RICHER MARINE
BTS SIO 2

MEDGENIX

2023

TABLE OF CONTENTS

CAHIER DES CHARGES

01 — Context

Le laboratoire Galaxy Swiss Bourdin (GSB) est issu de la fusion entre le géant américain Galaxy et le conglomérat européen Swiss Bourdin, lui même déjà union de trois petits laboratoires .
Une conséquence de cette fusion, est la recherche d'une optimisation de l'activité du groupe ainsi constitué en réalisant des économies d'échelle dans la production et la distribution des médicaments (en passant par une nécessaire restructuration et vague de licenciement), tout en prenant le meilleur des deux laboratoires sur les produits concurrents.

02 — Expression des besoins

- **Afin de diriger les projets, ils nécessite de faire des échanges réguliers (Appels, mails, message). Et cela empêche pour tous les membres du projet d'avoir une vision clair.**
- **Cela permettrait un gain de temps et une meilleure communication.**

03 — Les enjeux

Les enjeux sont de créer une plateforme qui reste simple d'utilisation qu'elle soit compréhensible.

04 — Analyse fonctionnelle

Medgenix va permettre de gérer les projets, dans un premier temps, nous pouvons nous authentifier, nous inscrire.
Dans un deuxième temps, un project peut-être ajouter, dans lequel on va ajouter des tâches à faire qui sont en cours, à faire ou en progrès.
On peut ajouter une tâche en fonction de son statut.

ENVIRONNEMENT

Ce projet a été créé dans un environnement Windows et MacOs.

Logiciels utilises

Les logiciels qui m'ont permis de faire ce projet sont :

- Visual Studio Code (Version 1.82)
- MariaDB (Version 11.1)
- Postman (Version 10.18)

Framework, langage modules

"MedGenix" est codé avec React (v18)TypeScript et les modules 'vite' (v4.4), 'react-router' (v6.15)pour le frontend. Concernant l'api il s'agit de NodeJS(v18.16) et les modules 'express' (v4.18), 'body-parser' (v1.20), 'bcrypt' (v5.1), jsonwebtoken(v9), mariadb (3.2), 'dotenv'(v16.3) et 'cors' (v2.8). Enfin la base de donnée traite des requêtes en SQL.

Mise en route du projet

L'API se lance avec "node index.js"

```
PS C:\Users\Marine\Desktop\MedGenix-Backend> node index.js
app is listening on port 3003
Connected to MariaDB!
```

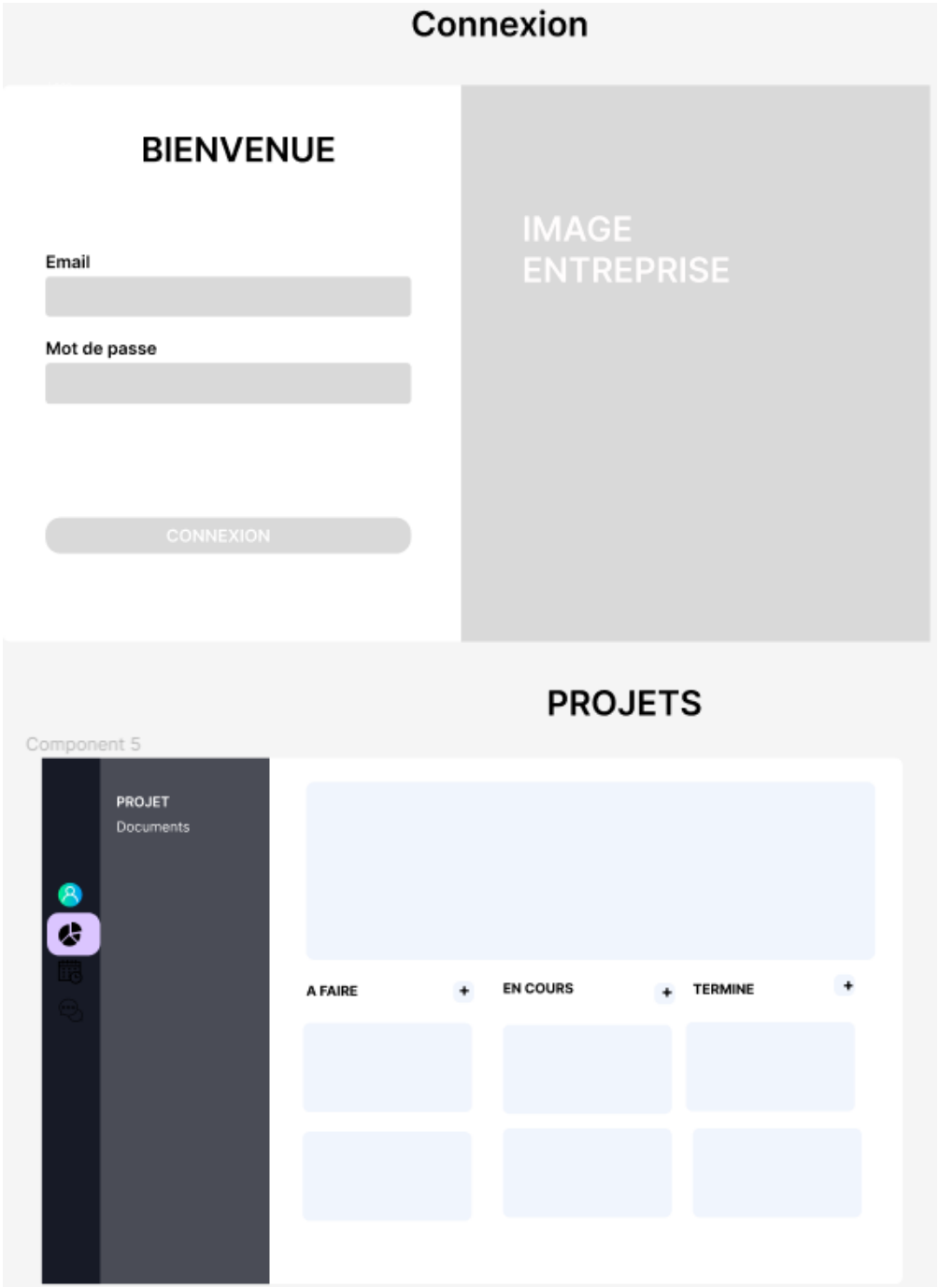
La base de donnée est en locale sur windows.

Le front-end se lance avec la commande "npm run dev"

```
PS C:\Users\Marine\Desktop\MedGenix-front> npm run dev

> medgenix-front@0.0.0 dev
> vite


  VITE v4.4.9  ready in 242 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h to show help
```

FRONT-END

Afin de réaliser la maquette je me suis aidée de “dribbble” pour réaliser la maquette sur “Figma”.



Avec une palette de couleur :

#e3ffc5 #2759ef #dbc5ff #f0f5fd #161a26

Initialisation projet

Vite permet de créer le projet en choisissant du React TypeScript avec cette commande dans la console en se déplacant dans l'endroit où le projet va être implémenté.

```
npm create vite@latest
```

Routes

On installe tous les modules, avec npm.

```
npm i react-router
```

Dans "App.tsx", on va importer les composants que l'on veut. Ici c'est :

```
import { Routes, Route } from 'react-router-dom';
```

On créer une route avec ses module, en appelant le composant / la fonction qui sera représentée. Cette fonction doit également être importée.

```
import HomePage from './Page/HomePage'
                    <Route>
<Route path="/" element={<HomePage />} />
                   </Routes>
```

Page d'accueil

La page d'accueil est sous le nom de "HomePage", dans lequel on retrouve le component "**NavHomePage**". Celui-ci est le menu de la page d'accueil.

Dans ce component on a des Link afin de rediriger vers les autres pages.

```
import { Link } from 'react-router-dom';
<div className={styles.nav}>
<img className={styles.logo} src={Logo} />
<Link to="/signUp" className={styles.item}> <img className=
{styles.ImgItem} src={Account}/> MON COMPTE</Link>
</div>
```

Dans ce component on a importé les 'Link', et il y a un exemple de son utilisation dans lequel on insère le lien vers une route. Ici, le lien mène à la page d'inscription.
On a également un fichier CSS, qui va permettre de faire la mise ne forme du menu.
Avec la classe "nav", on a fait la mise en forme avec mettant une direction, un debut, une couleur, la taille.

```
.nav {
padding-left: 0vh;
display: flex;
justify-content: flex-start;
align-items: flex-start;
flex-direction: column;
background-color: #171A26;
color: aliceblue;
width: 25vh;
height: 100vh;
}
```

Page de gestion de projet

La page de gestion de projet se nomme "ProjectPage", on y retrouve le nom du projet, les tâches à faire.
On a un deuxième menu qui se nomme "NavProjectPage", celui-ci va permettre de passer de projets en projets en allant également dans la page "Documents".

Dans ce menu, inutile d'expliquer le code car il s'agit du même fonctionnement que le premier menu présenté. On fonctionne également avec des "Link".

La page de projet représente donc la liste de tâches à faire, en cours et terminées.

Formulaires d'ajout d'utilisateur, de document, de projet et tâche

Les cinq formulaires sont regroupés car ils utilisent les mêmes components.

- Le premier component est "InputForm.tsx" :

Il représente la partie où un champ de character peut-être insérer avec une balise <input>.
On a le type de donnée, la valeur que l'on va récupérer, 'onChange' permet de représenter la valeur et pour finir "placeholder", va insérer le mot que l'on veut quand la case est vide.

```
<input
type="text"
value={value}
onChange={onChange}
placeholder={placeholder}
/>
```

Le CSS est simple avec la taille (width, height), la marge intérieur (padding).

"focus", va s'appliquer lorsque l'utilisateur clique dans le champ

```
input {
width: 85%;
padding: 1.2rem;
border-radius: 9px;
border: none;
height: 1vh;
}
input:focus {
outline: none;
box-shadow: 0 0 0 4px rgba(253, 242, 233, 0.5);
}
```

Formulaires d'ajout d'utilisateur, de document, de projet et tâche

Une fois importé, on ajuste le component avec les valeurs qui vont initialiser.

Dans ce script, on utilise "useState" de react afin de récupérer les valeur que l'on veut.
Ici, pour se connecter, on a besoin de l'email et du mot de passe de l'utilisateur.
On initialise, useState avec les variabes que l'on veut de l'user.
Seulement une balise est représentée pour mettre l'email mais il y'en a

```
import { useState } from 'react';
const [user, setUser] = useState<User>({ email: '',password: '',});
<InputForm
value={user.email}
onChange={(e) => setUser({ ...user, email: e.target.value })}
placeholder="Email"
/>
```

- Le deuxième component est "CustomButton.tsx" :

Il représente le boutton avec une balise <button>.
Cette balise comporte le type "submit" afin d'envoyer les valeurs vers l'api. Un objet text est créé afin de changer le contenu du bouton.

```
<button type="submit">
```

Formulaires d’ajout d’utilisateur, de document, de projet et tâche

Le code CSS comporte la taille, la couleur, les marges.
“cursor:pointer” signifie quand la curseur est sur le bouton.

```
button{
  background-color: #f48982;
      color: #fdf2e9;
     align-self: end;
  outline: 1px solid #f48982;
    display: inline-block;
    text-decoration: none;
      font-size: 20px;
     font-weight: 400;
    border-radius: 9px;
      border: none;
       width: 98%;
      height: auto;
     padding: 1.2rem;
    cursor: pointer;
  font-family: inherit;
          }
```

Afin se connecter avec la base de donnée, on créé une fonction qui va se connecter avec l'API. Elle traite la réponse et va renvoyer une erreur si la requête ne marche pas.

Cette fonction est appelée avec form.

```
function handleSubmit(e:
FormEvent<HTMLFormElement>) {
e.preventDefault();
console.log('Submitted details:', projet);

// Send the form data to the server

fetch('http://localhost:3003/project/addProject'
, {
method: 'POST',
body: JSON.stringify(projet),
headers: {
'Content-Type': 'application/json',
},
})
.then((response) => response.json())
.then((data) => {
// Handle the response from the
server
console.log('Response:', data);
})
.catch((error) => {
console.error('Error submitting
form:', error);
});
}
......
<form onSubmit={handleSubmit}>
....
</form>
```

Page d'inscription et de connexion

Les pages d'inscription et de connexion sont construites à l'aide des components <InputForm> et <CustomButton>. Afin de faire le lien avec le back-end on a procédé de cette manière.

BACK-END

Base de donnée

MariaDB va permettre de faire la base de donnée "medgenix". Avant de la créer on se connecte sur la console mariadb sur le porrt 3006 avec :

```
mariadb -u root -p
```

On créer la table et l'utilise en insérant les tables préalablement préparées de User, Project, Documents, Tasks.

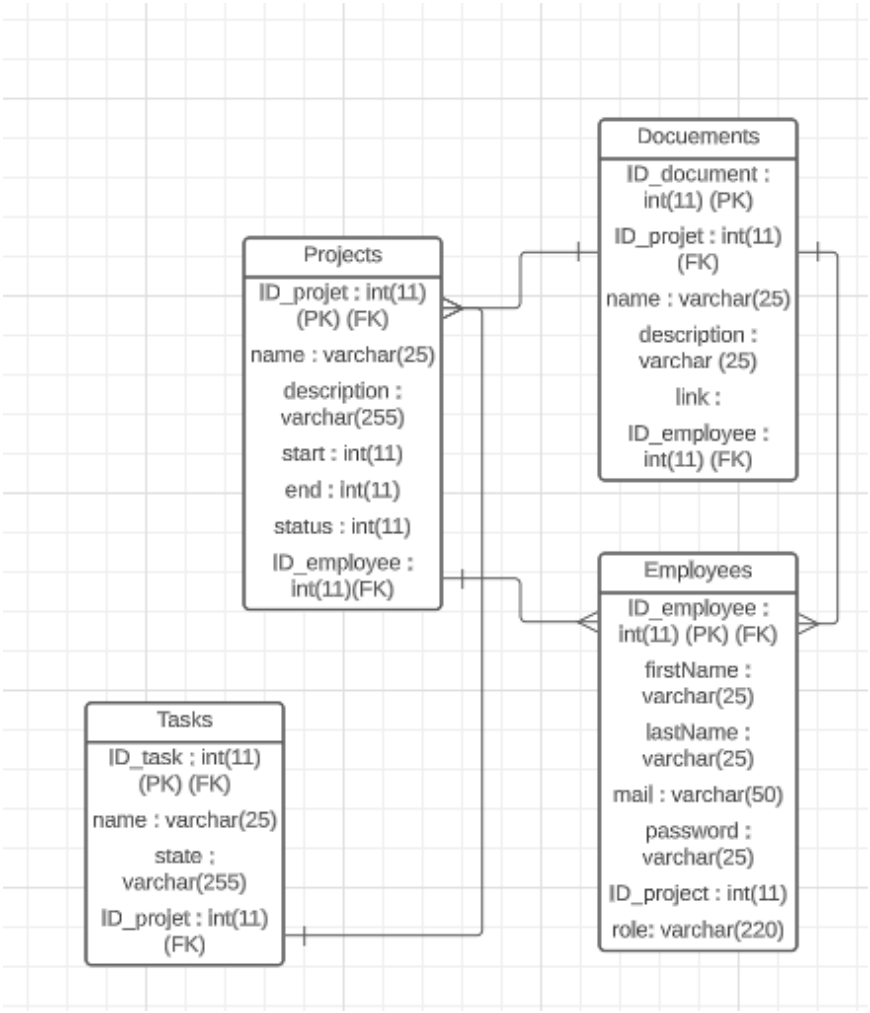


Schéma base de donnée "medgenix"

BACK-END

Base de donnée

```
create databae medgenix;
USE medgenix;
```

```
CREATE TABLE users (
    ID_user INT AUTO_INCREMENT PRIMARY KEY,
    firstName VARCHAR(255),
    lastName VARCHAR(255),
    email VARCHAR(255),
    password VARCHAR(255),
    ID_project INT,
    role VARCHAR(255),
    FOREIGN KEY (ID_project) REFERENCES Projects(ID_project)
);
CREATE TABLE projects (
    ID_project INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    content TEXT,
    start DATETIME,
    end DATETIME,
    ID_user INT,
    status VARCHAR(255),
    FOREIGN KEY (ID_user) REFERENCES Users(ID_user)
);
CREATE TABLE documents (
    ID_document INT AUTO_INCREMENT PRIMARY KEY,
    ID_project INT,
    content TEXT,
    link VARCHAR(255),
    ID_user INT,
    FOREIGN KEY (ID_project) REFERENCES Projects(ID_project),
    FOREIGN KEY (ID_user) REFERENCES Users(ID_user)
);
CREATE TABLE tasks (
    ID_task INT AUTO_INCREMENT PRIMARY KEY,
    ID_project INT,
    content TEXT,
    state VARCHAR(255),
    FOREIGN KEY (ID_project) REFERENCES Projects(ID_project)
);
```

Les tables sont dans l'ensemble reliée, comme mous pouvons le voir avec les Foreign Key. Elle permet d'utiliser les valeurs d'une table pour retrouver les projets en fonction de l'utilisateur.

BACK-END

Initialisation API

Afin de créer l'API, les commandes sont lancées dans un terminal à la racine du projet. Il est important d'avoir nodeJS et npm.
Le projet se lance avec node index.js

```
npm init -y
npm i express body-parser mariadb dotenv cors express-validator
bcrypt jsonwebtoken
npm i -D nodemon
node index.js
```

Dans index.js, on retrouve les modules que l'on veut utiliser : dotenv, body-parser, express, cors.

```
require('dotenv').config();
const bodyParser = require("body-parser");
const express = require("express");
const app = express();
const cors = require("cors");

app.use(cors());
app.use(bodyParser.json());
});
```

Le port est 3003 est utilisé pour cette API avec des les fonction qui vont appliquer les requêtes, les réponses et fonction qui va gérer les erreurs avec next().

```
const port = 3003;
app.use((req, res, next) => {
next();
});
app.listen(3003, () => {
console.log(`app is listening on port ${port}`);
});
```

BACK-END

Connexion base de donnée

Les données d'identifications se trouvent dans un fichier .env afin de sécuriser la connexion. Le module dotenv permet de faire le lien avec les autres fichiers.
La connexion se fait avec la fonction createPool(), elle récupère les identifiants fournies par le module dotenv depuis le .env.
La fonction getConnection(), va se connecter à la base de donnée. Si une erreur se produit lors de la connexion une erreur apparait en console.

```
require('dotenv').config();
const mariadb = require('mariadb');

const pool = mariadb.createPool({
    host: process.env.DB_HOST,
    port: process.env.DB_PORT,
    user: process.env.DB_USER,
password: process.env.DB_PASSWORD,
database: process.env.DB_DATABASE,
    acquireTimeout: 6000000
                });

pool.getConnection()
.then(connection => {
  console.log("Connected to DB!");
    connection.release();
                })
  .catch(error => {
console.error("Error connecting to DB:", error);
                });

module.exports = { pool };
```

Fichier "db.js"

BACK-END

Requête POST

La requête post la même pour l'ensemble des tables, à l'exception des valeurs et du cryptage de mot de posse qui est présent pour l'inscription.

On a deux dossiers : routes et controller.
Route contient les fonctions routes qui vont déterminer le chemin, récupère les valeurs, appelle la fonction qui fait la requête dans controller, puis traite le code retour. Les valeurs sont récupérées avec req.body, le mot de passe est hashé avec bcrypt et sa fonction hash(). Les valeurs sont prises en compte dans la fonction qui va envoyer la requête SQL, createUser().

```
const express = require("express");
const createUser = require('../controller/user');
const router = express.Router();

router.post(
  "/signup",
    [ ],
async (req, res) => {
    try {
const { firstName, lastName, email, password, ID_project,
role } = req.body;
 const hashedPassword = await bcrypt.hash(password, 12);

const userId = await createUser({
    firstName,
    lastName,
    email,
password: hashedPassword,
    ID_project,
    role,
    });
res.status(201).json({ message: "User created!", userId:
userId.toString() });
```

BACK-END

Requête POST

La fonction createUser() envoi une requête pour insérer des données dans la base de donnée avec les valeurs que l'on a récupérées. Pool créer la connexion avec la base de donnée.

```
const { pool } = require('../db');
const createUser = async (user) => {
try {
const { firstName, lastName, email, password, ID_project,
role } = user;
const result = await pool.query('INSERT INTO users
(firstName, lastName, email, password, ID_project, role) VALUES
(?, ?, ?, ?, ?, ?);', [firstName, lastName, email, password,
ID_project, role]);
return result.insertId;
} catch (error) {
throw error;
}
};
```

L'ensemble des requêtes POST, fonctionne de cette manière.

Requête GET By...

La requête pour récupérer les données en fonction d'une valeur. Nous allons voir le login car il s'agit d'un script qui prend en compte le hashage de mot de passe et la clé JWT..
Le login est différent car on vérifie qu'il s'agit de bon mot de passe avec bcrypt et sa fonction compare(). Ensuite, on récupère une clée JWT avec jsonwebtoken. Elle permet de créer des connexions sécurisées en donnant un temps défini de connexion.

Dans routes, on s'occupe de traiter les données.

BACK-END

La requête pour récupérer les données en fonction d'une valeur. Nous allons voir deux exemples, le login et tasks.
Le login est différent car on vérifie qu'il s'agit de bon mot de passe avec bcrypt et sa fonction conpare(). Ensuite, on récupère une clée JWT avec jsonwebtoken. Elle permet de créer des connexions sécurisées en donnant un temps défini de connexion.

Dans routes, on s'occupe de traiter les données.

```
router.post("/login", async (req, res, next) => {
                        try {
        const { email, password } = req.body;
             if (!email || !password) {
  return res.status(400).json({ error: "Invalid login data"
                        });
                           }

     const logUser = await login(req, res, next);

                    if (!logUser) {
   return res.status(401).json({ error: "Login failed" });
                           }

  res.status(200).json({ message: "Logged in!", user: logUser
                        });
                  } catch (error) {
                 console.error(error);
     res.status(500).json({ error: "Login failed" });
                           }
                        });
```

BACK-END

Le fichier dans routes, appel la fonction login et gère les erreurs.

```
router.post("/login", async (req, res, next) => {
try {
const { email, password } = req.body;

if (!email || !password) {
return res.status(400).json({ error: "Invalid login data"
});
}

const logUser = await login(req, res, next);

if (!logUser) {
return res.status(401).json({ error: "Login failed" });
}

res.status(200).json({ message: "Logged in!", user: logUser
});
} catch (error) {
console.error(error);
res.status(500).json({ error: "Login failed" });
}
});
```

Requêtes

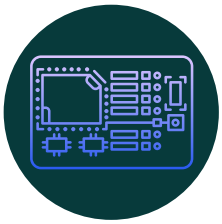
Les requêtes sont testée avec Postman, on y insère l'URL afin de faire une requête POST ou GET en fonction de différents paramètres que l'ont va prendre en compte comme l'id de l'utilisateur.

TESTS DE MEDGENIX

Afin de tester medgenix, voici comment on installe le projet

01 - Récupération du projet

Le projet est séparé en deux dossiers, sur github :
La partie front-end : https://github.com/MarineRcher/MedGenix-front.git
La partie Back-end : https://github.com/MarineRcher/MedGenix-Backend.git
Afin de les récupérer, dans un terminal et dans l'endroit où vous souhaitez l'installer, faire "git clone $cheminGitDonne"



02 - Installation des composants

Dans un terminal, aller dans le dossier front-end et back-end afin d'installer les modules, il faut lancer la commande "npm install".
Le front-end se lance avec la commande "npm run dev", le back-end se lance avec la commande "node index.js".

L'API est sur le port 3000.

03 - Base de donnée

Un fichier .env contient tous les identifiants de connexion à la base de donnée. Or, il ne sera pas déposé sur git mais mis ici.
Le fichier a importé pour la base de donnée se situe dans le répertoire Medgenix-backend.

DB_HOST="localhost"
DB_PORT=3306
DB_USER="root"
DB_PASSWORD="2f82BztZF2R4yi"
DB_DATABASE="medgenix"

JWT_SECRET="2f8tZF2R4yi"

CONCLUSION

Le projet Medgenix permet de mieux s'organiser et de gagner du temps dans les tâches à faire.
Certains problème liés aux modules sont apparus ils sont à jour actuellement.

PERCPECTIVES

Dans le futur, il serait intéressant de rajouter :

- Un chat pour discuter entre membre d'un projet
- Des documents à ajouter en fonction du projet
- Un calendrier afin de se tenir à jour dans les événements.

BIBLIOGRAPHIE

Maquette

dribbhttps://dribbble.com

Code

https://react.dev/learn/typescript
https://reactrouter.com/en/main
https://expressjs.com/
https://mariadb.org/documentation/