

SHARE-EVENT

Concepteur développeur D'applications



Raphaël Diop

SOMMAIRES

1. Introduction
2. Cible de l'application
 - 2.1 Profils des utilisateurs
 - 2.2 Besoin des utilisateurs
3. Conception de l'application
 - 3.1 Trello
 - 3.2 Figma
 - 3.3 LucidChart
4. Développement de l'application
 - 4.1 Front-End : React Native & Expo
 - 4.2 Back-End: NestJS et TypeORM
 - 4.3 Relation Front-end Back-End
5. Firebase
6. Démo de l'application
7. Sécurité
 - 7.1 JWT et Expo Secure Store
 - 7.2 Sanitize-html
 - 7.3 bcrypt
 - 7.4 jsonwebtoken
 - 7.5 Middleware
8. Conclusion

1- Introduction:

La photographie occupe une place centrale dans notre vie, et c'est particulièrement vrai lors d'événements. Toutefois, une question persiste : comment partager efficacement les précieux clichés pris par les participants avec l'ensemble des invités, y compris les organisateurs ? Notre application apporte la solution en offrant une plateforme de partage en temps réel dédiée aux événements.

Que ce soit pour un mariage, une fête d'anniversaire ou un événement professionnel, chacun prend des photos avec son smartphone. Avec notre application, ces clichés ne restent plus isolés dans chaque appareil, mais sont rassemblés, partagés et accessibles à tous. De plus, les organisateurs ont la possibilité d'utiliser ces images pour promouvoir leurs événements sur les réseaux sociaux.

2 - Cible de l'application

2.1 Profil des utilisateurs

Notre application est conçue pour tous ceux qui souhaitent créer, organiser et animer des événements. Qu'il s'agisse de célébrations familiales, d'événements professionnels ou d'activités entre amis, l'application s'avère un outil précieux pour le partage de souvenirs.

2.2 Besoins des utilisateurs

L'application répond aux défis que rencontrent les organisateurs d'événements en leur offrant une solution pour rassembler toutes les photos prises lors de leur événement. Par ailleurs, elle permet aux participants de partager leurs moments précieux et de garder une trace tangible de l'événement.

En résumé, l'application offre un moyen efficace et convivial de capturer, de partager et d'interagir avec les photos prises lors d'un événement. Elle apporte une réponse concrète aux besoins des organisateurs et des participants, tout en favorisant l'engagement et la promotion de l'événement sur les réseaux sociaux.

3 - Conception de l'application

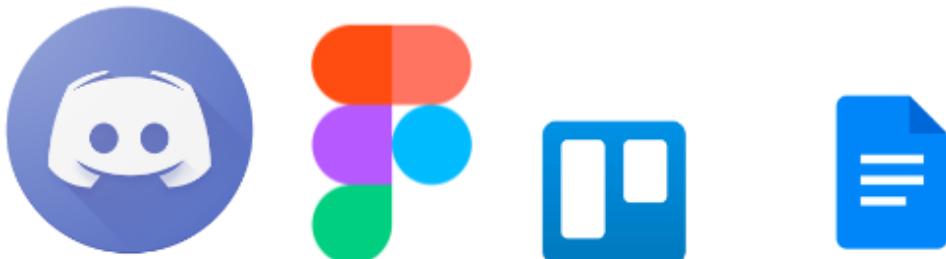
La conception de notre application a été guidée par un processus itératif et collaboratif. Grâce à des outils comme Figma et Lucidchart, nous avons pu élaborer des wireframes et des diagrammes d'architecture précis, contribuant à la clarté de l'interface utilisateur et à l'organisation technique.

Nous avons adopté la méthodologie Agile pour gérer le projet, favorisant une adaptation rapide aux changements et une amélioration continue. Des sprints réguliers ont permis à chaque membre de l'équipe de partager ses avancées, ses difficultés et ses idées.

De plus, le rôle de Scrum Master a été rotatif, ce qui a permis à tous de s'impliquer activement dans la gestion du projet. Ce rôle-clé s'est traduit par la coordination des réunions, la prise de notes et la facilitation de la communication.

Pour optimiser notre organisation, nous avons utilisé Trello pour la gestion des tâches, permettant à chaque membre de l'équipe de visualiser l'avancement du projet et ses responsabilités. Discord a été notre plateforme de choix pour la communication en télétravail, assurant une interaction fluide et efficace, quel que soit le lieu de travail de chacun. Enfin, Google Docs a été utilisé pour consigner les comptes-rendus de réunions et garder une trace écrite de notre processus de développement.

Cette combinaison d'outils et d'approches a permis de créer une application qui répond de manière précise et efficace aux besoins de nos utilisateurs.

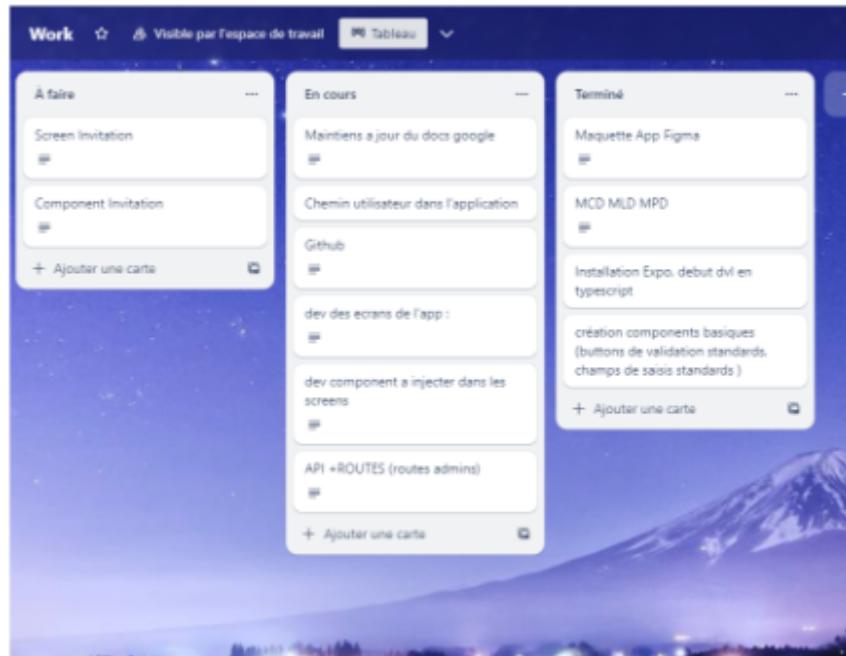


3.1 - Gestion de Projet avec Trello

Pour assurer un suivi efficace de nos tâches et une gestion ordonnée de notre flux de travail, nous avons utilisé Trello, un outil de gestion de projet bien établi. Grâce à sa structure basée sur les tableaux, les listes et les cartes, Trello nous a permis de visualiser facilement l'avancement de notre projet et de répartir les tâches de manière équitable au sein de notre équipe.

Chaque carte Trello représentait une tâche spécifique et était assignée à un ou plusieurs membres de l'équipe. Chaque carte pouvait être déplacée entre les listes, généralement organisées en 'À faire', 'En cours' et 'Terminé', ce qui nous donnait une vue d'ensemble claire du statut de chaque tâche et de l'avancement global du projet.

En utilisant Trello, nous avons pu maintenir une bonne organisation, un suivi des tâches et une communication claire au sein de l'équipe, ce qui a été essentiel pour la réalisation réussie de notre application."



3.2 - Conception avec Figma

Figma, une plateforme de conception d'interface utilisateur basée sur le cloud, a été notre outil de préférence pour la réalisation de notre application pour les raisons suivantes :

- Collaboration en temps réel : Figma facilite le travail collaboratif en temps réel sur un même design, favorisant une meilleure synergie et une communication plus fluide entre les membres de l'équipe.
- Accessibilité : En tant qu'outil basé sur le cloud, Figma peut être utilisé depuis n'importe quel navigateur web, ce qui a permis à chaque membre de l'équipe de travailler sur la conception de l'application, peu importe le lieu ou l'appareil utilisé.
- Fonctionnalités de conception avancées : Figma offre une palette d'outils de conception, nous permettant de créer des wireframes, de développer des composants réutilisables et de définir des styles partagés.
- Prototypage aisement : Figma permet de créer des prototypes interactifs pour une exploration et une validation plus efficaces des concepts de conception.

Notre approche de conception a commencé par le développement du wireframe, une maquette basse-fidélité qui établit l'agencement général de l'interface utilisateur. Par la suite, nous avons élaboré des maquettes haute-fidélité. Ces designs détaillés respectent la charte graphique de l'application et nous ont aidé à valider les choix de conception tout en recueillant des commentaires sur l'apparence et l'ergonomie de l'interface utilisateur.

Détails de la charte graphique :

- Couleur principale : #97C17E 
- Couleur secondaire : #E6C821 
- Couleur de fond : #FCFCFC
- Police pour les titres et entête : CINZEL
- Police pour le reste du texte : open sans

Vous trouverez ci-joint des captures d'écran illustrant le processus de conception, du wireframe initial jusqu'aux maquettes haute-fidélité finales.

Exemple Wireframe :



Exemple maquettes haute-fidélité :

HF-PROFIL

HF-CREATE-EVENT

HF-EVENT

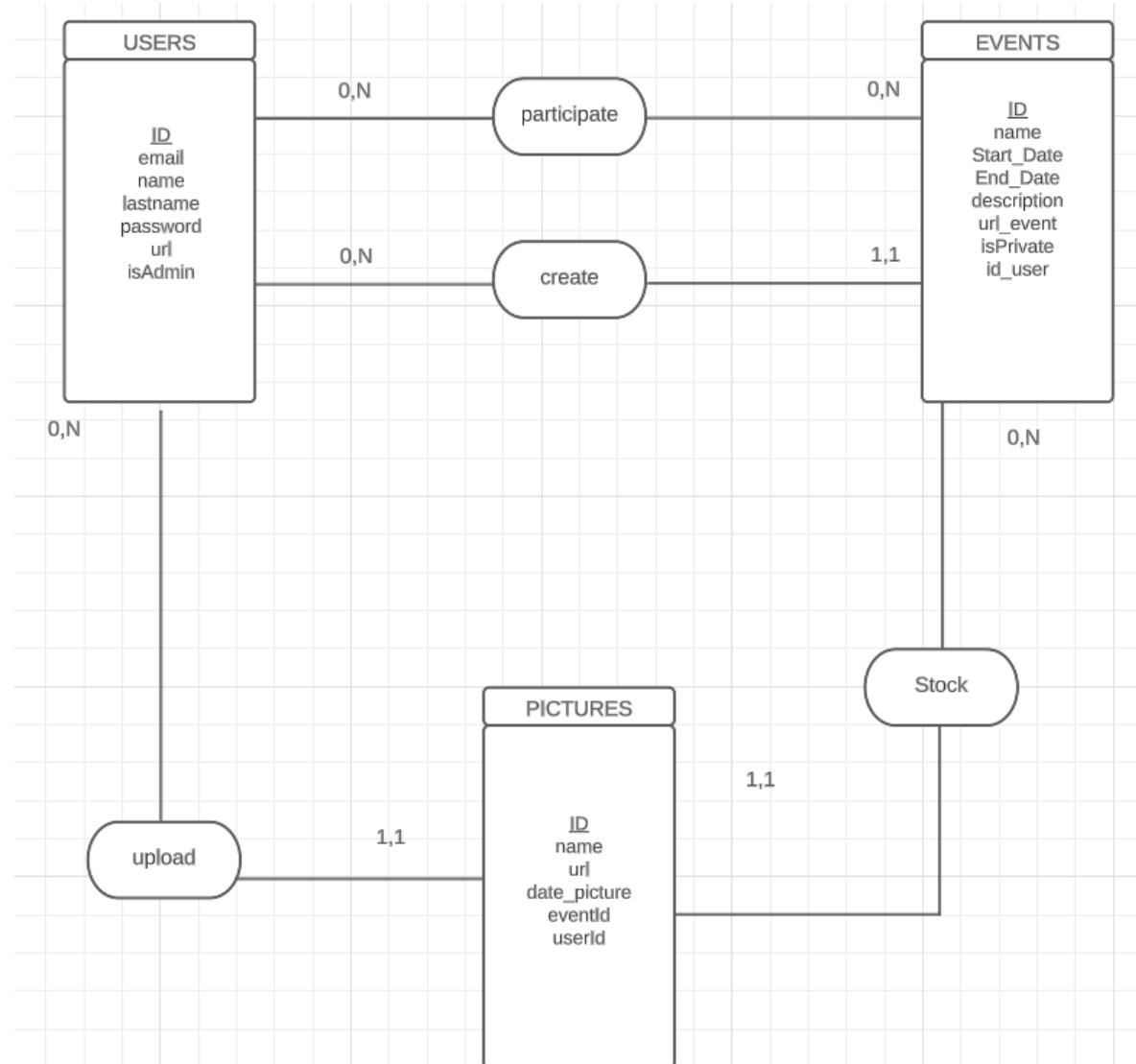
3.3 LucidChart

LucidChart est une plateforme en ligne pour la création de diagrammes et de modèles, offrant de multiples avantages à ses utilisateurs :

- Facilité d'utilisation : LucidChart se distingue par son interface utilisateur intuitive, permettant même aux novices de produire des diagrammes professionnels en quelques clics.
- Collaboration : Grâce à la capacité de collaboration en temps réel de LucidChart, travailler en synergie avec des collègues, des clients ou des partenaires externes est grandement facilité.
- Fonctionnalités avancées : Avec des options telles que l'intégration de données, l'automatisation de tâches et la personnalisation avancée, LucidChart s'adapte également aux besoins des utilisateurs plus expérimentés.
- Accessibilité : Étant une solution en ligne, LucidChart peut être utilisé sur n'importe quel appareil, offrant une accessibilité universelle, à tout moment et en tout lieu.
- Variété de modèles : LucidChart fournit une large gamme de modèles pour différents types de diagrammes, tels que les organigrammes, les cartes mentales, les diagrammes de flux, et les maquettes d'interfaces utilisateur, entre autres.

3.3 LucidChart

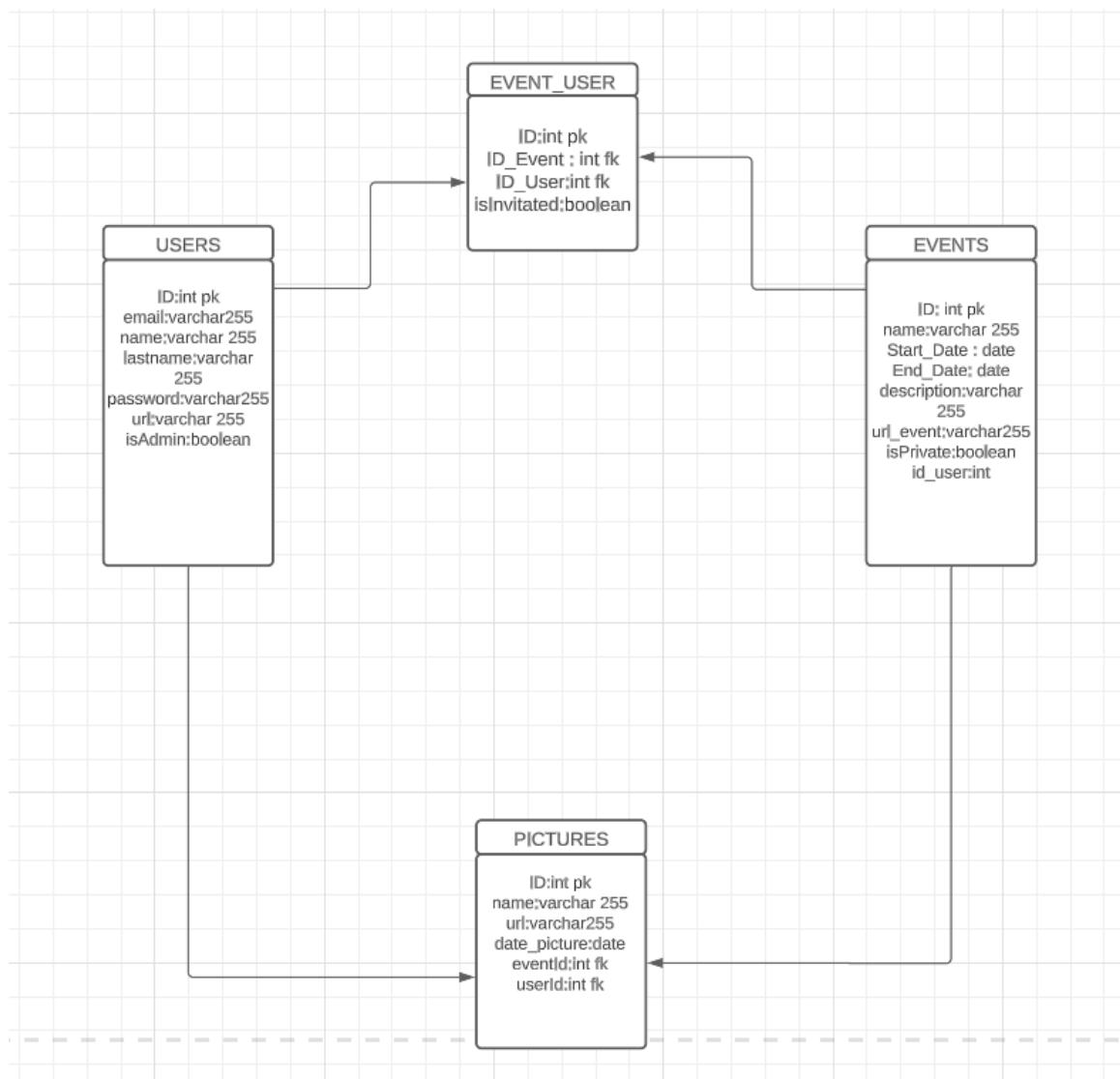
Dans la phase initiale de conception, nous avons créé le Modèle Conceptuel de Données (MCD) afin d'identifier les entités, leurs attributs, et les relations entre elles. Voici un aperçu du MCD :



3.3 LucidChart

Une fois le MCD validé par l'équipe, nous avons procédé à l'élaboration du Modèle Logique de Données (MLD). Le MLD sert à transformer le MCD en une représentation logique qui peut être mise en œuvre, incluant des entités, des relations, des clés primaires et des clés étrangères.

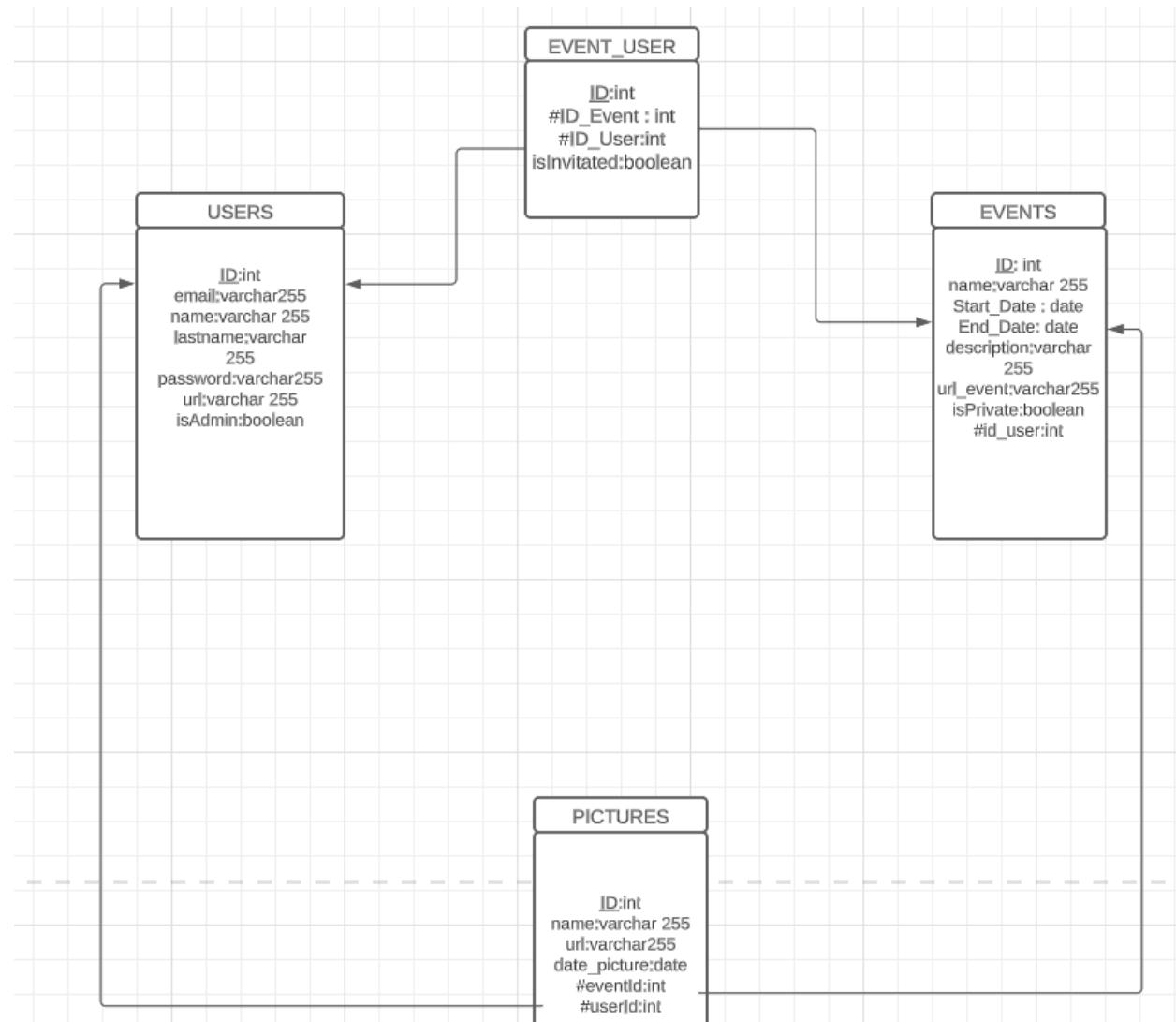
Voici à quoi ressemble notre MLD :



3.3 LucidChart

Enfin, nous avons transformé le MLD en un Modèle Physique de Données (MPD). Le MPD permet de convertir les entités et les relations du MLD en tables et en contraintes adaptées à la base de données réelle.

Voici notre MPD :



4.Développement de l'application

4.1 Front-End: React Native avec Expo

L'application a été construite en utilisant React Native avec Expo, un ensemble d'outils qui permet aux développeurs de construire et de déployer rapidement des applications natives pour iOS et Android à partir d'un seul codebase JavaScript.

Une partie importante du développement de notre application a été l'organisation de la navigation entre les différents écrans de l'application. Nous avons utilisé la bibliothèque '@react-navigation/stack' pour créer des piles de navigation, qui permettent aux utilisateurs de naviguer entre les écrans en les empilant les uns sur les autres.

```
function ProfilStack() {
  return (
    <Stack.Navigator initialRouteName="Profile">
      <Stack.Screen name="Profile" component={ProfilScreen} options={{ headerShown: false }} />
      <Stack.Screen name="MyEvents" component={MyEventsScreen} />
      <Stack.Screen name="MyProfil" component={UpdateProfilScreen} />
      <Stack.Screen name='ManageEvent' component={AdminEventScreen} />
    </Stack.Navigator>
  );
}
```

4.1 Front-End: React Native avec Expo

Ensuite, nous avons utilisé `createBottomTabNavigator` pour créer une barre de navigation inférieure avec des icônes pour chaque pile de navigation. Ces icônes permettent aux utilisateurs de passer facilement d'une pile de navigation à une autre.

```
Function TabNavigator() {
  return (
    <Tab.Navigator
      initialRouteName="Home"
      screenOptions={({ route }) => ({
        headerShown: false,
        tabBarIcon: ({ focused, color, size }) => {
          let iconName;
          if (route.name === 'Home') {
            iconName = (
              <Entypo name="home" size={24} color={focused ? '#CBA85F' : 'white'} />
            );
          } else if (route.name === 'Profil') {
            iconName = (
              <AntDesign name="user" size={24} color={focused ? '#CBA85F' : 'white'} />
            );
          } else if (route.name === 'PublicEvents') {
            iconName = (
              <Entypo name="list" size={24} color={focused ? '#CBA85F' : 'white'} />
            );
          } else if (route.name === 'CreateEvent') {
            iconName = (
              <Ionicons name="create" size={24} color={focused ? '#CBA85F' : 'white'} />
            );
          }
          return iconName;
        },
        tabBarActiveTintColor: '#CBA85F',
        tabBarInactiveTintColor: 'white',
        tabBarStyle: [
          backgroundColor: '#98A68F',
        ],
      })}
    >
      <Tab.Screen name="Home" component={EventsStack} />
      <Tab.Screen name="Profil" component={ProfilStack} />
      <Tab.Screen name="PublicEvents" component={PublicEventsScreen} />
      /* <Tab.Screen name="CreateEvent" component={CreateEventScreen} /> */
    </Tab.Navigator>
```

4.1 Front-End: React Native avec Expo

Enfin, dans le composant App lui-même, nous avons utilisé le hook useState de React pour garder une trace de si l'utilisateur est connecté ou non, et le hook useEffect pour vérifier l'état de connexion de l'utilisateur lors du chargement de l'application. En fonction de l'état de connexion de l'utilisateur, nous affichons soit le TabNavigator (si l'utilisateur est connecté), soit le AuthStack (si l'utilisateur n'est pas connecté).

```
export default function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  useEffect(() => {
    const checkIfLoggedIn = async () => {
      const token = await SecureStore.getItemAsync('acess_token');

      if (token) {
        setIsLoggedIn(true);
        console.log(token);
      } else {
        setIsLoggedIn(false);
      }
    };
    checkIfLoggedIn();
  }, []);

  return (
    <NavigationContainer>
      {isLoggedIn ? (
        <TabNavigator />
      ) : (
        <Stack.Navigator>
          <Stack.Screen
            name="AuthStack"
            component={AuthStack}
            options={{ headerShown: false }}
          />
        </Stack.Navigator>
      )}
    </NavigationContainer>
  );
}
```

4.1 Front-End: React Native avec Expo

Cette approche nous a permis de construire une application avec une interface utilisateur riche et interactive, offrant une expérience utilisateur fluide et intuitive. Le fait d'avoir un seul codebase JavaScript a également simplifié le processus de développement et de maintenance, car nous n'avons pas eu à écrire de code spécifique pour iOS ou Android.

4.2 Backend : NestJS avec TypeORM

NestJS est un framework de développement d'applications backend en Node.js qui utilise TypeScript comme langage de base. NestJS fournit une architecture modulaire qui nous permet de structurer notre code de manière efficace et organisée.

Dans notre application, nous avons utilisé le module TypeORM de NestJS pour interagir avec notre base de données MySQL. Nous avons également défini plusieurs modules, chacun étant responsable de sa propre fonctionnalité spécifique. Par exemple, le UsersModule s'occupe de toutes les fonctionnalités liées aux utilisateurs, tandis que le PicturesModule gère tout ce qui concerne les images. Le code ci-dessous montre notre AppModule, qui importe tous ces modules.

```
import { TypeOrmModule } from '@nestjs/typeorm';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UsersModule } from './users/users.module';
import { User } from './users/model/entities/users.entity';
import { PicturesModule } from './pictures/pictures.module';
import { Picture } from './pictures/model/entities/pictures.entity';
import { EventsModule } from './events/events.module';
import { AuthModule } from './auth/auth.module';
import { AdminModule } from './admin/admin.module';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'api_share_event',
      synchronize: true,
      logging: true,
      entities: [User,Picture,Event],
      subscribers: [],
      migrations: [],
      autoLoadEntities: true,
    }),
    UsersModule,PicturesModule,EventsModule,AuthModule,AdminModule,
    controllers: [AppController],
    providers: [AppService],
  ]
}

export class AppModule {}
```

Dans cet exemple, notre AppModule importe plusieurs autres modules, chacun étant responsable de sa propre fonctionnalité spécifique

4.2 Backend : NestJS avec TypeORM

Nous avons utilisé le pattern repository de TypeORM pour interagir avec notre base de données. Le code ci-dessous montre comment nous utilisons le repository dans notre UsersService pour effectuer différentes opérations sur les utilisateurs.

```
import { Injectable } from '@nestjs/common';
import { Repository } from 'typeorm';
import { User } from './model/entities/users.entity';
import { CreateUserDto } from './dto/create-user.dto';
import { UserInterface } from './model/users.interface';
import { UpdateUserDto } from './dto/update-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import * as bcrypt from 'bcrypt';

@Injectable()
export class UsersService {
    constructor(
        @InjectRepository(User) private readonly usersRepository:
        Repository<User>,
    ) {}

    async create(user: CreateUserDto) {
        const passwordHash = await bcrypt.hash(user.password, 10);
        const getUserbefore = await
        this.usersRepository.findAndCountBy({isAdmin:true});

        if(getUserbefore[1] >=1){
            return this.usersRepository.save(
            {
                email:user.email,
                name:user.name,
                url:user.url,
                lastname:user.lastname,
                password: passwordHash,
                isAdmin:false,
            }
        )}
        else{
            return this.usersRepository.save(
            {
                email:user.email,
```

```

        name:user.name,
        url:user.url,
        lastname:user.lastname,
        password: passwordHash,
        isAdmin:true,
    }
)
);
}

findAll(): Promise<User[]> {
    return this.usersRepository.find();
}

findOneByEmail(email:string){
    return this.usersRepository.findOneBy({email});
}
findOne(id: number) {
    return this.usersRepository.findOneBy({ id });
}

async updateUser(id: number, user: UpdateUserDto): Promise<any> {
    var passwordHash= user.password;
    if(user.password){
        passwordHash = await bcrypt.hash(user.password,10);
    }
    return this.usersRepository.update(id,
    {
        email:user.email,
        name:user.name,
        url:user.url,
        lastname:user.lastname,
        password:passwordHash,
        isAdmin:user.isAdmin,
    });
}

async remove(id: number) {
    const getUserbefore= await
this.usersRepository.findAndCountBy({isAdmin:true});
    if(getUserbefore[1]==1){
        return"your application need an administrator";
    }else{
        return this.usersRepository.delete({ id });
    }
}

```

4.2 Backend : NestJS avec TypeORM

Nous avons également défini plusieurs contrôleurs pour exposer différentes routes de l'API. Par exemple, notre UsersController expose des routes pour enregistrer un nouvel utilisateur, récupérer tous les utilisateurs, trouver un utilisateur par son email, récupérer un utilisateur spécifique par son ID, mettre à jour les informations d'un utilisateur et supprimer un utilisateur. Vous pouvez voir un extrait du UsersController ci-dessus.

```
import { Controller, Post, Body, Logger } from '@nestjs/common';
import { GetUserDto } from 'src/users/dto/get_user.dto';
import { AuthService } from './auth.service';

@Controller('auth')
export class AuthController {
    constructor(private authService: AuthService) {}

    @Post('login')
    async login(@Body() GetUserDto: GetUserDto) {
        return this.authService.login(GetUserDto);
    }

    @Post('refresh')
    async refresh(@Body() refresh: string) {
        return this.authService.refresh(refresh);
    }
}
```

Pour l'authentification des utilisateurs, nous avons utilisé le module JWT de NestJS. Notre AuthService génère des jetons d'accès et de rafraîchissement lorsqu'un utilisateur se connecte avec succès. Le jeton d'accès est utilisé pour authentifier l'utilisateur lors des requêtes suivantes, tandis que le jeton de rafraîchissement est utilisé pour générer un nouveau jeton d'accès lorsque celui-ci est expiré.

```
import { Injectable, NotFoundException, UnauthorizedException } from
'@nestjs/common';
import { UsersService } from '../users/users.service';
import { GetUserDto } from '../users/dto/get_user.dto';
import { JwtService } from '@nestjs/jwt';
import * as bcrypt from 'bcrypt';

@Injectable()
export class AuthService {
    constructor(private readonly UsersService: UsersService, private readonly jwtService: JwtService) { }

    async validateUser(email: string, password: string): Promise<any> {
        const user = await this.UsersService.findOneByEmail(email);
        if (user) {
            const { password, ...result } = user;
            return result;
        }
        return null;
    }

    async login(GetUserDto: GetUserDto) {
        const foundUser = await this.UsersService.findOneByEmail(GetUserDto.email);

        if (!foundUser) {
            return "l'email est incorrect";
        }
        const passwordMatch = await
bcrypt.compare(GetUserDto.password, foundUser.password);

        if (!passwordMatch) {
            return 'l email ou le mot de passe est incorrect'
        }

        const payload = {
            id: foundUser.id,
            url: foundUser.url,
            email: foundUser.email,
            name: foundUser.name,
            lastname: foundUser.lastname,
            isAdmin: foundUser.isAdmin,
        }
        const token = this.jwtService.sign(payload);
        return { token, user: foundUser };
    }
}
```

```

    }
    const refresh_token = this.jwtService.sign(payload, { expiresIn: '30d' });
    return {
      access_token: this.jwtService.sign(payload),
      refresh_token,
    }
  }
}

async refresh(refreshToken:string):Promise<any> {

  const decoded = await this.jwtService.verify(refreshToken);
  const user = await this.UsersService.findOne(decoded.id);

  if (!user) {
    throw new UnauthorizedException('Invalid token');
  }

  try {
    const payload = {
      id:user.id,
      url:user.url,
      email:user.email,
      name:user.name,
      lastname:user.lastname,
      isAdmin:user.isAdmin,
    }
    const accessToken = this.jwtService.sign(payload);
    return {
      access_token: accessToken,
    };
  } catch (err) {
    throw new UnauthorizedException('Invalid token');
  }
}
}

```

En somme, notre application backend utilise efficacement NestJS et TypeORM pour structurer le code et interagir avec la base de données. Chaque partie de l'application est responsable d'une tâche spécifique, ce qui facilite la maintenance et l'évolution du code à l'avenir.

4.3 Relation Back-End et Front-End

La communication entre notre front-end et notre back-end est établie à travers une API REST que nous avons développée sur notre serveur NestJS. Cette API fournit un ensemble d'endpoints pour différentes opérations liées aux utilisateurs, aux images et à d'autres fonctionnalités de l'application.

Fetch et Post Data : Notre application React Native utilise la bibliothèque Axios pour envoyer des requêtes HTTP à notre API. Par exemple, lors de l'authentification, nous utilisons la méthode POST pour envoyer des informations d'authentification à notre endpoint d'authentification.

Authentification : Pour sécuriser notre API, nous utilisons des tokens JWT. Lorsqu'un utilisateur se connecte, le back-end génère un token qui est renvoyé au front-end. Ce token est ensuite stocké localement et inclus dans l'en-tête Authorization de chaque requête Axios. Ce processus permet au serveur de vérifier l'identité de l'utilisateur à chaque requête.

```
try {
  const response = await axios.post(`http://10.10.7.33:3000/auth/login`, { email, password });
  await SecureStore.setItemAsync('acess_token', response.data.acess_token);
```

Gestion des erreurs : Axios nous permet également de gérer facilement les erreurs. Si une requête échoue, Axios rejette la promesse avec une erreur que nous pouvons attraper et gérer. Ceci est particulièrement utile pour informer les utilisateurs lorsque quelque chose ne va pas. Par exemple, lors de la connexion, si le format de l'e-mail est invalide ou si l'e-mail ou le mot de passe sont incorrects, nous définissons une erreur et arrêtons le chargement.

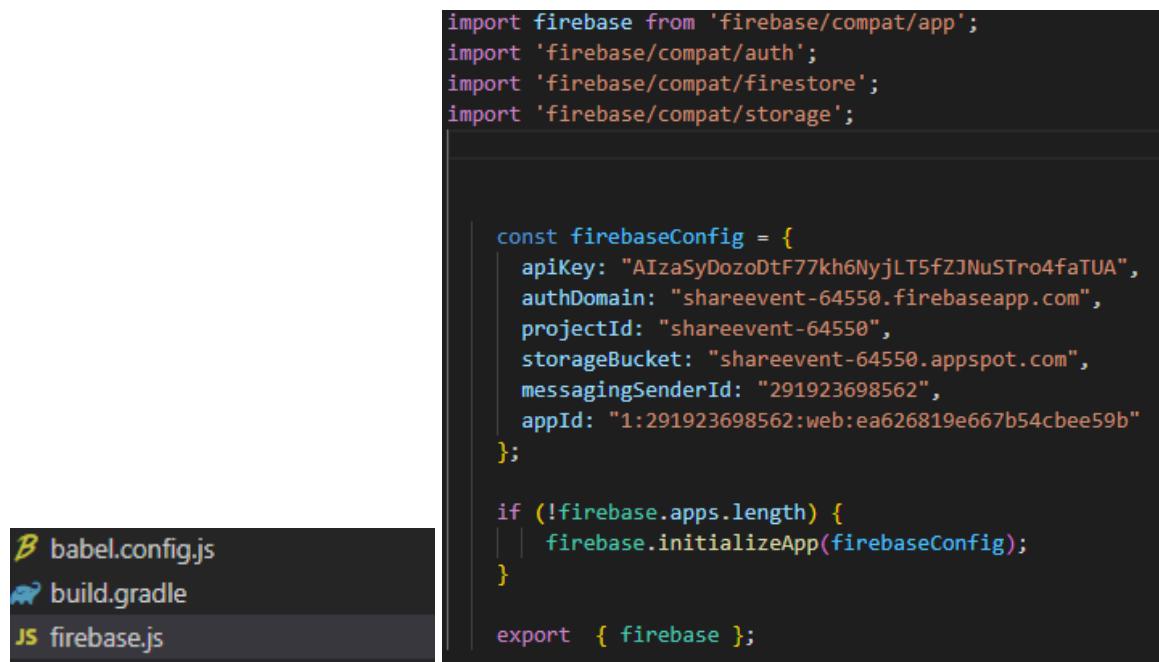
```
if (!validateEmail(email)) {
    setError({ message: 'Invalid email format', statusCode: 0 });
    setIsLoading(false);
    return;
}
try {
    // axios post request
} catch (e) {
    setError({ message: 'Invalid email or password', statusCode: 0 });
    setIsLoading(false);
}
```

Ces différentes interactions entre le front-end et le back-end sont cruciales pour le fonctionnement de notre application. Elles permettent un flux d'information fluide et sécurisé entre les deux, ce qui assure une expérience utilisateur optimale.

5. Firebase

Nous utilisons Firebase Storage, un service de stockage d'objets basé sur le cloud fourni par Firebase, pour stocker les images sélectionnées par les utilisateurs. Voici comment nous intégrons Firebase Storage dans notre application React Native.

Tout d'abord, nous utilisons le package expo-file-system pour gérer les opérations de fichier. Nous avons également configuré Firebase dans notre application à l'aide du module firebase fourni par Firebase.



```
import firebase from 'firebase/compat/app';
import 'firebase/compat/auth';
import 'firebase/compat/firestore';
import 'firebase/compat/storage';

const firebaseConfig = {
  apiKey: "AIzaSyDozoDtF77kh6NyjLT5fZJNuSTro4faTUA",
  authDomain: "shareevent-64550.firebaseio.com",
  projectId: "shareevent-64550",
  storageBucket: "shareevent-64550.appspot.com",
  messagingSenderId: "291923698562",
  appId: "1:291923698562:web:ea626819e667b54cbee59b"
};

if (!firebase.apps.length) {
  firebase.initializeApp(firebaseConfig);
}

export { firebase };
```

The code editor shows the following file structure:

- B** babel.config.js
- M** build.gradle
- JS** firebase.js

5. Firebase

Lorsque l'utilisateur sélectionne une image, nous utilisons la fonction `saveImage2` pour télécharger l'image dans Firebase Storage et enregistrer son URL dans notre base de données. Voici comment nous procédons :

```
const saveImage2 = async (uri: string, eventId: number | null) => {
  try {
    const response = await fetch(imageUri);
    const blob = await response.blob();
    const filename = imageUri.substring(imageUri.lastIndexOf("/") + 1);
    var ref = firebase.storage().ref().child(filename).put(blob);

    try {
      ref.then(async (snapshot) => {
        const url = await snapshot.ref.getDownloadURL();
        console.log(url);
        console.log(userId);

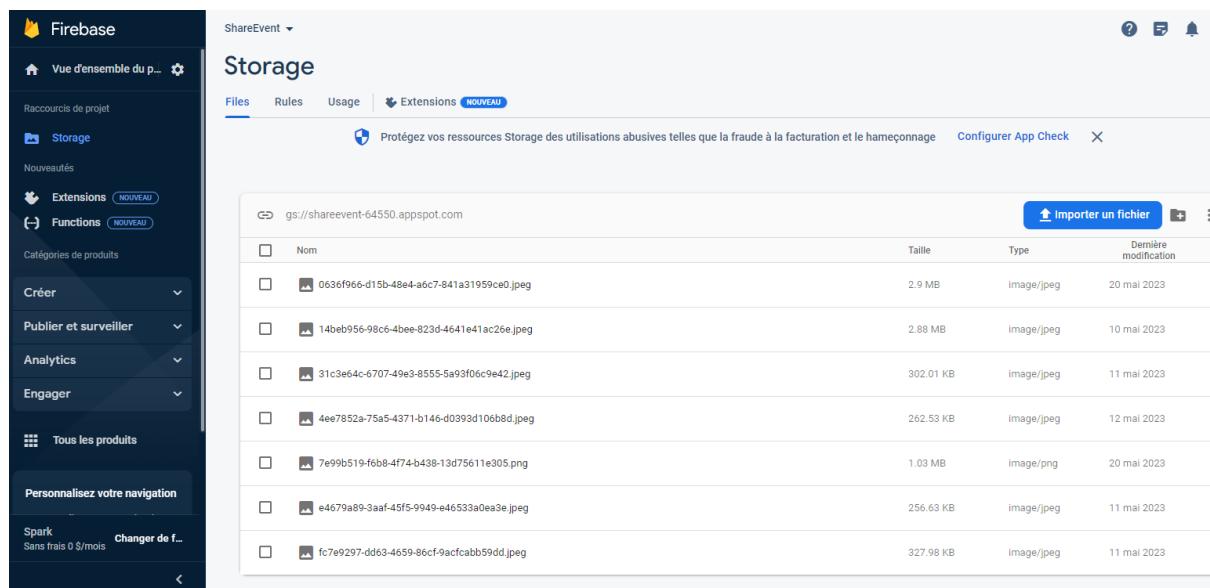
        // Conditionally update the request data based on the uploadPurpose prop
        const requestData =
          uploadPurpose === "event"
            ? {
                name: filename,
                url: url,
                userId: userId,
                eventId: eventId,
              }
            : {
                name: filename,
                url: url,
                userId: userId,
              };
      });
    }

    const response = await axios.post(
      `http://192.168.1.27:3000/pictures/register`,
      requestData
    );
    console.log("Response from server:", response); // Log the server response
    if (onImageUploaded) {
      onImageUploaded(url);
    }
  }
}
```

5. Firebase

Dans cette fonction, nous utilisons fetch pour récupérer le blob de l'image à partir de son URI, puis nous utilisons firebase.storage().ref().child(filename).put(blob) pour télécharger le blob dans Firebase Storage avec le nom de fichier approprié. Une fois le téléchargement terminé, nous utilisons `snapshot.ref.getDownloadURL()` pour obtenir l'URL de téléchargement de l'image.

Ensuite, nous conditionnons les données de la demande en fonction de la valeur de uploadPurpose. Par exemple, si uploadPurpose est "event", nous incluons également l'ID de l'événement associé à l'image. Ensuite, nous envoyons une requête POST à notre endpoint backend avec les données de la demande, y compris le nom de fichier et l'URL de l'image

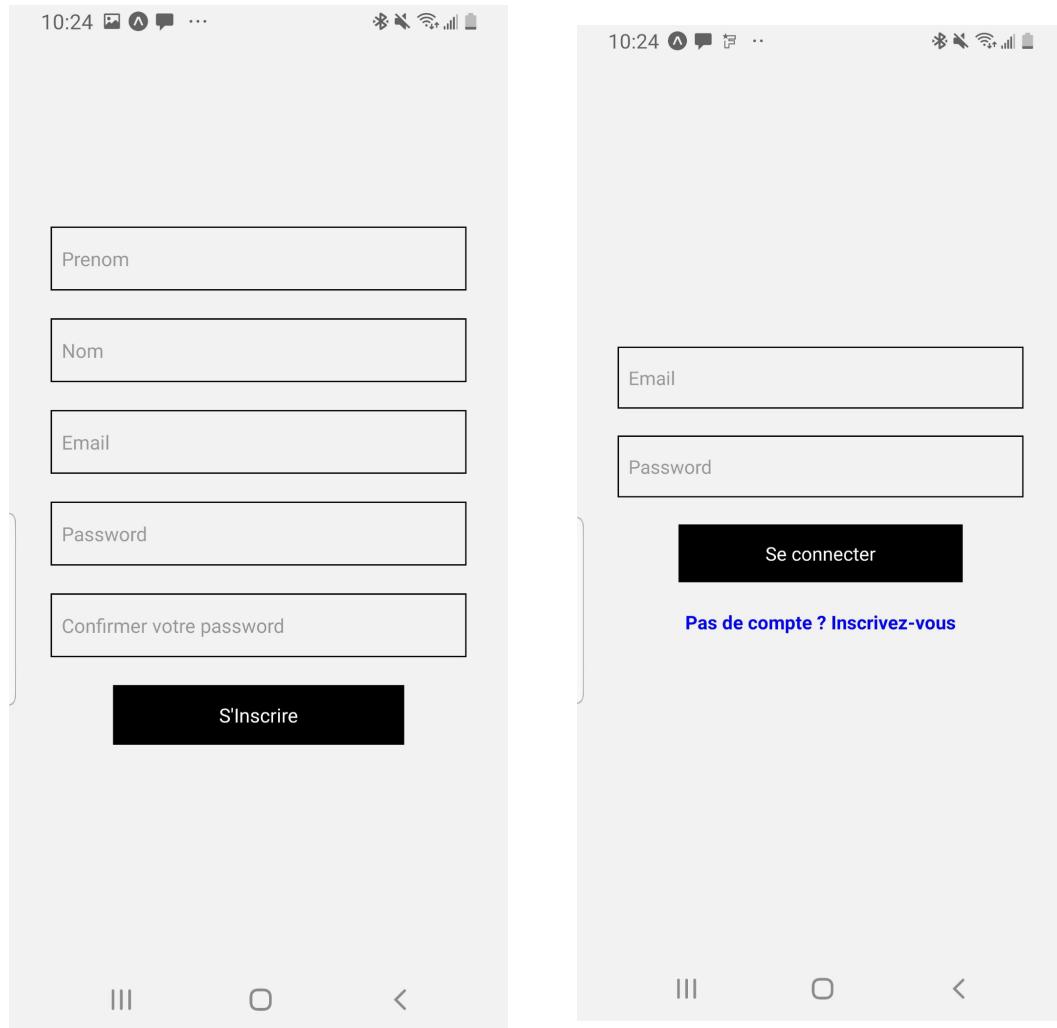


The screenshot shows the Firebase Storage interface. On the left, there's a sidebar with project navigation options like Vue d'ensemble du p..., Storage, Extensions, Functions, and Analytics. The main area is titled 'Storage' and shows a table of uploaded files:

Nom	Taille	Type	Dernière modification
0636f966-d15b-48e4-a6c7-841a31959ce0.jpeg	2.9 MB	image/jpeg	20 mai 2023
14beb956-98c6-4bee-823d-4641e41ac26e.jpeg	2.88 MB	image/jpeg	10 mai 2023
31c3e64c-6707-49e3-8555-5a93f06c9e42.jpeg	302.01 KB	image/jpeg	11 mai 2023
4ee7852a-75a5-4371-b146-d0393d106b8d.jpeg	262.53 KB	image/jpeg	12 mai 2023
7e999b519-f6b8-4f74-b438-13d75611e305.png	1.03 MB	image/png	20 mai 2023
e4679a89-3aa9-45f5-9949-e46533a0ea3e.jpeg	256.63 KB	image/jpeg	11 mai 2023
fc7e9297-dd63-4659-86cf-9acfabb59dd.jpeg	327.98 KB	image/jpeg	11 mai 2023

6.Demo Application

Voici en image un parcours utilisateur de la connexion ou inscription



6.Demo Application

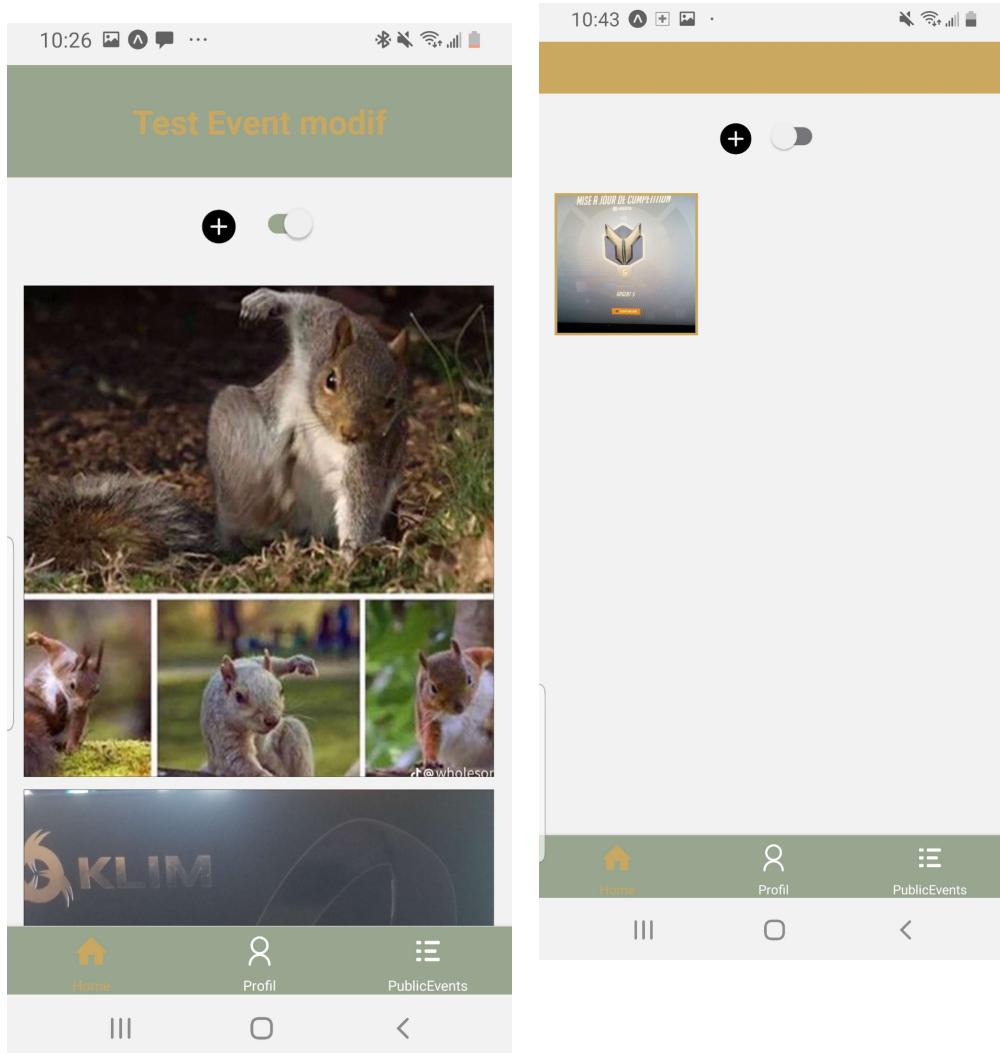
Une fois inscrit/connecté l'utilisateur sera redirigé vers la HomePage de l'application



L'utilisateur aurait en première proposition les events qu'il a créer navigant à swipant de gauche à droite et en dessous les events auxquels il participe.

En cliquant sur l'event voulu la navigation mène sur le mur de l'event en question .

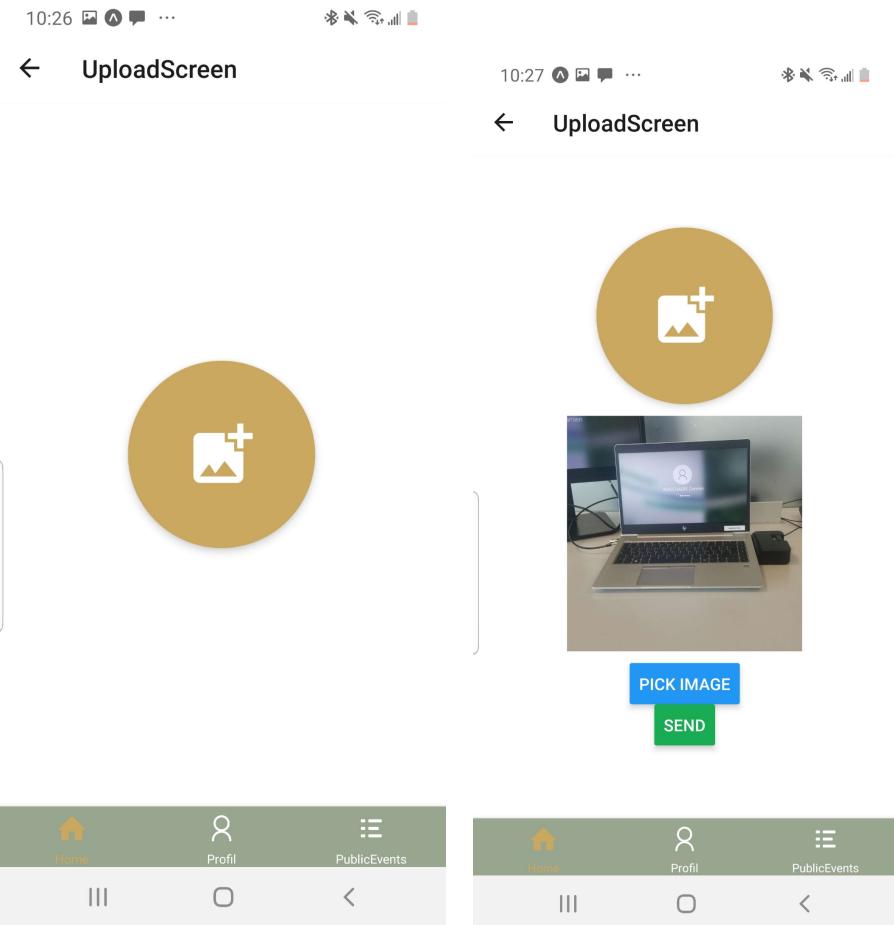
6.Demo Application



Ici la page de l'event sélectionné depuis la HomePage, deux vue sont possible pour l'utilisateur une en mode miniature et l'autre en taille adapté à l'écran qui est scrollable pour faire défiler le mur .

6.Demo Application

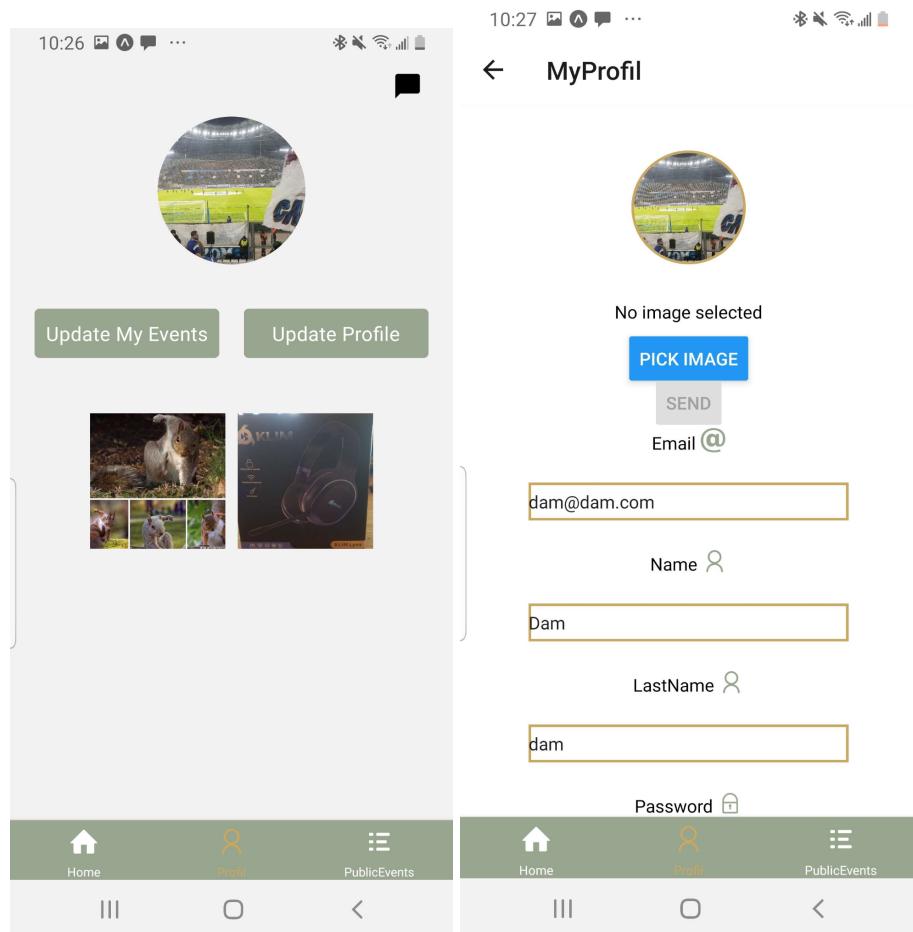
L'utilisateur sur ce mur peut ajouter des Photos si il le souhaite en cliquant sur le bouton + (add) qui le mènera à l'écran de upload Image



Sur cet écran l'utilisateur peut ainsi poster une image sur l'event en la sélectionnant depuis sa bibliothèque.

6.Demo Application

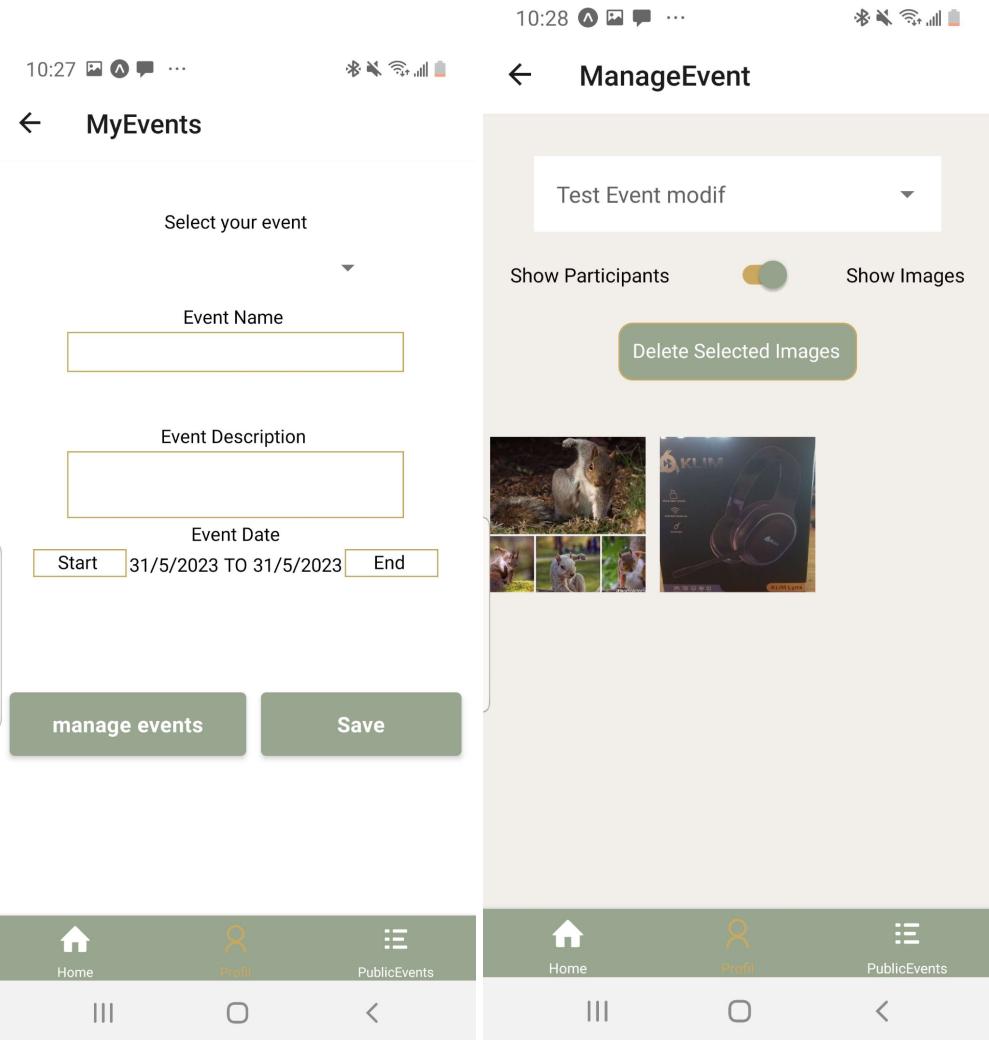
Sur la page profil de l'utilisateur il peut retrouver toutes les images qu'il a postées peut importe l'event ainsi que deux boutons permettant la navigation vers la gestion des events et la page de modification du profil



Sur la page profil l'utilisateur pourra modifier ses informations de profil.

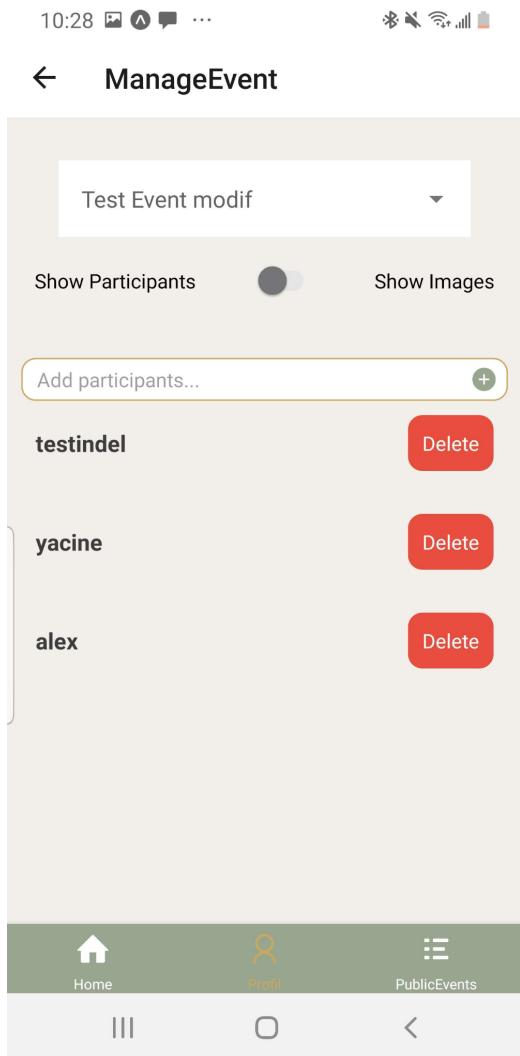
6.Demo Application

Le Screen Manage Event permet à l'utilisateur de gérer ses événements dans un premier temps les informations de l'event.



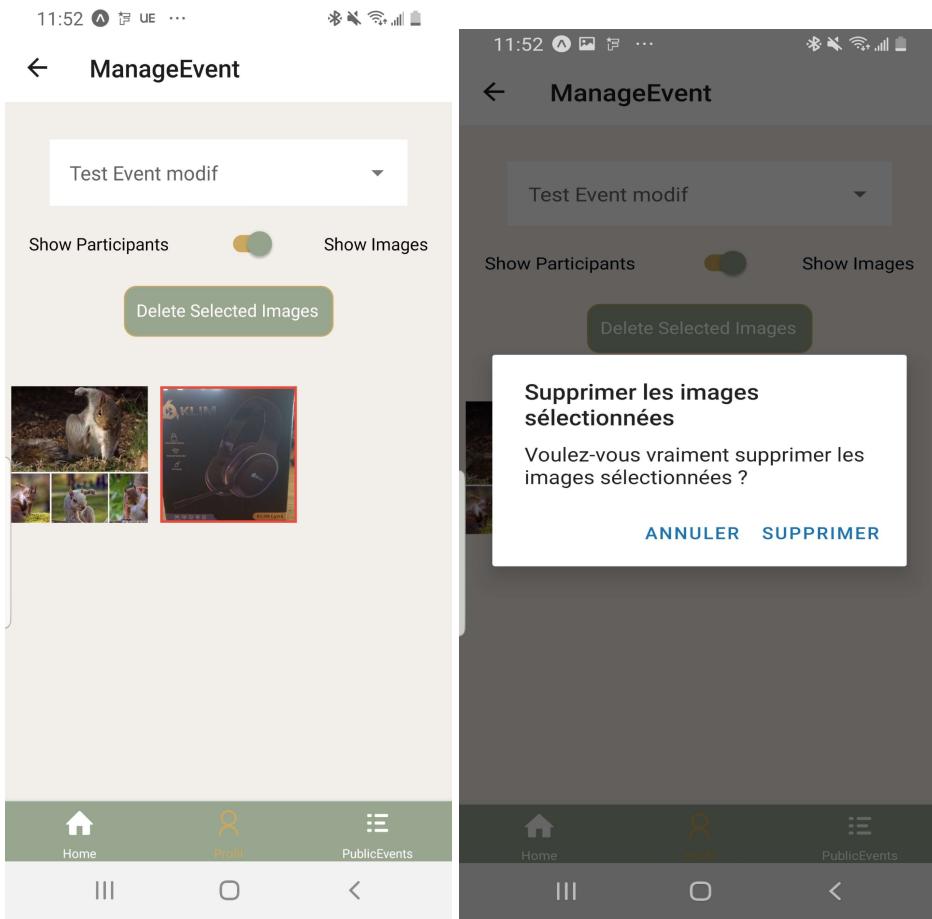
Le deuxième écran lui permet de gérer la suppression d'image sur un event non voulu par le créateur de l'event ainsi que les participants .(voir ci dessous)

6.Demo Application



Ici l'utilisateur créateur de l'event peut gérer les participants à son event, les supprimants de la liste de participants à l'event et leur accès au mur .

6.Demo Application



Les images sélectionnées seront encadrées par une bordure rouge et ensuite un message de confirmation pour la suppression.

7. Sécurité

7.2 Nettoyage des entrées utilisateur pour prévenir les attaques XSS :
Les attaques XSS (Cross-Site Scripting) sont une menace courante pour la sécurité des applications web. Pour prévenir ces attaques, j'ai utilisé la bibliothèque sanitize-html pour nettoyer les données entrées par les utilisateurs avant de les envoyer au serveur.

Par exemple, lors de l'enregistrement d'un nouvel utilisateur, nous appliquons le nettoyage des champs tels que le nom, le prénom, l'e-mail et le mot de passe en utilisant sanitize-html. Cela supprime tout contenu potentiellement dangereux et réduit considérablement le risque d'exécution de code malveillant sur le client. Voici un exemple de code qui montre comment utiliser sanitize-html pour nettoyer les données avant de les envoyer au serveur

```
import sanitizeHtml from 'sanitize-html';
```

```
try {
    //Nettoyage des champs pour éviter les XSS
    const cleanName = sanitizeHtml(name);
    const cleanLastName = sanitizeHtml(lastname);
    const cleanEmail = sanitizeHtml(email);
    const cleanPassword = sanitizeHtml(password);
```

```
await axios.post(`http://10.10.14.90:3000/users/register`, {
    name: cleanName ,
    lastname : cleanLastName,
    email: cleanEmail,
    password: cleanPassword
});
setIsLoading(false);
setError({ message: 'Welcome to Share Event', statusCode: 200 });
//navigation.navigate('LogIn');
} catch (e) {
    setError({ message: 'Something went wrong ! please retry in a few minutes', statusCode: 0 });
    setIsLoading(false);
}
```

7.Sécurité

7.3 bcrypt,middleware pour la sécurité côté backend :

Dans notre application backend, nous avons mis en place plusieurs mesures de sécurité pour protéger les données des utilisateurs et assurer l'intégrité de l'application. Nous avons utilisé bcrypt pour le hachage sécurisé des mots de passe, jsonwebtoken pour la génération et la vérification des tokens JWT, ainsi qu'un middleware pour assurer l'accès autorisé à certaines ressources.

Lorsque les utilisateurs s'inscrivent ou mettent à jour leur mot de passe, j'ai utilisé la bibliothèque bcrypt pour effectuer un hachage sécurisé de leur mot de passe. Le hachage de mot de passe est une technique de sécurité qui transforme le mot de passe en une chaîne de caractères aléatoire et irréversible. Cela garantit que même en cas de violation de la base de données, les mots de passe des utilisateurs ne peuvent pas être facilement récupérés.

Dans L'exemple ci-dessous, nous utilisons bcrypt.hash() pour hacher le mot de passe fourni par l'utilisateur avant de l'enregistrer dans la base de données. Cela garantit que les mots de passe sont stockés de manière sécurisée et ne peuvent pas être déchiffrés. Voici un exemple de code qui montre comment utiliser bcrypt pour le hachage sécurisé des mots de passe :

```
async create(user: CreateUserDto) {
  const passwordHash= await bcrypt.hash(user.password,10);
  const getUserbefore= await this.usersRepository.findAndCountBy({isAdmin:true});

  if(getUserbefore[1] >=1){
    return this.usersRepository.save(
      {
        email:user.email,
        name:user.name,
        url:user.url,
        lastname:user.lastname,
        password: passwordHash,
        isAdmin:false,
      }
    )
  }
}
```

7.Sécurité

7.4 jsonwebtoken

Pour l'authentification des utilisateurs et la gestion des sessions, nous utilisons la bibliothèque jsonwebtoken (JWT). JSON Web Tokens sont des chaînes de caractères qui contiennent des informations cryptées sur l'utilisateur et peuvent être utilisées pour vérifier l'identité de l'utilisateur à chaque requête.

Lorsque les utilisateurs se connectent avec succès, un token JWT est généré et renvoyé au client. Ce token est ensuite inclus dans l'en-tête des requêtes suivantes pour authentifier l'utilisateur. Nous utilisons la bibliothèque jsonwebtoken pour générer et vérifier ces tokens JWT.

```
if (!passwordMatch) {
    return 'l email ou le mot de passe est incorrect';
}

const payload = {
    id:foundUser.id,
    url:foundUser.url,
    email:foundUser.email,
    name:foundUser.name,
    lastname:foundUser.lastname,
    isAdmin:foundUser.isAdmin,
}

return {
    access_token: this.jwtService.sign(payload),
```

7. Sécurité

7.5 Middleware :

En plus de bcrypt et jsonwebtoken, nous utilisons un middleware pour assurer l'accès autorisé à certaines ressources de l'application. Le middleware que nous utilisons vérifie si un token JWT valide est présent dans l'en-tête de la requête et s'il contient les autorisations nécessaires pour accéder à la ressource demandée.

Le middleware que nous avons fourni précédemment s'appelle AdminGuard. Il vérifie si l'utilisateur a un token valide et s'il a le statut d'administrateur. Si les conditions sont remplies, l'accès est autorisé. Sinon, l'accès est refusé.

Voici un exemple de code qui montre comment utiliser le middleware AdminGuard pour protéger une route spécifique dans notre application backend :

```
@Injectable()
export class AdminGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  canActivate(context: ExecutionContext): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = request.headers.authorization; // assuming the token is sent in the "Authorization" header
    console.log(token);
    if (token == null || token== undefined) {
      return false;
    }

    try{
      const decodedToken = this.jwtService.decode(token) as { isAdmin: boolean };
      console.log(decodedToken.isAdmin)
      if(decodedToken.isAdmin == true){
        return true
      }
      else{
        return false
      }
    }
    catch{
      return false
    }
  }
}
```

7.Sécurité

Ainsi, avec l'utilisation de ces techniques et mesures de sécurité, nous nous assurons que notre application offre un environnement sécurisé pour les utilisateurs et protège leurs données confidentielles.

8. Conclusion

En conclusion, notre application mobile a été soigneusement conçue et développée en utilisant une gamme d'outils et de technologies modernes pour fournir une expérience utilisateur de premier ordre et une fonctionnalité robuste. La conception de l'interface a été réalisée avec Figma, nous permettant de créer un design visuel attrayant et une interface utilisateur intuitive. Pour le développement front-end, nous avons utilisé React Native et Expo, avec TypeScript pour apporter la robustesse et la sécurité du typage statique à notre code JavaScript. Cette combinaison offre une expérience utilisateur fluide et performante sur les plateformes iOS et Android à partir d'un seul codebase.

Pour le back-end, nous avons utilisé NestJS avec TypeORM pour créer une API RESTful efficace et bien structurée. Le choix de ces technologies a permis de mettre en place une architecture solide, flexible et facile à maintenir.

Nous avons également mis en œuvre des middlewares personnalisés pour renforcer la sécurité de notre application. Par exemple, notre middleware AdminGuard permet de restreindre l'accès à certaines fonctionnalités et ressources uniquement aux utilisateurs ayant les priviléges d'administrateur, garantissant ainsi l'intégrité et la confidentialité des données.

En dépit des défis rencontrés, le projet a mis en évidence l'efficacité de ces technologies dans le développement d'applications mobiles. Le résultat est une application fonctionnelle, esthétiquement plaisante et facile à utiliser qui répond bien aux besoins des utilisateurs.

9. Annexe

Recherche Anglophone : <https://docs.expo.dev/tutorial/image-picker/>

2 Pick an image from the device's media library

`expo-image-picker` provides the `launchImageLibraryAsync()` method that displays the system UI for choosing an image or a video from the device's media library.

We can use the button with the primary theme we created in the previous chapter to pick an image from the device's media library. We'll create a function to launch the device's image library to implement this functionality.

In `App.js`, import the `expo-image-picker` library and create a `pickImageAsync()` function inside the `App` component:

Choisir une image depuis la librairie du téléphone(appareil).

`expo-image-picker` fournit la méthode `launchImageLibraryAsync()` qui affiche l'interface utilisateur pour choisir une image ou une vidéo de l'appareil .

Nous pouvons utiliser le bouton avec le thème primaire que nous avons créé dans le chapitre précédent pour choisir une image de la bibliothèque de l'appareil.

Nous allons créer une fonction pour lancer la bibliothèque d'image de l'appareil afin d'implémenter cette fonctionnalité .