4.*",
23    "symfony/dotenv": "5.4.*",
24    "symfony/expression-language": "5.4.*",
25    "symfony/flex": "1.17|2",
26    "symfony/framework-bundle": "5.4.*",
27    "symfony/property-access": "5.4.*",
28    "symfony/property-info": "5.4.*",
29    "symfony/proxy-manager-bridge": "5.4.*",
30    "symfony/runtime": "5.4.*",
31    "symfony/security-bundle": "5.4.*",
32    "symfony/serializer": "5.4.*",
33    "symfony/twig-bundle": "5.4.*",
34    "symfony/validator": "5.4.*",
35    "symfony/yaml": "5.4.*",
36    "vich/uploader-bundle": "1.19"
37  },
38  "config": {
39    "allow-plugins": [
40      "composer/package-versions-deprecated": true,
41      "symfony/flex": true,
42      "symfony/runtime": true
43    ]
44  }
45}
```

je peux très facilement en télécharger avec composer qui est un logiciel gestionnaire de dépendances libre écrit en PHP. Il permet à ses utilisateurs de déclarer et d'installer les bibliothèques dont le projet principal a besoin.

pour la suite de la création de notre api, on se rend dans le fichier ".env" de notre projet pour modifier la ligne " DATABASE_URL", cela nous permet de dire où est ce que doctrine va créer notre base de données ainsi que l'emplacement où sera faites les migrations :

```
DATABASE_URL="mysql://root:@127.0.0.1:3306/apicotepte?serverVersion=5.7"
```

Une fois cela fait, je n'ai plus qu'à créer ma base de données dans le SGBD et son schéma dans mon projet avec les lignes de commandes suivantes :

```
symfony console doctrine:database:create
```

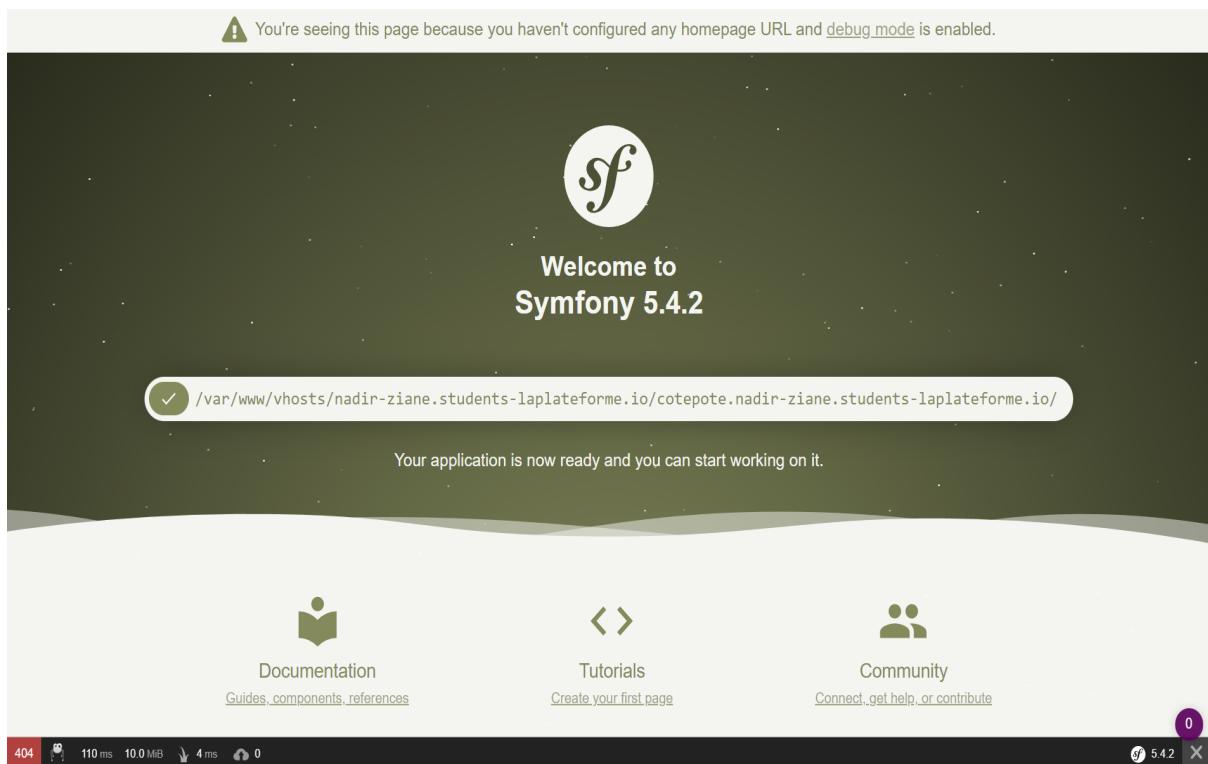
```
symfony console doctrine:schema:create
```

à la suite de ces lignes de commandes, je me retrouve avec ma base de données créé dans mon SGBD :

Pour vérifier si toutes les actions que j'ai effectués ont bien fonctionnés, je démarre le serveur PHP intégré avec la ligne de commande suivante :

```
symfony serv
```

Pour vérifier si cela à bien fonctionné, j'ouvre <https://localhost>
Si cela fonctionne bien, il y aura cette vue sur le navigateur :



Lors de l'installation de la Api Platform de cette manière, l'API sera exposée en tant que "/api/".

Donc j'ouvre <http://localhost:8000/api/> pour voir la documentation de l'API.

The screenshot shows the API Platform documentation interface. At the top, there is a header with the API Platform logo and a status bar showing "0.0.0 OAS3". Below the header, there is a search bar labeled "Servers" with a dropdown menu containing a single item: "/". To the right of the search bar is a green "Authorize" button with a padlock icon. The main content area is titled "Bet" and lists various HTTP methods and their corresponding URLs and descriptions:

Method	URL	Description
GET	/api/bets	Retrieves the collection of Bet resources.
POST	/api/bets	Creates a Bet resource.
GET	/api/bets/{id}	Retrieves a Bet resource.
DELETE	/api/bets/{id}	Removes the Bet resource.
PATCH	/api/bets/{id}	Updates the Bet resource.
POST	/api/bets/{id}/image	Creates a Bet resource.

API Platform expose une description de l'API au format OpenAPI (anciennement connu sous le nom de Swagger). Il intègre également une version personnalisée de Swagger UI , une belle interface restituant la documentation OpenAPI.

Il faut cliquer sur une opération pour afficher ses détails. et je peux également envoyer des requêtes à l'API directement depuis l'interface utilisateur.

Mon projet API Platform est désormais 100% fonctionnel, il ne reste plus qu'à implémenter mes différentes entités et y mettre les annotations nécessaires pour le bon fonctionnement de mon api.

4.2.4.Symfony / API Platform : Création d'une Collection

Une fois l'étape de la mise en place de mon projet en Symfony - Api Platform réalisé, la première étape que j'ai effectué est la création de mes entités, c'est là où Doctrine ORM rentre en jeu :

Doctrine facilite grandement la création de mes entités ainsi que mes champs, il prend tout en compte pour la création / modélisation de ma base de données.

A l'aide de quelques lignes de commande je crée mon modèle de données et gère la persistance (la sauvegarde) dans une table.

Voici les commandes utiles pour rendre la création d'entité plus fluide, création d'une entité, création de la table correspondante, opération nommée migration.

Tout d'abord :

```
php bin/console make:entity
```

Cette ligne de commande me sert à créer une entité, l'étape suivante est de choisir le nom de cette entité (ou nom de l'entité que je souhaite modifier).

```
Class name of the entity to create or update (e.g.  
AgreeableJellybean):  
> Product  
created: src/Entity/Product.php  
created: src/Repository/ProductRepository.php  
Entity generated! Now let's add some fields!  
You can always add more fields later manually or by re-running this  
command.
```

l'étape qui suit est la création de mes champs, cela se fait en quelques étapes :

```
New property name (press <return> to stop adding fields):  
> name  
Field type (enter ? to see all types) [string]:  
> string  
Field length [255]:  
>  
Can this field be null in the database (nullable) (yes/no) [no]:  
>  
updated: src/Entity/Product.php
```

tout d'abord le type (si c'est une string,, un booléen, un datetime)
dans mon cas c'est une string et par là suite je dois choisir la longueur de la chaîne de caractère et enfin si elle peut être nul.

je n'ai plus qu'à répéter l'opération pour chaque champ de mon entité.

une fois la création de mon entité effectué je n'ai plus qu'à faire une migration pour que ma base de données dans mon SGBD (Système de Gestion de Base de Données) se met à jour :

cela se fait en une ligne de commande :

```
php bin/console make:migration
```

Maintenant je fais la migration proprement dite, cette migration est possible que si dans le fichier .env, il y a la chaîne de connexion bien remplie avec les identifiant et mot de passe, nom de la base de donnée et du host.

Il suffit de lancer la ligne de commande suivante :

```
php bin/console doctrine:migrations:migrate
WARNING! You are about to execute a database migration that could
result in schema changes and data loss. Are you sure you wish to
continue? (y/n)y
Migrating up to 20190612091941 from 0

++ migrating 20190612091941

    -> CREATE TABLE product (id INT AUTO_INCREMENT NOT NULL, name
VARCHAR(255) NOT NULL, price INT NOT NULL, PRIMARY KEY(id)) DEFAULT
CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci ENGINE = InnoDB

++ migrated (took 128.9ms, used 12M memory)

-----
++ finished in 133.7ms
++ used 12M memory
++ 1 migrations executed
++ 1 sql queries
```

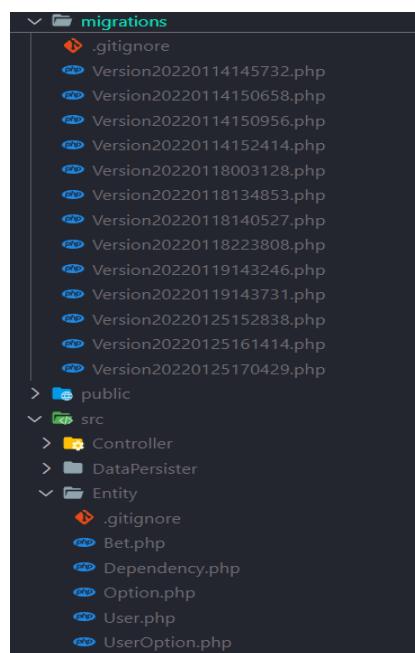
maintenant je retrouve ma table en base de données :

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra	Action
1	id	int(11)			Non	Aucun(e)		AUTO_INCREMENT	Modifier Supprimer Plus
2	email	varchar(180)	utf8mb4_unicode_ci		Non	Aucun(e)			Modifier Supprimer Plus
3	roles	longtext	utf8mb4_unicode_ci		Non	Aucun(e)			Modifier Supprimer Plus
4	password	longtext	utf8mb4_unicode_ci		Non	Aucun(e)			Modifier Supprimer Plus
5	name	varchar(255)	utf8mb4_unicode_ci		Non	Aucun(e)			Modifier Supprimer Plus
6	surname	varchar(255)	utf8mb4_unicode_ci		Non	Aucun(e)			Modifier Supprimer Plus
7	file_path	longtext	utf8mb4_unicode_ci		Oui	NULL			Modifier Supprimer Plus
8	updated_at	datetime			Oui	NULL			Modifier Supprimer Plus

et je vois également que dans ma table "doctrine_migration_versions" une ligne est ajouté à chaque migration :

	version	executed_at	execution_time
Éditer	Copier	Supprimer DoctrineMigrations\Version20220114145732	2022-01-14 14:57:47
Éditer	Copier	Supprimer DoctrineMigrations\Version20220114150658	2022-01-14 15:07:04
Éditer	Copier	Supprimer DoctrineMigrations\Version20220114150956	2022-01-14 15:10:03
Éditer	Copier	Supprimer DoctrineMigrations\Version20220114152414	2022-01-14 15:24:20

une fois toutes ces étapes réalisées, je vois que dans l'architecture de mes fichiers est mis également à jour, un fichier s'ajoute dans le dossier migrations et également un fichier sera ajouter dans le dossier pour les entités :



Également dans l'interface swagger de mon API je vois qu'une nouvelle collection de routes API concordante à l'entité que je viens de créer et de migrer avec ces méthodes (GET, POST, DELETE, PUT, PATCH), le CRUD de base entre autres, avec la possibilité de tester mes routes, voir quelles informations sont renvoyées après l'appel API que je fais :

The screenshot shows the API Platform Swagger interface. At the top, there's a navigation bar with a logo, the text "API PLATFORM", and a "Authorize" button. Below the header, a "Servers" dropdown is set to "/".

The main content area displays the "Bet" resource collection. It lists several operations:

- GET /api/bets**: Retrieves the collection of Bet resources.
- POST /api/bets**: Creates a Bet resource.
- GET /api/bets/{id}**: Retrieves a Bet resource.
- DELETE /api/bets/{id}**: Removes the Bet resource.
- PATCH /api/bets/{id}**: Updates the Bet resource.
- POST /api/bets/{id}/image**: Creates a Bet resource.
- POST /api/bets/{id}/setExpired**: Permet de set un pari en expiré.

Below the collection, a specific operation is expanded:

POST /api/bets Creates a Bet resource.

Description: Creates a Bet resource.

Parameters: No parameters

Request body (required): application/json

Description: The new Bet resource

Example Value | Schema

```
{
  "title": "string",
  "description": "string",
  "user": "string",
  "options": [
    {
      "description": "string"
    }
  ]
}
```

Responses

Code	Description	Links
201	Bet resource created	GetBetItem The <code>id</code> value returned in the response can be used as the <code>id</code> parameter in <code>GET /api/bets/{id}</code> . Operation `getBetItem` Parameters { "id": "\$response.body#/id" }
	Media type: application/json Controls Accept header.	
	Example Value Schema	
	{ "id": 0, "uuid": "string", "title": "string", "description": "string", "createdAt": "2022-06-21T01:19:56.788Z", "expired": true, "user": "string", "options": [{"description": "string"}] }	

Globalement, ce qui va nous intéresser c'est le fichier qui sera ajouter dans le dossier entité :

c'est tout simplement la classe qui va gérer toute notre table et depuis laquelle je vais pouvoir ajouter des annotations pour configurer mes routes API

Cela va comprendre majoritairement le type de sérialisation et désérialisation que je vais choisir :

La sérialisation est ce que mon api va me retourner, et la désérialisation est ce que j'envoie à mon API.

cela est gérée dans le contexte de normalisation et de dénormalisation :

En ce qui concerne le contexte de dénormalisation, cela va définir quelle données je souhaite intercepter de mon API, cela va se faire à travers les annotations dans ma classe :

via ce type d'écriture commenter, API Platform va tout simplement lire les annotations au sein de "@ApiResource(...)" pour voir les règles concernant les routes API, ce qui va principalement définir les règles de normalisation ainsi que de dénormalisation de mes routes sont les deux suivantes :

```
/*
 * @ORM\Entity(repositoryClass=BetRepository::class)
 * @ApiResource(
 *     normalizationContext = {"groups" = {"read:bet"}},
 *     denormalizationContext = {"groups" = {"create:bet"}},
 *     itemOperations = {"get", "put", "patch", "delete"}
 * )
```

À l'intérieur, je vais tout simplement dire quel groupe de lecture j'affecte à l'un et à l'autre pour savoir quels éléments sont concernés dans ma base de données.

Ces noms de groupes, je n'ai plus qu'à les rappeler au dessus des attributs dans ma classe pour que je décide lesquels seront concernées, la manière est la suivante :

```
class Bet
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     * @Groups({"read:bet"})
     */
    private $id;

    /**
     * @ORM\Column(type="string", Length=255)
     * @Groups({"read:bet"})
     */
    private $uuid;

    /**
     * @ORM\Column(type="string", length=255)
     * @Groups({"read:bet", "create:bet"})
     * @Assert\Length(
     *     min = 5,
     *     max = 255
     * )
     */
    private $title;
```

Je mets dans l'annotation “@Groups()” l'attribut concerné pour que je puisse le retrouver dans mon contexte de normalisation, ou celui de dénormalisation.

4.2.5.Symfony / API Platform : Ajout des relations

Pour ajouter les cardinalités entre nos Collections, Doctrine met à disposition lors de la création de ma table des possibilités de les paramétriser. Celui-ci nous permet d'établir les liens entre nos modèles en choisissant le type de cardinalité correspondant à mon MCD.

Dans une relation ManyToOne, c'est le côté Many qui doit définir la relation, ici c'est l'entité Bet qui est le côté Many et User qui est le côté one, c'est donc dans l'entité Bet que nous allons définir la relation. Nous allons le faire à travers des lignes de commandes :

```
Field type (enter ? to see all types) [string]:  
> ManyToOne  
  
What class should this entity be related to?:  
> Article  
  
Is the Comment.article property allowed to be null (nullable)? (yes/no) [yes]:  
> no  
  
Do you want to add a new property to Article so that you can access/update Comment objects from it - e.g. $article->getComments()? (yes/no) [yes]:  
> yes  
  
A new property will also be added to the Article class so that you can access the related Comment objects from it.  
  
New field name inside Article [comments]:  
>  
  
Do you want to activate orphanRemoval on your relationship?  
A Comment is "orphaned" when it is removed from its related Article.  
e.g. $article->removeComment($comment)  
  
NOTE: If a Comment may *change* from one Article to another, answer "no".  
  
Do you want to automatically delete orphaned App\Entity\Comment objects (orphanRemoval)? (yes/no) [no]:  
> yes  
  
updated: src/Entity/Comment.php  
updated: src/Entity/Article.php
```

comme dans l'exemple ci dessous, il nous demande d'abord quelle est l'entité à modifier, ensuite le nom du champs que l'on souhaite attribué, par la suite définir que c'est un type ManyToOne, ensuite lui définir un nom, si le champs peut être nul et si je souhaite ajouter une nouvelle propriété à article et quelle sera son nom ou accepter le nom par default et si je souhaite donner la possibilité à Symfony de supprimer en cascade les données de la table en question si celle relié est supprimer et je décide que oui.

4.3. Spécificités techniques : Frontend

4.3.1. React Native

De nos jours, plusieurs types d'applications mobiles existent. D'un côté, on retrouve les applications mobiles natives, conçues pour être exécutées sur un système d'exploitation spécifique, elles requièrent l'apprentissage d'un langage propre à la plateforme. Cela nous aurait amené à faire un gros travail en codant plusieurs fois l'application selon le nombre d'OS avec lesquels elle doit être compatible.

À l'inverse des applications natives, on a les applications cross platform. Leur développement requiert l'utilisation de frameworks tels que Ionic ou Cordova, ces derniers vont à partir d'une codebase pouvoir rendre une application compatible avec les différentes plateformes en ne la codant qu'une seule fois.

Cela représente un gain de productivité non négligeable, cependant ce modèle d'application présente quelques inconvénients, ce type de framework ne va pas donner accès aux composants natifs du téléphone, limitant ainsi les possibilités de développement.

De plus, les performances seront plus basses de ce qu'apporte une application native.

Heureusement, d'autres solutions existent pour remédier aux problèmes cités précédemment.

Ainsi, pour ne pas avoir à apprendre deux langages spécifiques mais garder l'avantage d'une application native, nous avons fait le choix de développer Côté Pote grâce à la librairie EXPO React Native.

EXPO React Native permet de développer des applications iOS, Android à partir de la même codebase Javascript / Typescript et ses concepts sont pratiquement identiques à ceux React, à la différence que EXPO React Native ne manipule pas le DOM.

Les applications mobiles construites avec EXPO React Native donnent des applications entièrement natives, qui utilisent les mêmes API que si elles étaient développées dans Xcode ou Android Studio.

4.3.2.Expo

Pour pouvoir mettre en place l'environnement de développement pour React Native où plus généralement pour la grande majorité des Framework Front-End, je sais que j'ai besoin de NodeJS qui est une plateforme logicielle libre en JavaScript, orientée vers les applications réseau événementielles.

Il y a également NPM ou Node Packet Manager, écrit en grande partie en JavaScript, est indissociable du succès de node.

Il permet de gérer ou bien de publier de nouveaux logiciels au sein de l'écosystème NodeJS.

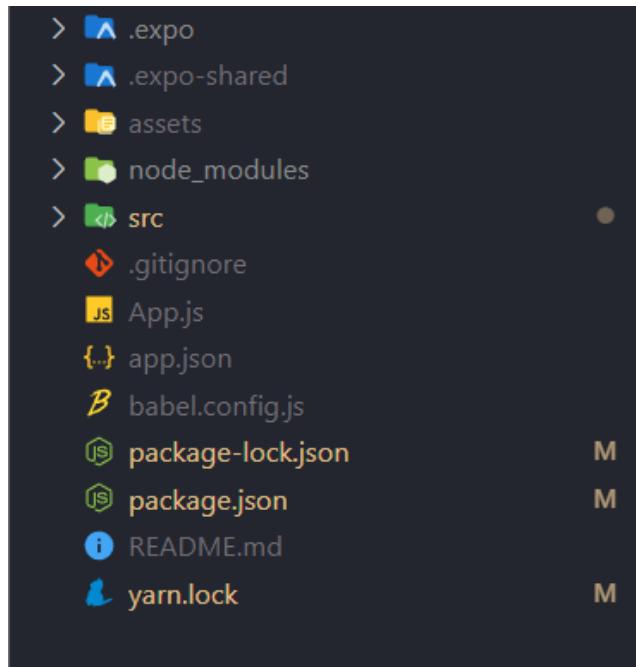
Pour installer React Native, nous avons utilisé la librairie Expo-CLI. Expo fournit un SDK permettant la création et le déploiement d'applications mobiles avec React Native en fournissant un ensemble d'outils et de services construits autour de ce dernier.

Par exemple, il nous permet à l'aide de l'utilitaire « watchman » de traquer nos fichiers et compiler notre code à la volée pour visualiser les résultats sur différents simulateurs iOS / Android ou directement sur notre appareil mobile personnel ce qui apporte un confort supplémentaire pendant le développement.

Pour créer un nouveau projet géré par Expo, il faut exécuter la commande Expo init qui va créer l'environnement nécessaire au développement de notre application.

```
expo init my-app
```

Une fois cette opération faite, je me retrouve avec une architecture de ce genre :



Ce qui est pratique une fois de plus avec ce type de framework, je peut aisément utiliser et télécharger des librairies pour pouvoir s'en servir par la suite sur mon projet, je peut retrouver simplement dans le fichier "composer.json" toutes les librairies et dépendances que j'ai sur mon projet :

```
package.json > private
1  {
2    "name": "app-cotepte",
3    "version": "1.0.0",
4    "main": "node_modules/expo/AppEntry.js",
5    "scripts": {
6      "start": "expo start",
7      "android": "expo start --android",
8      "ios": "expo start --ios",
9      "web": "expo start --web",
10     "eject": "expo eject"
11   },
12   "dependencies": {
13     "@expo-google-fonts/dev": "^0.2.2",
14     "@expo-google-fonts/poppins": "^0.2.2",
15     "react-native-async-storage/async-storage": "^1.15.17",
16     "react-native-community/masked-view": "0.1.11",
17     "react-native-bottom-tabs": "0.0.9",
18     "react-navigation/drawer": "0.6.3",
19     "react-navigation/native": "0.6.0",
20     "react-navigation/stack": "0.6.11",
21     "axios": "0.25.0",
22     "buffer": "6.0.1",
23     "expo": "~44.0.0",
24     "expo-app-loading": "~1.3.0",
25     "expo-constants": "~13.0.1",
26     "expo-font": "~10.0.4",
27     "expo-google-fonts": "0.0.0",
28     "expo-image-picker": "0.12.0.1",
29     "expo-secure-store": "0.11.1.0",
30     "expo-status-bar": "0.12.0",
31     "react": "17.0.1",
32     "react-dom": "17.0.1",
33     "react-native": "0.64.3",
34     "react-native-gesture-handler": "~2.1.0",
35     "react-native-keyboard-aware-scroll-view": "0.9.5",
36     "react-native-reanimated": "2.3.1",
37     "react-native-safe-area-context": "0.7.3.2",
38     "react-native-screens": "3.10.1",
39     "react-native-web": "0.17.1",
40     "save": "2.4.0"
41   },
42 }
```

4.3.3.React Navigation

Maintenant il faut savoir que la navigation entre différente vue est très différente de celle du web classique, il n'y a pas de "href" qui ramène d'un lien à l'autre.

Il y a React Navigation qui est une bibliothèque populaire pour le routage et la navigation dans une application React Native, Cette bibliothèque aide à résoudre le problème de la navigation entre plusieurs écrans et du partage des données entre eux.

Pour naviguer entre les écrans, j'utilise StackNavigator, il fonctionne exactement comme une pile d'appels.

Chaque écran vers lequel je navigue est poussé vers le haut de la pile. Chaque fois que j'appuie sur le bouton Retour, les écrans se détachent du haut de la pile.

tout d'abord je commence par installer la librairie avec cette commande sur mon terminal :

```
npm install @react-navigation/native
```

Ensuite, j'installe @react-navigation/stack et ses dépendances

Pour Côté Pote, il est possible de naviguer au travers de l'application grâce à un composant TabBar, situé au bas de l'écran, qui permet de basculer entre les écrans principaux de l'application.

Pour que ces composants soient fonctionnels, ils doivent être les enfants du composant <NavigationContainer>, fourni par React Navigation, et que l'on va appeler à la racine de notre application dans le composant App.js. <NavigationContainer> est la base qui va nous permettre de gérer notre arbre de navigation ainsi que ses différents états et fournir différentes fonctionnalités propres à la plateforme où il sera exécuté.

Après avoir créé mon <NavigationContainer> nous appelons la fonction `createBottomTabNavigator` qui va nous créer notre TabBar et retourner un objet contenant deux propriétés, `Navigator` et `Screen`. Chaque composant <Tab.Screen> correspond à une Tab au bas de l'écran et nous permet de changer d'écran.

Ces derniers doivent tous être contenu dans un composant parent <Tab.Navigator> et contenir au minimum les deux props suivantes pour permettre la transition entre les écrans :

- Une prop `component`, correspondant au functional component de l'écran qui doit être rendu
- Une prop `name`, qui va nous permettre de naviguer vers mon écran grâce au nom qui lui est attribué

```
const Tab = createBottomTabNavigator();

const Tabs = () => {
  const [pic, setPic] = useState("");
  const profilePic = async () => {
    let mag = await UserInfoService.getAllUserInfo();
    if (mag) {
      setPic(JSON.parse(mag).fileurl);
    } else {
      return;
    }
  };
  profilePic();
}

return (
  <Tab.Navigator
    screenOptions={{
      headerShown: false,
      tabBarStyle: {
        backgroundColor: "black",
        height: 100,
        borderTopColor: "black",
        ...styles.tab,
      },
      tabBarShowLabel: false,
    }}
  >
  <Tab.Screen
    name="Home"
    component={AccueilScreen}
    options={{
      tabBarIcon: ({ focused }) => (
        <View
          style={{ alignItems: "center", justifyContent: "center", top: 5 }}
        >
          <Image
            source={require("../assets/icons/home-icon.png")}
            resizeMode="contain"
            style={{
              width: 40,
              height: 40,
              tintColor: focused ? "white" : "grey",
            }}
          />
        </View>
      ),
    }}
  />
)
```

Après avoir créé nos Tabs, nous avons créé les écrans qui seront affichés au sein de ces derniers.

La création d'un écran se fait de la même manière que la création d'une Tab à l'exception de la fonction que l'on va appeler qui est `createNativeStackNavigator`.

Grâce à cette fonction, nous pouvons créer une Stack contenant plusieurs écrans et établir une navigation entre eux grâce à la gestion évènementielle des composants fournis par React Native.

```
import React, { useEffect } from "react";
import "react-native-gesture-handler";
import { NavigationContainer } from "@react-navigation/native";
import { createStackNavigator } from "@react-navigation/stack";
import NewBetScreen from "../screens/NewBetScreen";
import CreateBetScreen from "../screens/CreateBetScreen";
import SearchBetScreen from "../screens/SearchBetScreen";

const { Navigator, Screen } = createStackNavigator();

const BetNavigator = () => {
  return (
    // <NavigationContainer>
    <Navigator screenOptions={{ headerShown: false }}>
      <Screen name="CreateBet" component={CreateBetScreen} />
      <Screen name="SearchBetById" component={SearchBetScreen} />
      <Screen name="NewBet" component={NewBetScreen} />
    </Navigator>
    // </NavigationContainer>
  );
};

export { BetNavigator };
```

4.3.4. Axios : Interaction avec la base de données

Pour interagir avec mes données j'ai utilisé la librairie Axios.

Cette librairie est un client HTTP nous permettant de faire des requêtes en BDD et de recevoir nos réponses sous forme de Promesse.

Il fournit plusieurs méthodes facilitant l'écriture des requêtes et contrairement à Fetch, il permet un meilleur support des réponses renvoyées par le serveur car elles n'ont pas besoin d'être converties au format JSON et peuvent être exploitées lorsque la Promesse est résolue.

Pour effectuer une requête avec Axios, nous devons importer le module axios au sein de notre fichier en charge de la requête.

Ce module comporte plusieurs fonctions dans lesquelles nous devons renseigner en paramètre les informations liées à la requête.

Chaque fonction est un raccourci lié à la méthode d'un verbe http et pour chaque requête il faut passer en paramètre l'URL de notre ressource et un objet contenant notre body dans le cas d'une écriture en BDD.

```
await axios
  .get(
    "https://cotepte.nadir-ziane.students-laplateforme.io/api/users/" + id,
  {
    headers: {
      "Content-type": "application/json",
    },
  }
)
.then(async (response) => {
  if (response.data.id) {
    console.log(response.data.fileurl);
    let fileurl = response.data.fileurl;
    setPic(
      "https://cotepte.nadir-ziane.students-laplateforme.io" + fileurl
    );
    console.log(pic);
  } else {
    console.log("erreur, pas de retour de l'api");
  }
})
.catch(async (e) => {
  console.log(e);
});
```

5. Présentation du jeu d'essai

5.1. Parcours utilisateur pour la connexion d'un user

Je vais vous présenter les extraits de code retracant les étapes du processus de la connexion d'un utilisateur.

Le point d'entrée de l'application se faisant au sein du fichier App.js, c'est à partir de ce même composant que le rendu de la Tab Bar ou de la stack de navigation est fait.

j'ai une tab bar dans le cas où un utilisateur est connecté ou une stack de navigation avec 3 vues si l'utilisateur n'est pas authentifié.

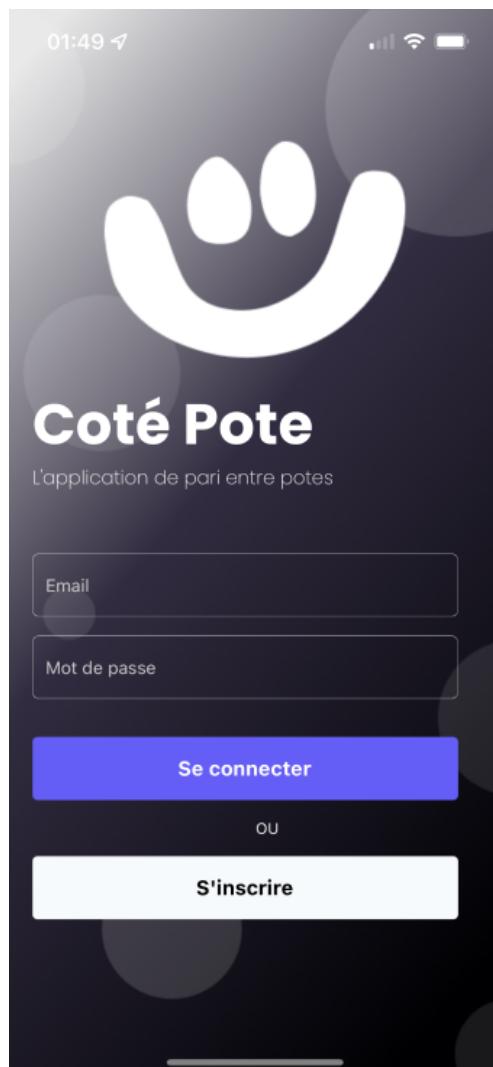
L'onglet Connexion est la première de la pile et donne accès à la première page du parcours utilisateur pour la connexion.

```
return (
  <AuthContext.Provider value={authContext}>
    {state.userToken !== null ? (
      <>
        <NavigationContainer>
          <Navigator screenOptions={{ headerShown: false }}>
            <Screen name="Tab" component={Tabs} />
          </Navigator>
        </NavigationContainer>
      </>
    ) : (
      <>
        <NavigationContainer>
          <Navigator screenOptions={{ headerShown: false }}>
            <Screen name="Connexion" component={ConnexionScreen} />
            <Screen name="Inscription" component={InscriptionScreen} />
            <Screen name="Loading" component={LoadingScreen} />
          </Navigator>
        </NavigationContainer>
      </>
    )}
  </AuthContext.Provider>
);
```

Cette page, ainsi que les suivantes sont les enfants d'une Stack d'écran créée grâce à la fonction `createStackNavigator` au sein de l'application. Lorsque cette fonction est appelée, elle met à disposition deux propriétés, `Navigator` et `Screen`, j'importe également `NavigationContainer` qui va englober `Navigator` et les `Screens`.

`<Screen>` dispose de plusieurs props dont deux importantes, "name" qui permet de naviguer vers le composant et ensuite "component" qui va effectuer le rendu de ce que je veux afficher.

Le screen qui sera affiché par défaut sera "Connexion" Le rôle de cet écran est d'afficher les inputs mail et mot de passe, c'est input vont me permettre de faire un call API avec leurs valeurs en paramètre pour interroger mon API.



```

const authContext = React.useMemo(
() => ({
  signIn: async (data) => {
    let token;
    await axios
      .post(
        "https://coteplateforme.nadir-ziane.students-laplateforme.io/api/login",
        {
          username: data.login,
          password: data.password,
        },
        {
          headers: {
            "Content-type": "application/json",
          },
        }
      )
      .then(async (response) => {
        if (response.data?.token) {
          await SecureStore.setItemAsync("token", `${response.data.token}`);
          token = response.data.token;
          await SecureStore.setItemAsync("data", `${response.data}`);
          UserService.getUser();
          await SecureStore.setItemAsync("isOk", "1");
        } else {
        }
      })
      .catch(async (e) => {
        console.log(e);
        await SecureStore.setItemAsync("error", "pas connecté");
        Alert.alert("Erreur", "Identifiant ou mot de passe incorrect", [
          {
            text: "OK",
          },
        ]);
      });
    dispatch({ type: "SIGN_IN", token: token });
  },
}),
);

```

J'effectue cette requête dans un hook useMemo qui me sert à renvoyer une valeur mémorisée, que je passe ensuite en paramètre à la méthode React.createContext() avec lequel je vais englober tous mes composants dans celui-ci pour pouvoir y avoir accès de manière global sans me soucier de comment faire passer de l'enfant au parent ou inversement des données, des méthodes ou des variables de manière global dans mon application.

```

import * as React from 'react';

const AuthContext = React.createContext();

export default AuthContext;

```

```

return (
  <AuthContext.Provider value={authContext}>
    {state.userToken !== null ? (
      <>
        <NavigationContainer>
          <Navigator screenOptions={{ headerShown: false }}>
            <Screen name="Tab" component={Tabs} />
          </Navigator>
        </NavigationContainer>
      </>
    ) : (
      <>
        <NavigationContainer>
          <Navigator screenOptions={{ headerShown: false }}>
            <Screen name="Connexion" component={ConnexionScreen} />
            <Screen name="Inscription" component={InscriptionScreen} />
            <Screen name="Loading" component={LoadingScreen} />
          </Navigator>
        </NavigationContainer>
      </>
    )}
  </AuthContext.Provider>
);

```

Je stocke le résultat de la réponse dans le storage interne du mobile de l'utilisateur grâce à aux méthodes mises à disposition dans la librairie "SecureStore".

Les hooks sont des fonctions faisant partie de l'écosystème React et apportent au sein des composants de type fonction, des fonctionnalités auparavant accessibles seulement aux composants de type classe.

- Le hook useEffect permet de gérer les cycles de vie d'un composant et déclencher des actions lorsque le rendu de celui-ci est terminé.
- Le hook useState il permet de gérer des états au sein du composant et mettre à jour ce dernier si un changement survient sur le state, ayant pour effet de lancer un nouveau rendu pour mettre à jour l'écran.

j'utilise également un reducer pour gérer le state de mon application :

```
export default function App() {
  const [state, dispatch] = React.useReducer(
    (prevState, action) => {
      switch (action.type) {
        case "RESTORE_TOKEN":
          return {
            ...prevState,
            userToken: action.token,
            isLoading: false,
          };
        case "SIGN_IN":
          return {
            ...prevState,
            isSignout: false,
            userToken: action.token,
          };
        case "SIGN_OUT":
          return {
            ...prevState,
            isSignout: true,
            userToken: null,
          };
      }
    },
    {
      isLoading: true,
      isSignout: false,
      userToken: null,
    }
);
```

Comme on peut le voir plus haut dans l'illustration de mon call api, à la toute fin de l'exécution de cette partie du script, j'utilise la méthode dispatch à laquelle je vais passer en paramètre le type sur laquelle je vais changer le state de mon application, en l'occurrence "SIGN_IN" et ensuite la props à lui transmettre qui sera l'action là en l'occurrence mon token et par la suite le userToken prendra la valeur de l'action.
dans mon cas si le userToken est différent de null, je donne la possibilité au user de se connecter et donc de changer de stack de navigation pour passer à la tabBar où sont stockées toutes les vues accessible par un user connecté grâce à une ternaire :

La ternaire conditionnel est le seul opérateur JavaScript qui comporte trois opérandes. Cet opérateur est fréquemment utilisé comme raccourci pour la déclaration de conditions, cela se lit de cette manière :

```
condition ? exprSiVrai : exprSiFaux
```

Pour éviter toutes difficultés dans l'ordre d'exécution des requêtes, dû à l'asynchronisme du langage Javascript, j'ai utilisé les mots-clefs `async/await` dans les fonctions qui appellent mes requêtes ainsi que dans les requêtes elles-mêmes. Ceci étant, cela améliore également la lisibilité du code et facilite le traitement des réponses renvoyées par l'API.

Concernant la navigation d'une vue à l'autre je vais prendre comme exemple la navigation entre la vue de connexion et celle d'inscription :

pour naviguer entre 2 vues, je dois d'abord passer dans la props de mon composant “`{navigation}`” :

```
export default function connexion({ navigation }) {
  const { signIn } = useContext(AuthContext);

  let [fontsLoaded] = useFonts({ Poppins_200ExtraLight });
  const [login, setLogin] = useState("");
  const [password, setPassword] = useState("");

  const goTo = () => {
    navigation.navigate("Inscription");
};
```

et ensuite je stock dans une constante la méthode “`navigation.navigate("nom_du_screen")`” pour signaler que lorsque j'appellerai cette constante, je veux que la vue qui a pour nom “`Inscription`” soit afficher.

```
<Pressable
  onPress={() => {
    goTo();
  }}
  style={styles.button1}
>
```

Je l'utilise dans une balise `Pressable` qui est un button et donc lorsque que j'appuierai dessus je serai sur la vue inscription.

pour faire une sorte de bouton de retour, il y a une méthode prévue dans navigation, la librairie couvre beaucoup de type de cas :

```
const goTo = () => {
  navigation.goBack();
};
```

Concernant la mise en place de ma user interface, il ne s'agit pas de HTML à proprement dit mais de JSX, cela ressemble à du HTML :

```
<KeyboardAwareScrollView showsVerticalScrollIndicator={false}>
  <View style={styles.container}>
    <Text style={styles.title}>Côté Pote</Text>
    <Text style={styles.txt}>L'application de pari entre potes</Text>
    <TextInput
      style={styles.ipt}
      onChangeText={setLogin}
      value={login}
      placeholder="Email"
      placeholderTextColor="#CFCFCF"
    />
    <TextInput
      style={styles.ipt2}
      onChangeText={setPassword}
      secureTextEntry={true}
      value={password}
      placeholder="Mot de passe"
      placeholderTextColor="#CFCFCF"
    />
    <Pressable
      onPress={() => {
        connexion();
      }}
      style={styles.button}
    >
```

JSX (JavaScript Extension), est une extension de React qui permet d'écrire du code JavaScript qui ressemble à du HTML. En d'autres termes, JSX est une syntaxe de type HTML utilisée par React.

Il faut savoir que les balises ne sont pas les mêmes, mais la documentation de react native est très bien faite à ce niveau et l'on peut même chercher sur google l'équivalent de tel ou tel balise html en React Native.

Pour le style, ce n'est toujours pas du CSS :

Avec React Native, je stylise mon application en utilisant JavaScript. Tous les composants de base acceptent un accessoire nommé style. Les noms et les valeurs de style correspondent généralement au fonctionnement de CSS sur le Web, sauf que les noms sont écrits en utilisant la camelCase, par exemple backgroundColor plutôt que background-color...

Il faut commencer par importer "StyleSheet" depuis la librairie react-native et ensuite déclaré une constante où l'on va stocker la méthode "StyleSheet.Create({})" et à l'intérieur déclarer toute les classes dont j'ai besoins :

```
const styles = StyleSheet.create({
  container: {
    marginLeft: 20,
    marginTop: 250,
  },
  title: {
    fontFamily: "Poppins_700Bold",
    fontSize: 45,
    fontWeight: "bold",
    color: "white",
  },
  txt: {
    fontFamily: "Poppins_200ExtraLight",
    fontSize: 15,
    fontWeight: "bold",
    color: "white",
  },
  ipt: {
    height: 50,
    width: 335,
    borderWidth: 0.25,
    borderColor: "white",
    borderRadius: 5,
    padding: 10,
    marginTop: 50,
    color: "white",
  },
})
```

et ensuite pour attribuer du style à un élément dans mon code je le fais de la manière suivante :

```
<Text style={styles.txt}>s'inscrire</Text>
```

6.Extrait d'une recherche à partir de site anglophone

Durant le développement de ce projet, j'ai rencontré de nombreuses situations qui ont nécessité une recherche d'informations.

En grande majorité, les solutions recherchées trouvaient réponses sur des sites anglophones (surtout StackOverFlow).

Pour décrire une de ces situations, je me suis tourné vers StackOverFlow pour savoir comment déboguer un problème que j'avais avec la librairie ImagePicker que j'ai utilisé dans ce projet.

The screenshot shows a Stack Overflow question page. The question is titled "how to select image from expo-image-picker react native". It was asked 3 months ago and modified 3 months ago, with 328 views. The question text is:

I try to select image from gallery and show it on Image component. I read the expo documentation and follow the step . when I try it on web it's work. but when I try it on android phone , after I select image and crop my app redownloading . I don't know where make mistake

One answer is shown, starting with:

```
import React, { useState, useEffect } from 'react';
import * as ImagePicker from 'expo-image-picker';
import { Button, Image, View, Platform } from 'react-native';

export default function GalleryComponenet() {
  const [image, setImage] = useState(null);

  useEffect(() => {
    (async () => {
      if (Platform.OS !== 'web') {
        const { status } = await ImagePicker.requestMediaLibraryPermissionsAsync();
        if (status !== 'granted') {
          alert('Sorry, Camera roll permissions are required to make this work')
        }
      }
    })();
  }, []);
  const chooseImage = async () => {
```

The sidebar on the right includes sections for "The Overflow Blog" (with links to Stripe and Waymo documentation), "Featured on Meta" (with links to developer survey results and traffic management tool testing), and "Hot Meta Posts" (with a link to a magic link for editing help).