

RICHER MARINE
ITH KAGNANA

FÉVRIER 2023

CAHIER DES CHARGES

APPLICATION
PLENTY



REACT NATIVE
& POSTGRESQL

Sommaire

1. Contexte
2. Expression du ou des besoins
3. Les objectifs
4. Les enjeux
5. Analyse fonctionnelle

Contexte

Plenty

Suite à la hausse des prix, le prix des courses à doubler. On parle même d'une autre augmentation des prix début mars. Même si la hausse de prix est minime, faire les courses est un besoin essentiel qui peut vite se retrouver très coûteux. Nous pensons à tous les étudiants ou à toutes les personnes qui ne peuvent pas se permettre de voir le prix de leurs courses augmenter.

Expression du ou des besoins

- Pouvoir réduire le prix de ses courses
- Retrouver ses bons plans
- Avoir sa propre liste de course

Les objectifs

Notre objectif est de regrouper tous les bons plans alimentaires sur la même appli pour permettre à nos utilisateurs de réduire le prix de leur courses.

Nous voulons faire en sorte que notre application s'ancre dans la vie de tous les jours et qu'elle facilite la vie de nos utilisateurs.

Nous avons aussi un objectif personnel qui est de progresser dans le domaine du développement avec des outils avec lesquels nous ne sommes pas familiers mais qui nous permettent d'enrichir nos connaissances et notre raisonnement.

Les enjeux

Après avoir appris des langages orientés sur le développement Web, tel que HTML, CSS et JavaScript. nous souhaitons découvrir à travers ce projet, l'utilisation de ces langages pour les applications mobiles avec React Native. De plus, pour la base de données, on découvre un nouveau langage, qui n'est autre que SQL.

De plus, nous allons apprendre le travail en groupe sur un même projet et à se répartir les tâches ainsi que l'utilisation d'outils comme GitHub.

Nous aimerais également souligner que la mise en ligne de notre base de données ainsi que de notre api en ligne nous a donné l'envie de pousser notre projet encore plus loin. Nous avons également beaucoup appris avec l'utilisation de Docker. Même si c'est notre premier projet, nous nous sentons très enrichies par ce projet.

Analyse fonctionnelle

Notre application présente plusieurs fonctionnalités :

- Ajouter des produits alimentaires sur le critère du prix, donc le prix le moins cher.
 - Nom du produit
 - Photo du produit
 - Magasin
 - Prix
 - Catégorie
- Pouvoir trier les produits par catégorie
- Pouvoir rechercher un produit par nom
- Pouvoir signaler si le produit a changé de prix ou s'il n'existe plus
- Pouvoir mettre en favoris un produit
- Créer un compte pour :
 - Retrouver ses produits en favoris
 - Faire sa liste de courses
 - Donner son avis

Langage et outils utilisés :

- React Native (Javascript)
- Postgresql (en base de donnée)
- Postman
- Docker
- GIT & Github
- Visual Studio Code
- Android Studio
- Node.js + Express.js

Nos répertoires github :

<https://github.com/MarineRcher/plenty>

<https://github.com/MarineRcher/Plenty-API>

Nos choix d'outils pour la réalisation de ce projet

- Framework

Le choix s'est porté sur React Native, framework d'application mobile créé par Facebook. Ce choix n'est pas anodin car le premier projet que nous avons réalisé ensemble était un site web avec React. Malgré notre différence de connaissance sur le framework React nous avions toutes les deux un objectif différent pendant ce projet.

L'utilisation de React Native permettra alors de donner de nouveaux enjeux.

React Native permet de créer des applications mobiles pour Android et iOS. On peut tester et accéder à notre application avec l'intermédiaire de l'application Expo Go.

- Base de donnée :

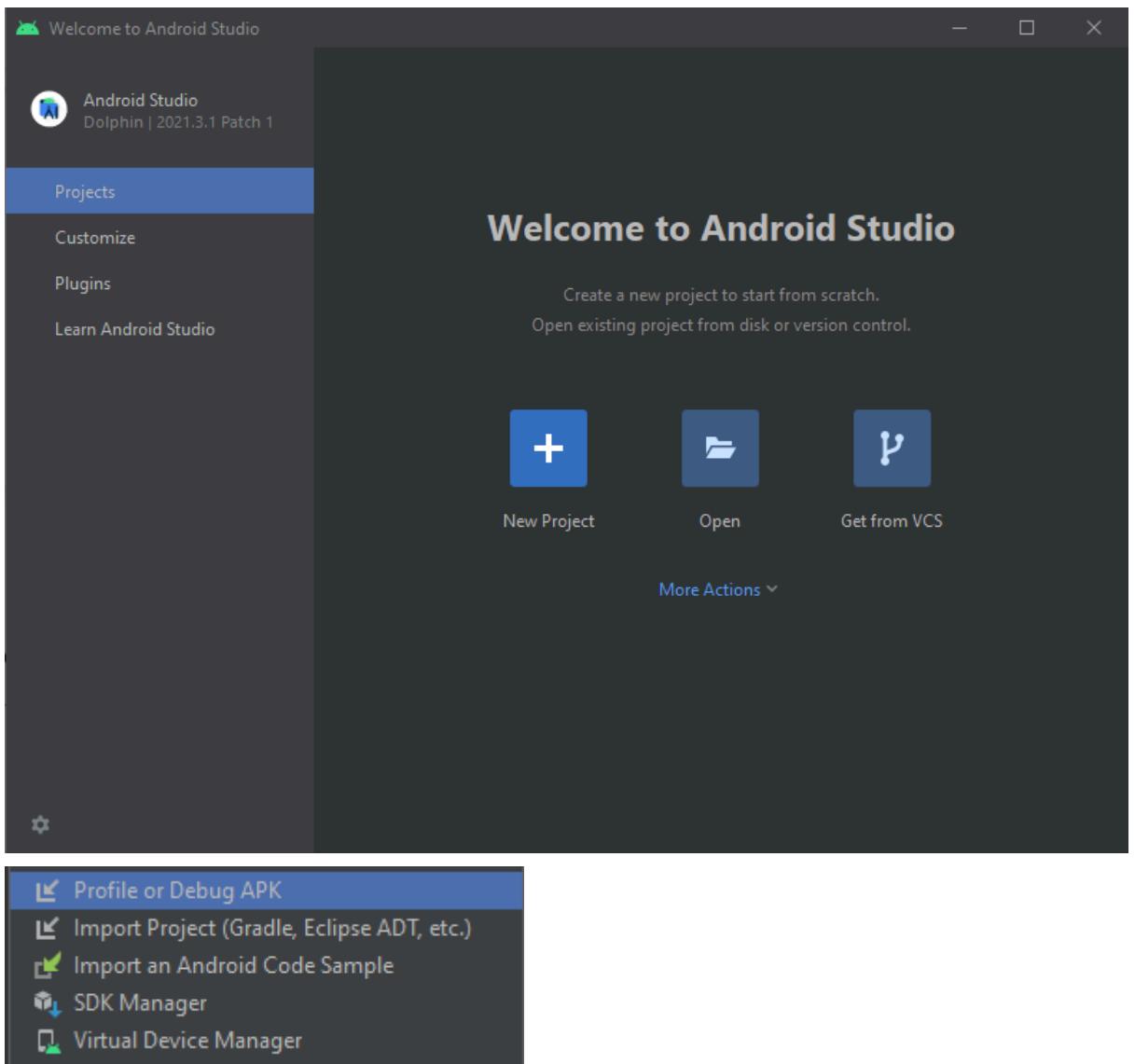
Le choix initial était SQLite en tant que base de données pour notre application. En effet, SQLite n'a pas besoin de serveur ce qui lui permet de s'exécuter dans l'application puisqu'une base de données SQLite est un fichier directement installé dans votre appareil.

Après réflexion, utiliser PostgreSQL pour gérer notre base de données semble une solution plus adéquate. PostgreSQL a beaucoup plus de fonctionnalités que SQLite et permet d'avoir une API REST.

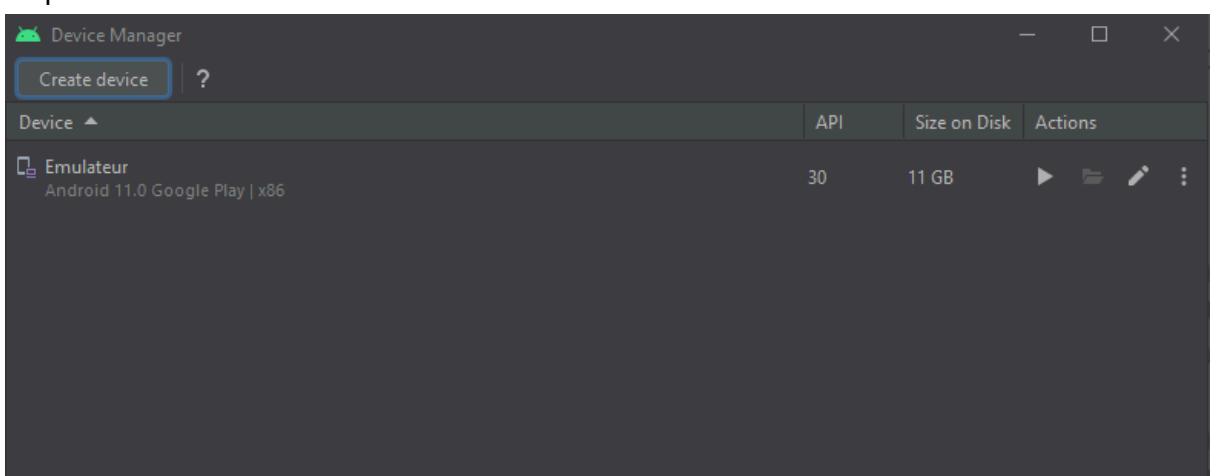
Pour tester notre application sur votre mobile ou un émulateur en local

Pour tester notre application sur un émulateur Android :

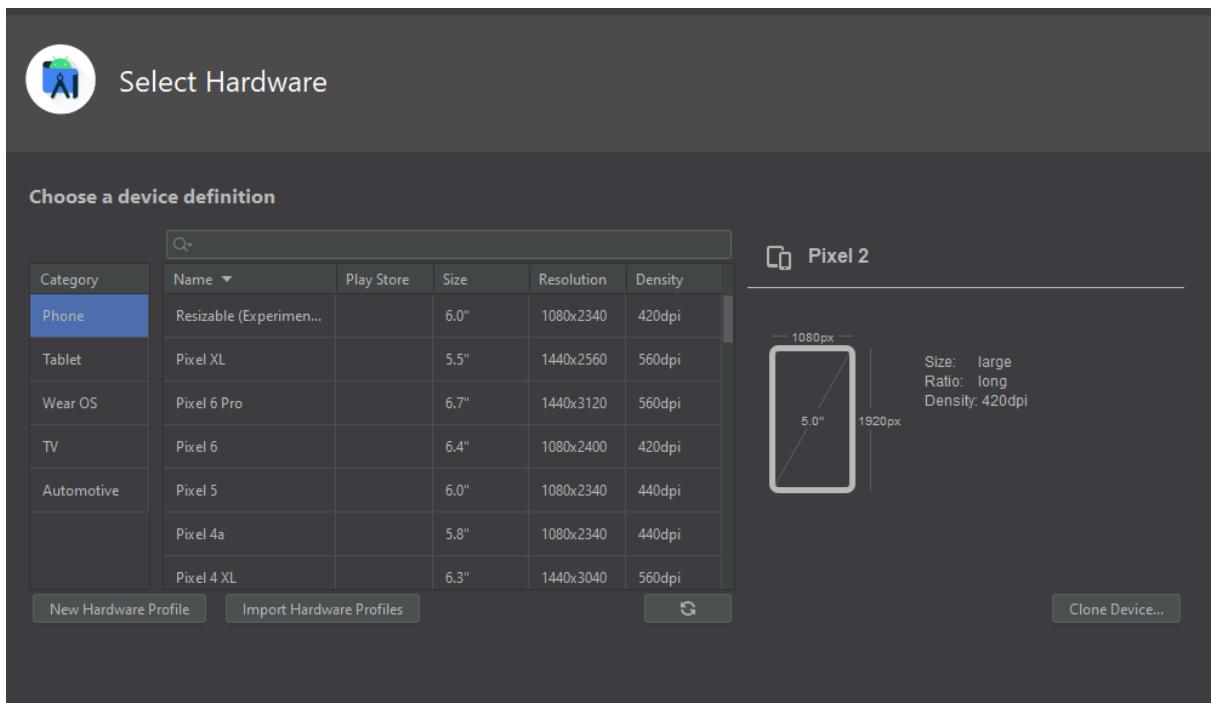
1. Télécharger Android Studio
2. Sélectionner "More Actions" puis cliquez sur "Virtual Device Manager"



3. Créez un nouveau téléphone
- Cliquez sur "Create device"



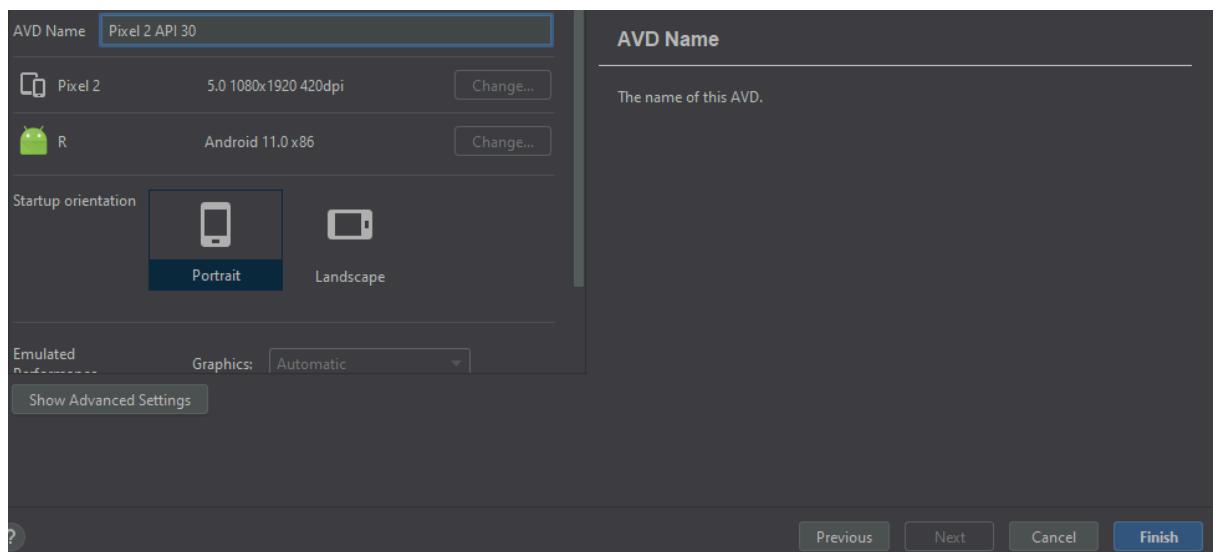
Puis choisissez la taille de votre appareil.



Puis votre version d'Android.

Release Name	API Level	ABI	Target
TiramisuPrivacySan... ↴	TiramisuPrivacySan	x86_64	Android API TiramisuPrivacy
Tiramisu ↴	33	x86_64	Android Tiramisu (Google Play)
Sv2 ↴	32	x86_64	Android 12L (Google Play)
S ↴	31	x86_64	Android 12.0 (Google Play)
R	30	x86	Android 11.0 (Google Play)
Q ↴	29	x86	Android 10.0 (Google Play)
Pie ↴	28	x86	Android 9.0 (Google Play)
Oreo ↴	27	x86	Android 8.1 (Google Play)
Oreo ↴	26	x86	Android 8.0 (Google Play)

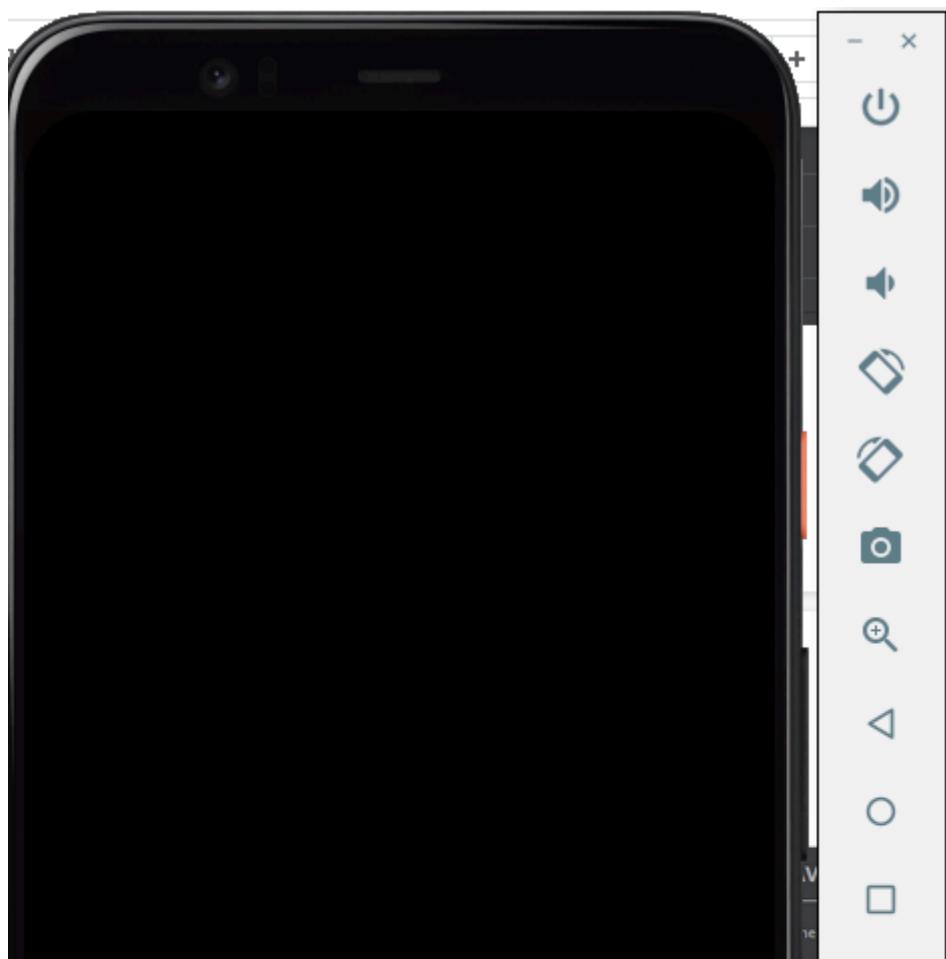
Donnez lui un nom et cliquer sur "Finish"



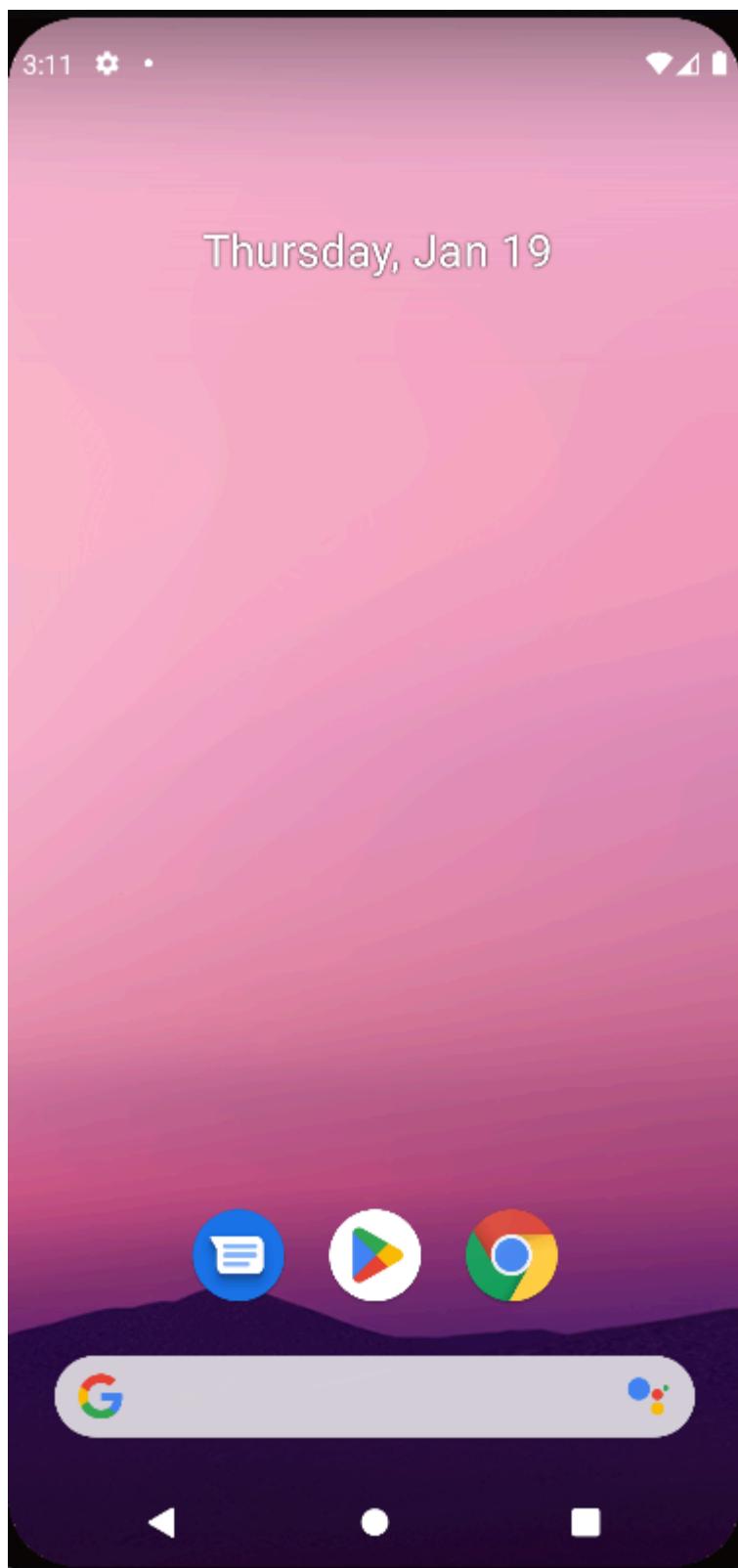
Ensuite pour lancer votre émulateur cliquer sur la flèche.



Allumez ensuite votre émulateur.



Et voilà !



4. Maintenant que vous avez un émulateur, allez dans le terminal et lancer le programme avec “npm start”
5. Dès que votre programme est lancé, choisissez l’option “a” pour lancer votre programme sur l’émulateur android. Et maintenant vous pouvez tester notre application :).

Tester notre application sur votre appareil Android ou Apple :

1. Télécharger sur votre appareil Expo Go.
2. Lancez notre programme avec “npm start”
3.
 - Si vous êtes sur **Android**, lancez l’application Expo Go et scanner le Qr Code qui s’affiche dans le terminal.
 - Si vous êtes sur **iPhone**, allez dans votre caméra et scannez le Qr Code. Ainsi, vous serez redirigé sur notre application.

Si le projet ne fonctionne pas, essayer de lancer le projet depuis la branche home

Pour changer de branche git :

- git checkout home

Puis ensuite faire dans le terminal :

- npm start

Initialisation du projet

● Backend

- Organisation :
 - Marine : les utilisateurs
 - Kagnana : les produits
- Installation de postgresql dans le serveur

on a pu avoir accès à un serveur distant hébergé par ovh. Voici les étapes qui ont permis d’installer postgresql.

1. Création du fichier .ssh : fichier config et clé privé
2. Lancer la console git dans le dossier tout en vérifiant bien que le path est le bon dans le fichier ? puis écrire `ssh ovhVPS`
3. Se connecter au serveur via une clé.
4. Comme c'est un serveur qui appartient à une connaissance, on avons créer un nouveau dossier à l'aide de la commande : `sudo plenty`
5. on avons ensuite accéder à ce dossier via la commande : `cd plenty`
6. Ensuite on avons ouvert un éditeur de texte qui va nous permettre de paramétrier le serveur postgresql avec la commande : `nano docker-compose.yml`

```

ubuntu@vps-6b962e28: ~/kan
GNU nano 6.2                               docker-compose.yml

version: '3.8'

services:

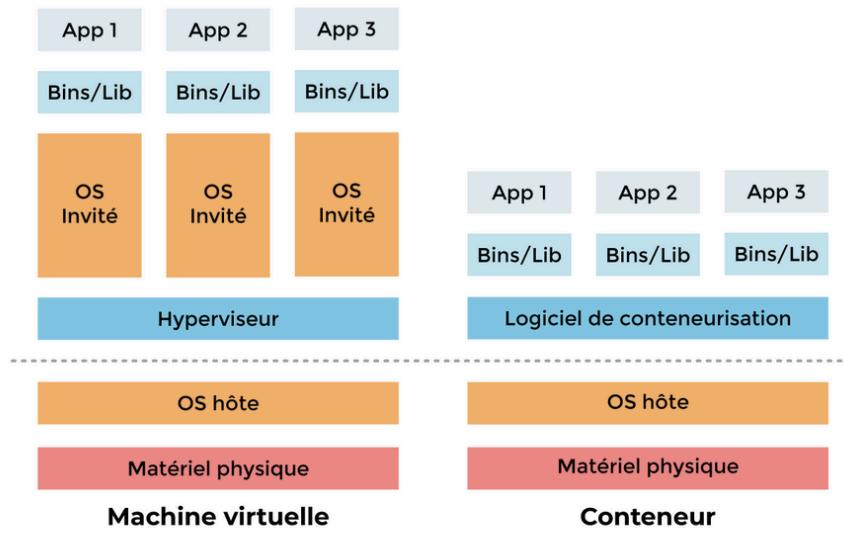
db:
  image: postgres:14.6-alpine
  restart: always
  ports:
    - 5432:5432
  environment:
    POSTGRES_PASSWORD: example
    POSTGRES_USER: kan
    POSTGRES_DB: plenty
  mem_limit: "250M"
  volumes:
    - ./data/sql:/var/lib/postgresql/data

```

- restart : permet de relancer le serveur si jamais il crash
- environment : on va donner un mot de passe, un utilisateur et un nom à la base de donnée
- mem-limit : permet de limiter la ram utilisée
- volumes : on lui donne un volume pour ne pas perdre nos données

Pourquoi avoir utilisé docker ?

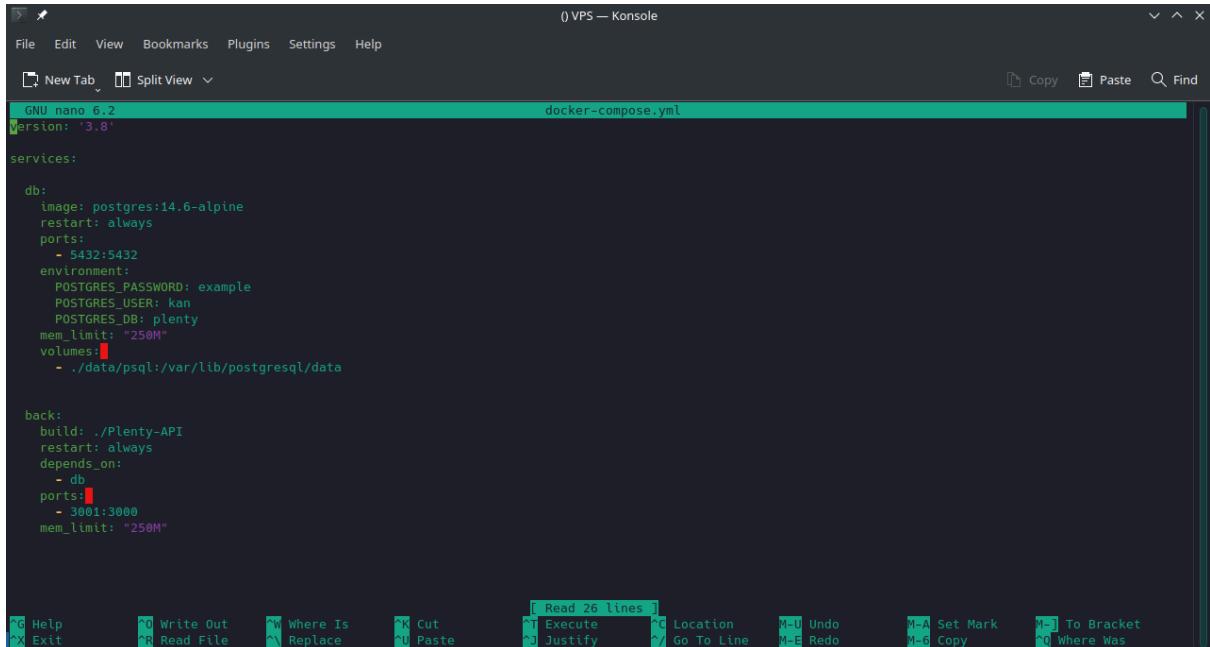
Docker permet de faire des containers qui on permettent d'avoir plusieurs instances qui tournent en même temps sans rien installer sur le serveur et ça permet de donner plus de contrôle sur les versions, la sécurité...etc. Cela permet d'exécuter une application sur n'importe quel serveur puisqu'on y englobe l'application ainsi que ses dépendances.



Comparaison entre les conteneurs et les machines virtuelles

Pour mieux comprendre Docker, et plus largement le concept de conteneur, on fait la comparaison avec une machine virtuelle. Contrairement à la machine virtuelle, le conteneur partage les ressources avec son hôte. C'est ce qui rend son démarrage plus rapide et un meilleur contrôle.

- Mettre le backend en ligne



```

GNU nano 6.2                               0 VPS — Konsole
File Edit View Bookmarks Plugins Settings Help
New Tab Split View
Copy Paste Find
Version: '3.8'

services:
  db:
    image: postgres:14.6-alpine
    restart: always
    ports:
      - 5432:5432
    environment:
      POSTGRES_PASSWORD: example
      POSTGRES_USER: kan
      POSTGRES_DB: plenty
    mem_limit: "250M"
    volumes:
      - ./data/postgresql:/var/lib/postgresql/data

  back:
    build: ./Plenty-API
    restart: always
    depends_on:
      - db
    ports:
      - 3001:3000
    mem_limit: "250M"

  
```

The screenshot shows a terminal window titled "VPS — Konsole" with a nano text editor open. The file is named "docker-compose.yml". The content of the file defines two services: "db" and "back". The "db" service uses the "postgres:14.6-alpine" image, restarts always, and maps port 5432 to 5432. It has environment variables for POSTGRES_PASSWORD, POSTGRES_USER, and POSTGRES_DB, and a memory limit of 250M. It also mounts a volume from the host directory "./data/postgresql" to the PostgreSQL data directory. The "back" service builds from the current directory ("./Plenty-API"), restarts always, depends on the "db" service, maps port 3001 to 3000, and has a memory limit of 250M.

Pour mettre l'API en ligne c'est le même principe que la base de données. On lui donne un build qui correspond au dossier contenant l'API. On lui donne le paramètre “depends_on” pour lui dire qu'il ne se lance pas tant que la base de donnée n'est pas lancée.

Ensuite dans l'api on ajoute un fichier Dockerfile



```

Dockerfile
1  FROM node:hydrogen-alpine
2
3  WORKDIR /app
4  COPY package.json .
5  RUN npm install
6  COPY . .
7  EXPOSE 3000
8
9  CMD ["node", "server.js"]
  
```

The screenshot shows a terminal window with a nano editor containing a Dockerfile. The Dockerfile starts with "FROM node:hydrogen-alpine", sets the working directory to "/app", copies "package.json" to the container, runs "npm install", copies the current directory contents to the container, exposes port 3000, and runs the application with "node server.js".

* FROM : correspond à l'image qu'on va utiliser. Hydrogen correspond à la version 18.x de node.

* WORKDIR : permet de dire à docker d'utiliser cette route comme location par défaut pour toutes ses commandes

* COPY : sert à copier le dossier choisi dans l'app

* RUN : sert à indiquer la commande à exécuter

* COPY . . : permet d'indiquer qu'on copie tout sur l'application

* EXPOSE : permet d'indiquer le port sur lequel on est

* CMD : sert à dire ce que l'on veut faire une fois qu'on a notre image dans notre container

Une fois qu'on a bien initialiser nos fichiers, on fait la commande : `sudo docker compose up -d -build`

Cette commande va servir à build notre api.

Comment faire pour mettre à jour son api ?

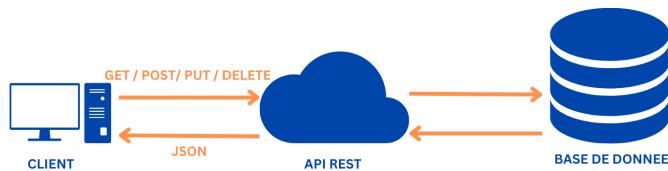
Il suffit de retourner dans le dossier .ssh, ensuite de faire la commande qui on permet d'accéder au serveur. On se redirige alors dans le dossier ou se trouve notre api, on fait `git pull`. Il ne faut pas oublier de refaire un `sudo docker compose up -d -build`. Et voilà !

Ensuite en ligne de commande on fait la commande :

- API Rest
 - Mise en contexte

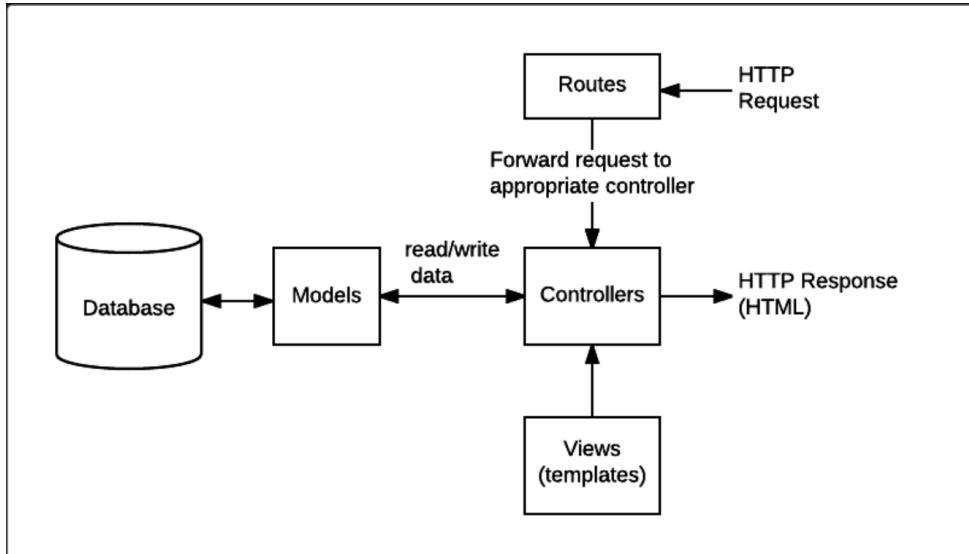
Qu'est ce qu'une API Rest ?

L'api va permettre de faire le lien entre la base de donnée et l'appli. Lorsqu'elle est Restful, cela concerne un modèle d'architecture propre.



Le rôle des routes et du controller :

Comme le schéma ci-dessous le montre, le controllers va transmettre les requêtes, ainsi que les réponses de la base de données. Quant aux routes, elles vont mener à différents endroits.



- Création API Rest

Pour commencer, on va créer l'API en utilisant Visual Studio Code, qui est un éditeur de code, ensuite node.js doit être installé afin d'avoir les modules associés au fonctionnement de l'API, puis Postgres qui va nous permettre d'avoir accès à notre serveur.

on devons créer un dossier dans lequel tous les fichiers liés à l'API seront, puis en l'ouvrant via VS Code, depuis terminal on lance la commande :

```
npm init -y
```

Celle-ci va créer le fichier “package.json”, qui permettre d’installer toutes les dépendances

```
{} package.json > ...
1  {
2    "name": "plenty--api",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    ▷ Debug
7    "scripts": {
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "keywords": [],
11   "author": "",
12   "license": "ISC",
13   "dependencies": {
14     "express": "^4.18.2",
15     "pg": "^8.9.0"
16   }
17 }
```

Ensuite, on lance la commande :

```
npm i --save express
```

Cela va permettre d'installer le package “express”, un dossier “node_modules” sera également installé pour faire fonctionner l’API à l'aide des dépendances. Un autre fichier est créé parallèlement, c'est “package.lock.json”, celui-ci est le même principe que “package.json”.

Pour en finir avec les installations, on lance la commande :

```
npm i pg
```

Il s'agit d'une sorte de connecteur avec la base de données Postgresql.

Toutes les dépendances sont installées, donc on peut véritablement commencer notre API Rest.

On crée un fichier “server.js”, dans lequel on va créer le serveur à l'aide du framework express et appeler la dépendance. Puis une variable pour le port que l'on va utiliser, ici c'est 3000 mais cela peut également être 8080 ou autres.

```
1  const express = require("express");
2  const app = express();
3  const port = 3000;
4
```

Puis, on crée une fonction qui va dire sur quel port on est.

```
app.listen(port, () =>
  console.log(`app listening on port ${port}`));
```

On peut maintenant tester en lançant depuis le terminal :

```
marinericher@MacBook-Pro-2 Plenty-API % node server.js
app listening on port 3000
```

Ensuite, si on allons sur : localhost:3000/

On aura un message “CANNOT GET”, c'est pour cela que on allons créer une requête :

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

Si on retourne sur le site, le “Hello World !” est affiché.

Pour préciser, req permet d'envoyer les requêtes et res les réponses.

Users

CREATION TABLE

Maintenant, on va dans notre serveur sur le terminal ‘PSQL’. Après s'être identifié on crée la table “users”. Pour cela on s'assure d'être au bon endroit avec

```
\conninfo
```

Ensuite on créa la table :

```
CREATE TABLE users (
    ID SERIAL PRIMARY KEY,
    username VARCHAR(255),
    Password VARCHAR(255));
```

ID est un identifiant qui va caractériser chaque user, cela va permettre de les retrouver plus facilement.

VARCHAR permet de définir un nombre de caractères pour les noms sur une variable.

On va ajouter des valeurs pour les futurs tests :

```
INSERT INTO users (username, password)
VALUES ('marine', 'motdepasse'), ('kan', 'autres');
```

On revient sur Visual Code, dans lequel on va créer un fichier “db.js” (db signifie database, et base de donnée en français).

Il va permettre de se connecter à la base de données, donc on va appeler ‘pg’, qui est Postgres. Et ensuite, on rentre tous les identifiants qui vont permettre de se connecter.

```
JS db.js      X
Plenty-API > JS db.js > ...
1 const Pool = require("pg").Pool;
2
3 const pool = new Pool({
4   host: "51.195.44.176",
5   user: "kan",
6   password: "example",
7   database: "plenty",
8   port: 5432,
9 });
10
11 module.exports = pool;
```

Il apparaît également sous la forme d'un ".env", sur le code de Plenty, qui est le front-end. Ce fichier est essentiel pour établir les connexions.

On crée un dossier "src", celui-ci va contenir un dossier pour chaque table. Ici, on peut mettre "users", qui est la première table qu'on a créée. Il contient les fichiers "controller.js", "routes.js" et "queries.js".

Dans le fichier "routes.js", celui-ci va contenir toutes les routes des users. Après avoir appelé les dépendances dont on a besoin, on va créer une requête "get" pour les users, comme pour "server.js".

```
const { Router } = require("express");
const router = Router();

router.get("/", (req, res) => {
  res.send("route users");
});

module.exports = router;
```

On ajoute dans "server.js", la route avec :

```
const usersRoutes = require(`./src/users/routes`);

app.use("/users", usersRoutes);
```

Et on peut tester, pour vérifier que c'est fonctionnel en lançant le serveur, "node server.js" et sur l'adresse : <http://localhost:3000/users>

Il apparaît "routes users", donc on peut continuer.

On va dans "controller.js", qui est lui qui s'occupe des "callbacks".

On commence par appeler la fonction "pool", qui sert de connexion avec PostgreSQL.

```
const pool = require("../db");
//on ajoute queries qui va nous servir prochainement
const queries = require("./queries");
```

AVOIR TOUS LES USERS

On va créer notre première fonction qui va nous permettre de récupérer tous les users avec “get”.

```
const getUsers = (req, res) => {

  /* queries.getUsers est dans "queries.js", dans lequel nous allons
  selectionner tous les users */

  pool.query(queries.getUsers, (error, results) => {
    if (error) throw error;
    res.status(200).json(results.rows);
  });

};

module.exports = {
  getUsers,
}
```

Puis dans routes, pour faire le lien avec le controller, on ajoute :

```
const controller = require("./controller");
```

Mais on va également appeler la fonction “getUsers” qui remplace la fonction get qui a servi de tester le bon fonctionnement de la route précédemment.

```
router.get("/", controller.getUsers);
```

Pour sélectionner tous les users depuis le terminal PSQL, il faut écrire :

```
SELECT * FROM users;
```

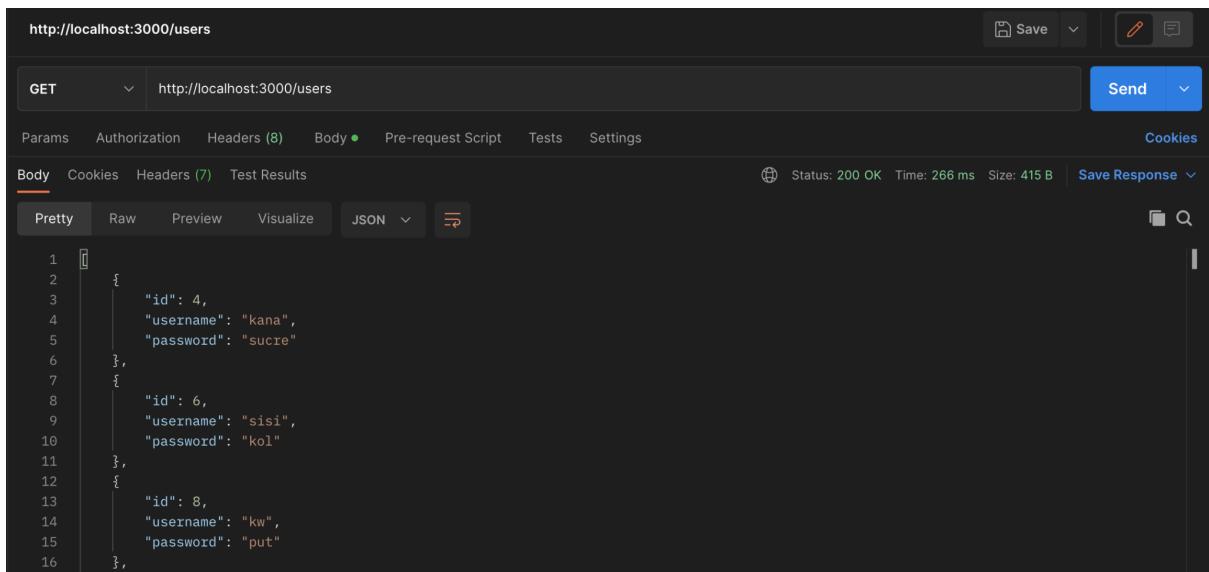
Donc pour les requêtes on va les utiliser. On va dans “queries.js”, pour créer une variable “getUsers” qui va contenir les commandes pour les obtenir :

```
const getUsers = "SELECT * FROM users";
```

On importe également les types de requêtes, ici on avons “getUsers” pour le moment:

```
module.exports = {
  getUsers,
}
```

on pouvons maintenant tester si on récupérons tous les users avec Postman:



The screenshot shows a Postman interface with a GET request to `http://localhost:3000/users`. The response body is a JSON array containing three user objects:

```
[{"id": 4, "username": "kana", "password": "sucré"}, {"id": 6, "username": "sisi", "password": "kol"}, {"id": 8, "username": "kw", "password": "put"}]
```

RÉCUPÉRER UN USER SPÉCIFIQUE PAR SON ID

On remarque ici que cela marche car on récupère tous les users actuellement sur le tableau. On peut donc continuer dans la même direction en cherchant à sélectionner un user spécifique. Avec SQL, pour un trouver un on doit écrire :

```
SELECT * FROM users WHERE id = 4;
```

Il va donc me montrer l'user 4.

On l'ajoute donc dans “queries.js”

```
const getUsersById = "SELECT * FROM users WHERE id = $1";
```

Puis dans routes on crée un router.get :

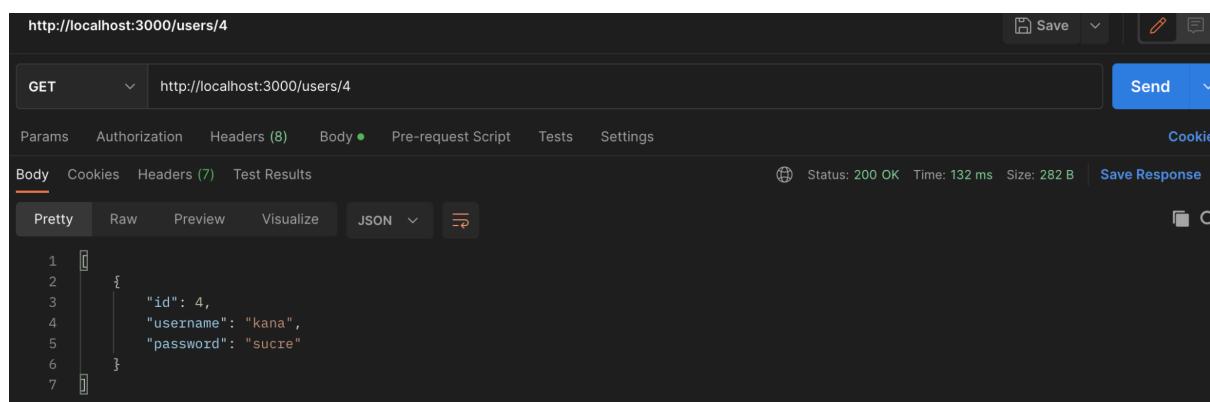
```
router.get("/:id", controller.getUsersById);
```

Dans controller :

```
const getUsersById = (req, res) => {
  const id = parseInt(req.params.id);
  pool.query(queries.getUsersById, [id], (error, results) => {
    if (error) throw error;
    res.status(200).json(results.rows);
  });
};
```

Il s'agit du même procédé que précédemment, mais afin de trouver l'user en fonction de l'id on a rajouté une variable qui va appliquer la règle pour le montrer.

Après avoir ajouté la fonction getUsersById dans les modules exports de queries et contrôler, on peut tester.



AJOUTER UN USER

Le test est réussi alors l'étape suivante est plus complexe car on va ajouter un user. Mais avant, on va devoir vérifier si l'username existe.

Pour vérifier avec PostgreSQL on utilise :

```
SELECT s FROM users s WHERE s.username = 4
```

on allons le rajouter dans queries :

```
const checkUsernameExists = "SELECT s FROM users s WHERE s.username = $1";
```

Ensuite, dans controller :

```
pool.query(queries.checkUsernameExists, [username], (error, results) => {
  if (results.rows.length) {
    res.send("Username existe déjà");
  }
})
```

Puis, pour ajouter un user en base de donnée :

```
INSERT INTO users (username, password) VALUES ($1, $2)
```

Cela va être dans la variable “addUsers” dans queries, puis dans controller on va créer une requête à partir de celle faite précédemment :

```
const addUsers = (req, res) => {
  const { username, password } = req.body;

  // check if username exist in db
  pool.query(queries.checkUsernameExists, [username], (error, results) => {
    if (results.rows.length) {
      res.send("Username existe déjà");
    }

    // add users to db
    pool.query(queries.addUsers, [username, password], (error, results) => {
      if (error) throw error;
      res.status(201).send("Users créé avec succès !");
    });
  });
};
```

Après avoir ajouter “checkUsernameExists” et “addUsers” dans les modules exporter, on peut vérifier et dans routes :

```
router.post("/", controller.addUsers);
```

Post sert à envoyer des données.

Dans Postman, on met une requête “post” et on ajoute les informations sur un nouvel user.

The screenshot shows two separate Postman requests:

Request 1: POST http://localhost:3000/users

- Method: POST
- URL: http://localhost:3000/users
- Body (JSON):

```
1 "username": "marine",
2 "password": "motdepasse"
```

Response 1: Status: 201 Created Time: 149 ms Size: 260 B

Request 2: GET http://localhost:3000/users/

- Method: GET
- URL: http://localhost:3000/users/
- Body (JSON):

```
1 [
```

Response 2: Status: 200 OK Time: 179 ms Size: 524 B

```
1 [
2   {
3     "id": 8,
4     "username": "kw",
5     "password": "put"
6   },
7   {
8     "id": 9,
9     "username": "shelby",
10    "password": "put"
11  },
12  {
13    "id": 10,
14    "username": "marine",
15    "password": "motdepasse"
16  },
17]
```

L'username a bien été créé.

On va également vérifier dans le cas où l'user existe déjà.

The screenshot shows the Postman interface. The request URL is `http://localhost:3000/users`. The method is `POST`. The `Body` tab is selected, showing the following JSON payload:

```

1  {
2   "username": "marine",
3   ... "password": "motdepasse"
4 }

```

The response status is `200 OK`, and the message in the body is `Username existe déjà`.

SUPPRIMER UN USER

On va maintenant supprimer un user :

Via le terminal ce sera :

```
DELETE FROM users WHERE id = 1;
```

c'est le même procédé que les étapes précédentes, ici cela va être la requête va être `DELETE` donc dans route on ajoute :

```
router.delete("/:id", controller.removeUsers);
```

dans queries on ajoute :

```
const removeUsers = "DELETE FROM users WHERE id = $1";
```

et également “removeUsers” dans l’export, puis dans Controller on va utiliser “getUsersById” pour le retrouver.

```

const removeUsers = (req, res) => {
  const id = parseInt(req.params.id);

  pool.query(queries.getUsersById, [id], (error, results) => {
    const noUserFound = !results.rows.length;
    if (noUserFound) {
      res.send("L'User n'existe pas dans la base de donnée");
    }

    pool.query(queries.removeUsers, [id], (error, results) => {
      if (error) throw error;
      res.status(200).send("User Supprimé avec succès");
    });
  });
};

```

On va sur postman vérifier qu' on peut supprimer l'user en n'oubliant pas de lancer le serveur.

The screenshot shows a Postman request configuration for a DELETE operation at `http://localhost:3000/users/11`. The response status is `200 OK` with a response body containing the message `User Supprimé avec succès`.

L'user n'existe plus !

The screenshot shows a Postman request configuration for a GET operation at `http://localhost:3000/users`. The response status is `200 OK` and the response body is a JSON array of three user objects:

```

[{"id": 4, "username": "kana", "password": "sucré"}, {"id": 6, "username": "sisi", "password": "kol"}, {"id": 8, "username": "kw", "password": "put"}, {"id": 9, "username": "shelby", "password": "put"}]

```

MODIFIER MOT DE PASSE D'UN USER

La dernière étape va être de pouvoir changer de mot de passe pour un user, en utilisant PUT.

Avec SQL, on utilise :

```
UPDATE users SET password = motdepasse WHERE id = 1;
```

On le rajoute dans queries :

```
const updateUser = "UPDATE users SET password = $1 WHERE id = $2";
```

Puis une route

```
router.put("/:id", controller.updateUser);
```

Dans controller, on va également utiliser “getUserById” pour le retrouver :

```
const updateUser = (req, res) => {
  const id = parseInt(req.params.id);
  const { password } = req.body;

  pool.query(queries.getUsersById, [id], (error, results) => {
    const noUserFound = !results.rows.length;
    if (noUserFound) {
      res.send("L'User n'existe pas dans la base de donnée");
    }

    pool.query(queries.updateUser, [password, id], (error, results) => {
      if (error) throw error;
      res.status(200).send("Mot de passe modifié avec succès");
    });
  });
};
```

et c'est on va faire le test final !

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:3000/users/10`. The method is `PUT`. The Body tab is selected, showing the following JSON payload:

```

1 {
2   ...
3   "password": "passe"

```

The response status is `200 OK`, time `147 ms`, and size `262 B`. The response body is:

```

1 Mot de passe modifié avec succès

```

Notre table Users est terminée !

Product

Le principe est le même que pour les utilisateurs. On va mettre les principales fonctions essentielles. Toutes les requêtes ont été testées avec Postman.

- Pour ajouter un produit

`http://51.195.44.176:3001/products/`

```

//queries.js
const getProducts = "SELECT * FROM products";

//controllers.js
//permet d'afficher tous les produits
const getProducts = (req, res) => {
  pool.query(queries.getProducts, (err, result) => {
    if (err) throw err;
    console.log("want data")
    res.status(200).json(result.rows)
  })
}

```

- Pour récupérer tous les produits stockés dans la base de données

`http://51.195.44.176:3001/products/`

```
JavaScript ▾
```

```
//queries.js
const addProduct = "INSERT INTO products (name, image, price, store, tag) VALUES
($1, $2, $3, $4, $5)"

//controllers.js
//permet d'ajouter un produit
const addProduct = (req, res) => {
    const { name, price, store, image, tag } = req.body

    pool.query(queries.addProduct, [name, image, price, store, tag], (err, result) => {
        if (err) throw err
        console.log("product added")
        res.status(201).send("Produit ajouté avec succès")
    })
}
```

- Pour modifier un produit

```
http://51.195.44.176:3001/products/:id
```

```
//queries.js
const editProduct = "UPDATE products SET name = $2, image = $3, price = $4, store = $5, tag = $6 WHERE id = $1"

//controllers.js
//permet de modifier un produit
const editProduct = (req, res) => {
    const id = parseInt(req.params.id)
    const { name, price, store, image, tag } = req.body

    pool.query(queries.editProduct, [id, name, image, price, store, tag], (err, result) => {
        if (err) throw err
        console.log("product edited")
        res.status(201).send(result)
    })
}
```

- Pour supprimer un produit

```
http://51.195.44.176:3001/products/:id
```

```

//queries.js
const deleteProduct = "DELETE FROM products WHERE id = $1"

//controllers.js
//permet de supprimer un produit
const deleteProduct = (req, res) => {
  const id = parseInt(req.params.id);
  pool.query(queries.deleteProduct, [id], (error, results) => {
    const noProdFound = !results.rows.length;
    if (noProdFound) {
      res.send("L'User n'existe pas dans la base de donnée");
    }
    pool.query(queries.deleteProduct, [id], (error, results) => {
      if (error) throw error;
      res.status(200).send("User Supprimé avec succès");
    });
  });
}

```

- On peut également rajouter une fonction pour chercher des produits avec leurs nom :

<http://51.195.44.176:3001/products/:name>

```

//queries.js
const getProductName = "SELECT * FROM products WHERE name = $1"

//controllers.js
//permet d'afficher les produits avec le même nom
const getProductName = (req, res) => {
  const name = req.params.name
  pool.query(queries.getProductName, [name], (err, result) => {
    if (err) throw err;
    res.status(200).json(result.name)
  })
}

```

```

{
  "0": {
    "id": 3,
    "name": "Lait",
    "image": "",
    "price": 0.8,
    "store": "carrefour",
    "tag": "dairy products",
    "reports": 4
  }
}

```

La requête fonctionne sur internet, on peut donc récupérer les produits.

- Si on veut rendre notre application dynamique, il faut qu'on soit capable de dire si le produit existe toujours ou non. On ajoute un système d'avertissement, donc pour ajouter des avertissements :

<http://51.195.44.176:3001/products/reports/:id>

```

//queries.js
const addReport = "UPDATE product SET reports = reports + 1 WHERE id = $1"

//controllers.js
//permet d'ajouter un avertissement sur le produit afin de l'effacer au bout d'un certain nombre
const addReport = (req, res) => {
  const id = parseInt(req.params.id)

  pool.query(queries.addReport, [id], (err, result) => {
    if (err) throw err
    res.status(201).send(result)
  })
}

```

● Frontend

- Organisation des pages

on avons choisi de répartir les tâches de la manière suivante :

- Marine s'occupe de toutes les pages, composants et appels API concernant les utilisateurs.
- Kagnana s'occupe de toutes les pages, composants et appels API concernant les produits.

Comment fonctionne React Native ?

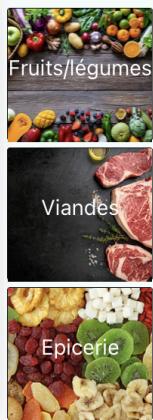
Tout comme React, React native fonctionne avec des “Class Component” ou des “Function Component”. La syntaxe utilisée dans notre projet est le JSX qui va permettre de structurer le rendu de composants.

- packages utilisés
 - react-router-native : permet de gérer les routes de notre application
 - react-native-dropdown-picker : permet de créer des select pour pouvoir sélectionner les catégories de produits spécifique
 - fontAwesome avec le package react-native-fontawesome : librairie d'icône
 - react-native-image-picker : permet de rechercher des images depuis le téléphone
 - react-hook-form : permet de créer et de gérer un formulaire
- Visuel des pages (iOS)

Page d'accueil, en fonction des sections cliquées ça met automatiquement les filtres correspondant.

19:58

4G



Ajouter un produit



Lorsque l'on veut ajouter un produit :

19:58

4G

Mettre l'url d'une image

Nom du produit

Magasin

Prix

Catégories



Ajouter un produit



20:00

4G

Catégories



Retrouver ici tous nos produits les moins chers



Lait
carrefour
0.80 €



Patate 1kg
Aldi
0.90 €



Sauce tomate
Carrefour
0.99 €



20:00

4G

Catégories	^
légumes	
fruits	
viandes	
produits laitiers	
divers	

Sauce tomate
Carrefour
0.99 € !



19:59

4G

An item has been selected



Retrouver ici tous nos produits les moins chers



Sauce tomate
Carrefour
0.99 €



On peut choisir les catégories que l'on souhaite avoir pour trier tous les produits.

19:58

4G

Votre recherche

Chercher



Rechercher tous les produits par leur nom



Explication des composants

Home page :

```
9   <View style={styles.container}>
10     <View style={styles.textContainer}>
11       <View style={styles.productContainer}>
12         <TouchableOpacity onPress={() => navigate('/products')}>
13           <ImageBackground style={{ height: "100%", borderRadius: 5, borderWidth: 0.7 }} source={{ uri: "https://img.freepik.com/photos-gratuite/photographie-alimentaire-differents-fruits-legumes_1195-1195.jpg" }}>
14             <Text style={{ fontSize: 20, color: "white", margin: 10, fontWeight: "bold" }}>Retrouvez ici tous les produits les moins chers</Text>
15           </ImageBackground>
16         </TouchableOpacity>
17       </View>
18     </View>
19     <View>
20       <TouchableOpacity onPress={() => navigate('/products/#fruitNvegetable', { state: { tags: ["vegetables", "fruits"] } })}>
21         <ImageBackground style={{ borderRadius: 2, borderWidth: 0.7, height: 87, marginBottom: 4 }} source={{ uri: "https://blendsmooth.b-cdn.net/wp-content/uploads/2020/12/fruits-legumes-1195-1195.jpg" }}>
22           <Text style={styles.text}>Fruits/légumes</Text>
23         </ImageBackground>
24       </TouchableOpacity>
25       <TouchableOpacity onPress={() => navigate('/products/#meat', { state: { tags: ["meat"] } })}>
26         <ImageBackground style={{ borderRadius: 2, borderWidth: 0.7, height: 87, marginBottom: 4 }} source={{ uri: "https://media.istockphoto.com/id/1288461867/fr/photo/varie%C3%A9-de-viandes-1195-1195.jpg" }}>
27           <Text style={styles.text}>Viandes</Text>
28         </ImageBackground>
29       </TouchableOpacity>
30       <TouchableOpacity onPress={() => navigate('/products/#divers', { state: { tags: ["divers"] } })}>
31         <ImageBackground style={{ borderRadius: 2, borderWidth: 0.7, height: 87 }} source={{ uri: "https://www.mgc-prevention.fr/wp-content/uploads/2016/01/fruits-secs_35177746.jpg" }}>
32           <Text style={styles.text}>Epicerie</Text>
33         </ImageBackground>
34       </TouchableOpacity>
35     </View>
36     <Button title="Ajouter un produit" onPress={() => navigate('/form/product')}></Button>
37   </View>
```

Plusieurs composants sont utilisées ici :

- <TouchableOpacity> : permet de créer des zones dans lesquelles on peut interagir.
Lorsque l'on touche la zone, elle on redirige vers une page
- <ImageBackground> : comme son nom l'indique, cette balise permet d'ajouter une image en tant que fond.
- <View> : balise qui se comporte comme la balise html <div>

Comment faire le css ?

Dans la même page que notre fonction composant, on donne une constante style qui va permettre de jouer le rôle d'un fichier css avec StyleSheet de React Native. Voici un exemple :

```
41 //style
42 const styles = StyleSheet.create({
43   container: {
44     display: "flex",
45     flexDirection: "column",
46     justifyContent: "center",
47   },
48   textContainer: {
49     display: "flex",
50     flexDirection: "row",
51     marginBottom: 40,
52   },
53   productContainer: {
54     textAlign: "center",
55     width: "70%",
56     height: 270,
57     marginRight: 10,
58     shadowColor: 'black',
59     shadowRadius: 7,
60     shadowOpacity: 0.2,
61     elevation: 5
62   },
63 },
64 },
```

Form page :

```
64      <KeyboardAvoidingView
65        |   behavior={Platform.OS === "ios" ? "padding" : "height"}
66      >
67        <TouchableWithoutFeedback onPress={Keyboard.dismiss}>
68          <View>
69            <Button title='Ajouter une image' onPress={uploadImage} />
70            <View style={styles.inputContainer}>
71              <Text style={styles.inputTitle}>Nom du produit</Text>
72              <Input name="name" control={control} />
73            </View>
74            <View style={styles.inputContainer}>
75              <Text style={styles.inputTitle}>Magasin</Text>
76              <Input name="store" control={control} />
77            </View>
78            <View style={styles.inputContainer}>
79              <Text style={styles.inputTitle}>Prix</Text>
80              <Input name="price" control={control} />
81            </View>
82            <View style={styles.inputContainer}>
83              <DropDownPicker
84                |   open={open}
85                |   style={{ marginBottom: 20 }}
86                |   value={selectedTags}
87                |   items={items}
88                |   setOpen={setOpen}
89                |   setValue={ setSelectedTags }
90                |   setItems={setItems}
91                |   containerStyle={{ backgroundColor : "white" }}
92            >
93          </View>
94          <View style={styles.containerButton}>
95            <Button
96              |   title="Ajouter un produit"
97              |   onPress={handleSubmit(onSubmit)}
98            >
99            <Button
100               |   title="Retour"
101               |   onPress={() => navigate('/')}
102             >
103           </View>
104         </View>
105       </TouchableWithoutFeedback>
```

Composants utilisés :

- <KeyboardAvoidingView> : permet de gérer l'affichage de la page en fonction de si le clavier est activé ou non. Pour que ça fonctionne il faut importer **Keyboard** de React Native.
- <TouchableWithoutFeedback> : permet de créer une zone avec laquelle on peut interagir sans forcément vouloir de feedback derrière
- <DropDownPicker> : un select qui permet de choisir le type de catégorie du produit. Ce n'est pas un élément qui fait partie de React Native naturellement, il faut installer le package.

```

42     const onSubmit = data => {
43         const sentData = {
44             name: data.name,
45             image: data.image || "",
46             price: data.price,
47             store: data.store,
48             tag: selectedTags
49         }
50         Alert.alert(JSON.stringify(sentData))
51         axios.post("http://51.195.44.176:3001/products", sentData,
52         {
53             headers: {
54                 "Content-Type": "application/json"
55             }
56         })
57     }

```

Voici la fonction appelée lorsque l'on valide le formulaire. On forme les données envoyées pour ensuite l'envoyer dans l'api. On lui donne en headers "Content-Type" pour lui préciser quel type de ressource on lui envoie.

Display Product :

On utilise des variables globales, des variables que l'on mets dans le "state" de la fonction composant. On utilise le hook "useState" qui permet d'initialiser la variable. useState prends le nom de la variable ainsi que la fonction que l'on va appeler pour modifier cette variable (par convention on mets le nom de la variable + "set" devant)

```

17 |     const location = useLocation()
18 |     const [productList, setProductList] = useState([])
19 |     const [selectedTags, setSelectedTags] = useState(location.state !== null ? location.state.tags : [])
20 |     const [open, setOpen] = useState(false)
21 |     const [items, setItems] = useState(product)

```

```

view > JS DisplayProducts.js > DisplayProducts > useEffect() callback > then() callback
39     return (
40         <View style={styles.container}>
41             <DropDownPicker
42                 open={open}
43                 style={{ marginBottom: 20 }}
44                 multiple={true}
45                 min={0}
46                 max={4}
47                 value={selectedTags}
48                 items={items}
49                 setOpen={setOpen}
50                 setValue={setSelectedTags}
51                 setItems={setItems}
52             />
53             <Text>Retrouver ici tous nos produits les moins chers</Text>
54
55             {productList.map((product, key) => {
56                 return selectedTags.length === 0 ? (
57                     <CardProduct
58                         image={product.image || none}
59                         name={product.name}
60                         store={product.store}
61                         price={product.price}
62                     />
63                 ) : (
64                     selectedTags.map((tag, key) => {
65                         return tag === product.tag ? (
66                             <CardProduct
67                                 image={product.image}
68                                 name={product.name}
69                                 store={product.store}
70                                 price={product.price}
71                             />
72                         ) : null
73                     })
74                 )
75             })}
76         </View>
77     )
78
79 }
80
81 }

```

Composants utilisés :

- <CardProduct> : composant que l'on a créé pour faciliter la mise en forme des données

```

21     return (
22       <View style={styles.container}>
23         <Image style={styles.image} source={{ uri: props.image }} />
24         <View style={styles.column}>
25           <Text>{props.name}</Text>
26           <Text>{props.store}</Text>
27           <Text>{props.price}</Text>
28         </View>
29         <TouchableOpacity style={styles.signal} onPress={() => addReport(props.id)}>
30           <FontAwesomeIcon
31             size={30}
32             icon="faExclamation"
33             style={styles.icon}
34           />
35         </TouchableOpacity>
36       </View>
37     )

```

Avec React on peut passer des props aux composants enfants. C'est avec ce procédé que l'on peut créer des composants propres que l'on peut utiliser partout dans notre site internet ou notre application mobile.

- Ici encore, on a le <DropDownPicker> qui a la même fonction que dans formPage.
- On utilise la fonction .map() qui va permettre d'afficher une CardProduct pour chaque produit que l'on a dans la variable **selectedTags**. Elle sert également à trier les produits en fonction des tags sélectionnés.
- On retrouve <TouchableOpacity> qui est utilisé ici pour permettre aux utilisateurs de signaler si un produit n'est plus en rayon lorsque l'on appuie sur le !.

Search Page :

```

16   //initialisation fonction
17   const getProducts = async (name) => {
18     const response = await axios.get("http://51.195.44.176:3001/products/" + name)
19     console.log(response.data)
20     setProductList(response.data)
21   }
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47   <View>
48     {productList === "" || productList.length === 0 ? (
49       <View>
50         {productList.map((product, key) => (
51           <CardProduct
52             id={product.id}
53             reports={product.reports}
54             image={product.image}
55             name={product.name}
56             store={product.store}
57             price={product.price}
58             getProducts={getProducts()}
59           />
60         )));
61       </View>
62     ) : (
63       <Text>Aucun résultat ....</Text>
64     )}
65   </View>

```

La liste ne s'affiche pas tout de suite, on attend que l'utilisateur fasse d'abord la recherche. Ensuite si la requête retourne des résultats on les affiche

Footer :

```
30      <View>
31          {keyboardStatus === false ? (
32              <View style={styles.container} >
33                  <TouchableOpacity
34                      onPress={() => navigate('/')}>
35                      <FontAwesomeIcon
36                          size={40}
37                          icon={faHouse}
38                          style={styles.icons}
39                      />
40                  </TouchableOpacity>
41                  <TouchableOpacity
42                      onPress={() => navigate('/')}>
43                      <FontAwesomeIcon
44                          size={40}
45                          icon={faUser}
46                          style={styles.icons}
47                      />
48                  </TouchableOpacity>
49                  <TouchableOpacity
50                      onPress={() => navigate('/')}>
51                      <FontAwesomeIcon
52                          size={40}
53                          icon={faGear}
54                          style={styles.icons} />
55                  </TouchableOpacity >
56              </View>
57          ) : null}
58      </View>
59  )
60 }
61 //style
```

Ici en plus des composants qu'on a vu avant, on utilise `<FontAwesomeIcon>` qui est un package qui sert à insérer des images provenant du site Font Awesome. Il faut également télécharger le package contenant les icônes gratuite et les importer comme par exemple `faGear`.

USERS

on va voir en détail comment créer une page pour se connecter, pour les pages d'inscriptions et de changement de mot de passe ce sera un peu le même procédé. C'est pour cela que la partie connexion va servir de modèle.

Connexion :

Dans App.jsm j'ajoute la route users dans la fonction App :

```
<Route exact path="/users" elements={<UserProfil />} />
```

J'importe ma fonction du profil user:

```
import UserProfil from "./view/users";
```

Je crée mon fichier SignIn (connexion) dans views et j'ajoute une fonction

```
import React from "react";
import { useNavigate } from "react-router-native"

const SignIn = () => {

  // va nous permettre d'utiliser les routes
  const navigate = useNavigate()
  return (
    <Text>Hello</Text>
  )
}
```

Le texte va me permettre de vérifier que ça marche

Dans footer.js, J'indique d'aller dans users.js lors de clique sur le personnage en bas sur l'appli.

```
<TouchableOpacity
  onPress={() => navigate('/users')}
  >
  <FontAwesomeIcon
    size={40}
    icon={faUser}
    style={styles.icons}
  />

```

on pouvons commencer notre écran de connexion:

on ajoute le logo en important depuis react-native et l'image.

```
import { Text, View, StyleSheet, Image, ScrollView } from 'react-native';
import Logo from '../assets/image/Logo.png';
```

On va remplacer le texte “hello” par notre image :

```
const SignIn = () => {
  const navigate = useNavigate();
  return (
    <ScrollView>
      <View style={styles.root}>
        <Image
          style={styles.logo}
          source={Logo}
        />
      </View>
    </ScrollView>
```

On ajoute des styles à l’ensemble de l’écran en appelant root et logo pour modifier l’allure de l’image.

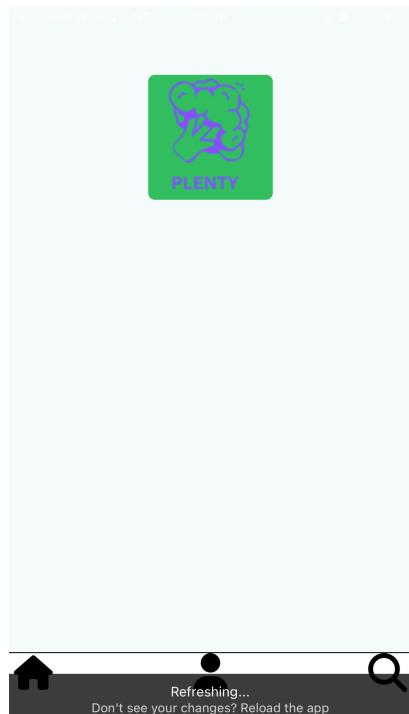
Par ailleurs, ScrollView permet de bouger l’écran de haut en bas.

```

const styles = StyleSheet.create({
  root: {
    //va centrer tout ce qui est sur l'écran
    alignItems: 'center',
  },
  logo: {
    //arrondissement des bordures
    borderRadius: 7,
    //reglage taille image
    maxWidth: 200,
    maxHeight: 200,
    //marges en dehors de l'image pour laisser de la place
    // avec les futurs autres composants
    margin: 60,
  },
});

);

```



Notre page marche parfaitement !

On va créer un composants pour créer les cases dans lequel on va entrer notre username, mot de passe... Il va également servir pour la page d'inscription et de mot de passe oublié

Dans le dossier “component”, je crée un fichier “CustomInput.js”, j’y inscris les lignes de base que l’on retrouve dans tous les fichiers :

```
import React from 'react'
import { View, Text, TextInput, StyleSheet } from 'react-native'

const CustomInput = (
  return (
    <View>
      <Text> Hello </Text>
    </View>

  )
}

export default CustomInput
```

Maintenant que tout est bon, on peut commencer !

on va nous aider de <TextInput />, qui va créer un champ dans lequel on pourra écrire.

On commence par y insérer placeholder qui va être un champ dans lequel on peut écrire.

```

const CustomInput = ({ value, setValue, placeholder, secureTextEntry }) => {
  return (
    <View style={styles.container}>
      <TextInput

        placeholder={placeholder}
        style={styles.input}

      />
    </View>
  )
}

const styles = StyleSheet.create({
  container: {

    // arrière plan du champ à remplir
    backgroundColor: 'white',
    // taille
    minWidth: '90%',
    minHeight: '6%',
    // couleur des bordures
    borderColor: '#e8e8e8',
    //taille bordure
    borderWidth: 1,
    //
    borderRadius: 5,
    // arrondissement bordures (coin qui vont etre arrondie)
    paddingHorizontal: 10,
    marginVertical: 5,
  },
  input: {},
});

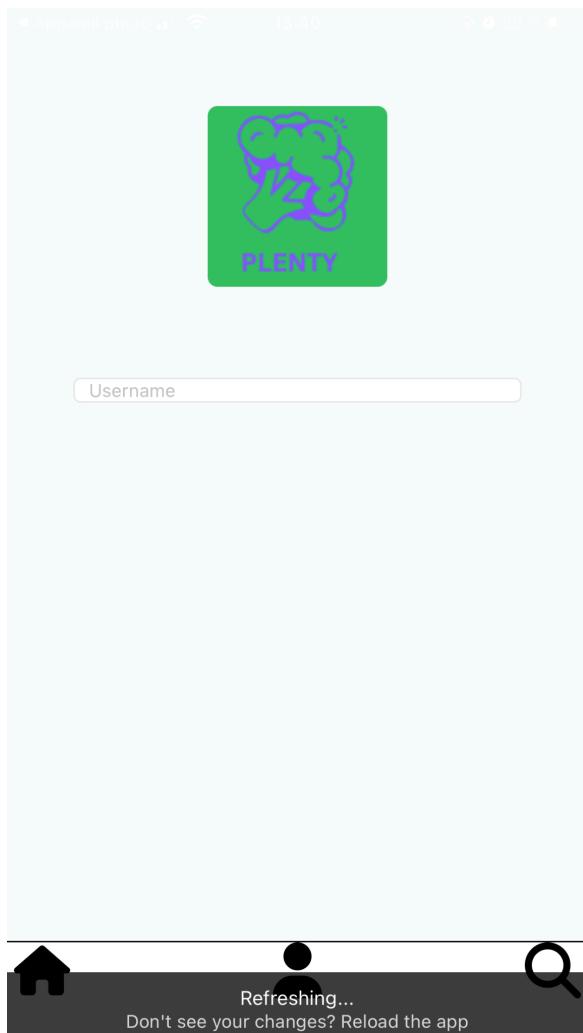
```

Dans signIn.js on ajoute :

```

<CustomInput
  placeholder="Username"
/>

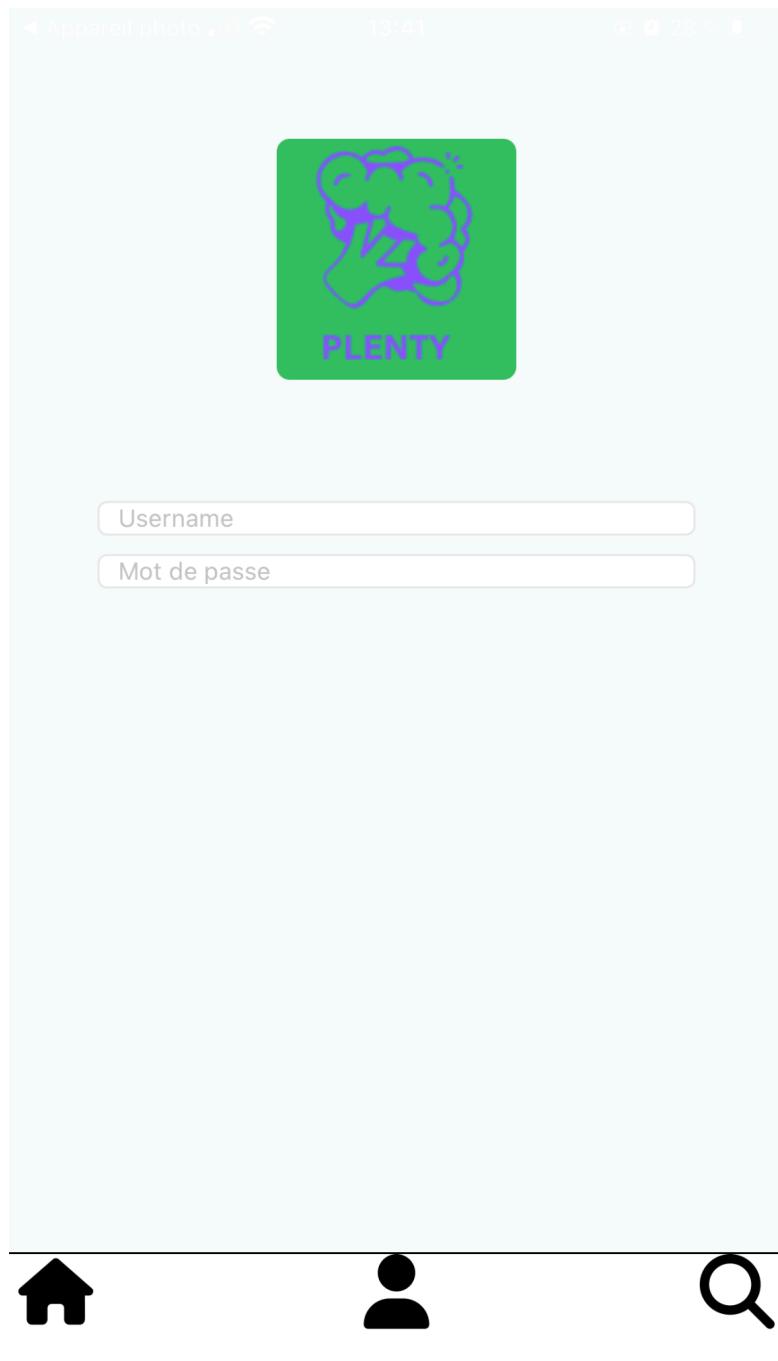
```



Sur l'écran, on a un champ pour les users.

on fait de même pour le champ mot de passe en ajoutant une balise <CustomInput />

```
<CustomInput  
    placeholder="Mot de passe"  
/>
```



On peut maintenant créer nos boutons, on va également créer un composant prévu à cet effet.

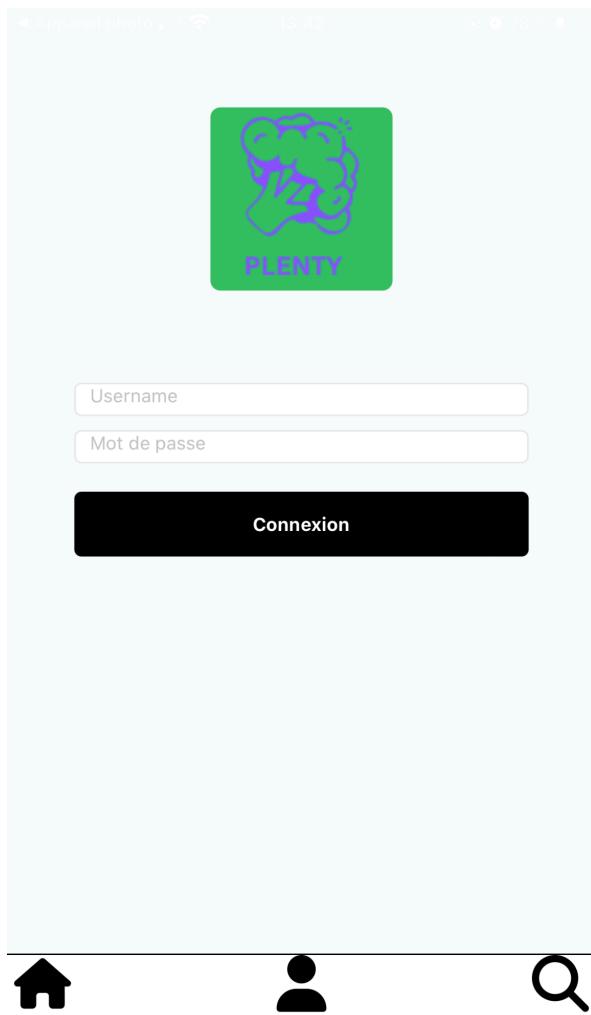
Dans component, on crée un fichier “CustomButton.js”, dans lequel j’insère les éléments de bases (comme pour “CustomInput”)

En n’oubliant pas de l’importer dans mon fichier “SignIn”, je l’insère dans la fonction et peut créer mon bouton en retournant dans mon composant “CustomButton”

Grâce à la div pressable je vais pouvoir créer mon bouton qui va permettre de mener à un endroit

```
return (
    <Pressable

// au click
    onPress={onPress}
// styles qui va changer selon le type que l'on va attribuer dans SignIn
    style={[styles.container, styles[`container_${type}`]]}>
// pareil pour le texte
    <Text style={[styles.text, styles[`text_${type}`]]}>{text}</Text>
</Pressable>
```



Le style donné en fonction des types de containers s'il n'est pas marqué ce sera le PRIMARY.

```
const styles = StyleSheet.create({
  container: {

    minWidth: '90%',

    padding: 15,
    marginVertical: 15,

    alignItems: 'center',
    borderRadius: 5,

  },
  container_PRIMARY: {
    backgroundColor: 'black',
  },
  container_TIERT: {},

  text: {
    color: 'white',
    fontWeight: 'bold',
  },
  text_TIERT: {
    color: 'gray',
  },
})
```

```
<CustomButton
  text="Connexion"
  onPress={onSignInPressed}
/>

<CustomButton
  text="Mot de passe oublié"
  onPress={onForgotPasswordPressed}
  type="TIERT"
/>

<CustomButton
  text="Se connecter avec Google"
  onPress={onSignInGoogle}
/>

<CustomButton
  text="Vous n'avez pas de compte ? Inscivez-vous"
  onPress={onSignUpPress}
  type="TIERT"
/>
```

UsernameMot de passe**Connexion**[Mot de passe oublié](#)[Se connecter avec Google](#)[Vous n'avez pas de compte ? Inscivez-vous](#)

Refreshing...

Don't see your changes? Reload the app

on ajoute les fonction qui vont indiquer les actions à faire lorsque on cliquons sur un bouton :

```
const [username, setUsername] = useState('');
const [password, setPassword] = useState('');

const onSignInPressed = () => {
    console.warn("Remplissez vos identifiants");
}

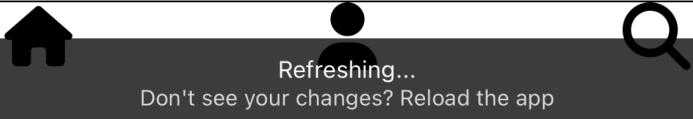
const onForgotPasswordPressed = () => {
    navigate('/connexion/resetpassword');
}

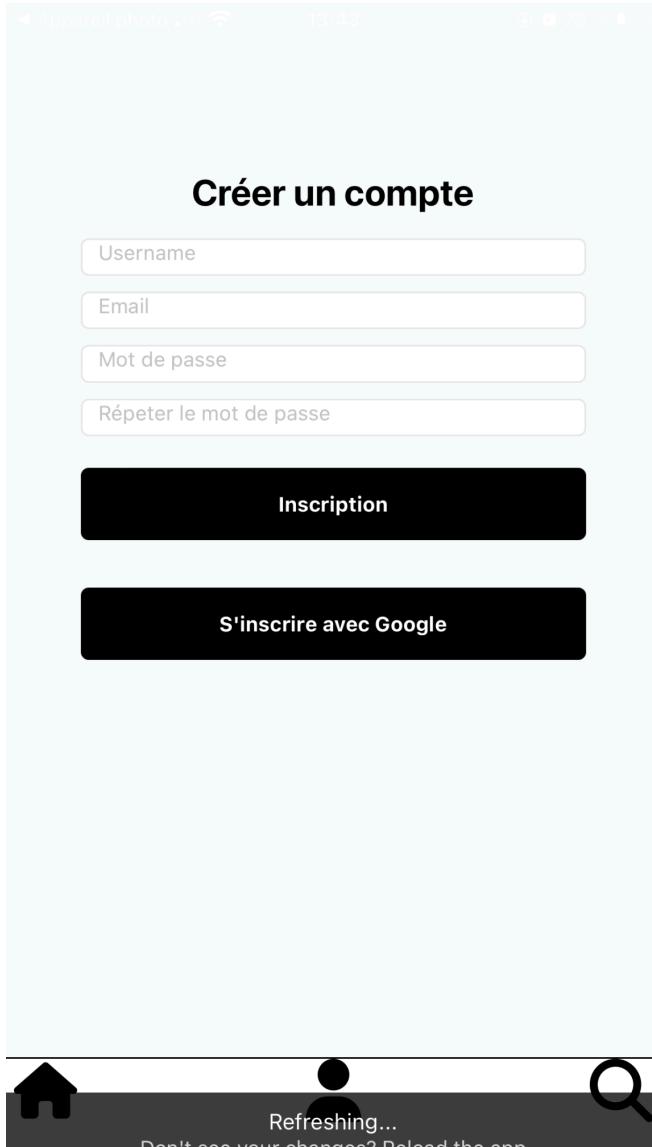
const onSignInGoogle = () => {
    console.warn('onSignInGoogle');
}

const onSignUpPress = () => {
    navigate('/connexion/inscription');
}
```

Inscription et changer le mot de passe :

Pour l'inscription et changer de mot de passe c'est le même procédé en ajustant en fonction du besoin.





Lien Api et Application :

Les requêtes via l'application ne marchent pas. Malgré le bon fonctionnement de l'API comme on a vu avec Postman. Cela doit venir de mon manque d'information dû au fait que je n'ai jamais fait de requête via l'application.

Ce qu'on aimerait faire

Le projet est loin d'être fini. On aimerait pousser notre projet plus loin en ajoutant des fonctionnalités tels que :

- Ajouter des images depuis la galerie pour les produits
- Prendre des images pour les produit
- Ajouter le scan de code barre, en s'aidant de librairie OpenSource
- Crypter les mot de passe
- connexion via google