React Native
Chapitre 2 - Module 1
Aller plus loin avec React Native

Déroulé et objectif

Cette formation est découpée en 2 chapitres et 3 modules, qui serviront de fil rouge à la formation.

À la fin de ce module, vous saurez :

- Stocker des données sur le stockage local d'un appareil.
- Structurer un projet complexe, en respectant les bonnes pratiques.
- Gérer des interactions avec l'utilisateur
- Traiter des données fournies par un utilisateur
- Créer un CRUD pour votre application
- Mettre en place un filtre de recherche
- Concevoir une UI améliorant l'UX de votre application

Objectif du module 1 : Créer une application de prise de notes avec une sauvegarde locale des données.

Objectif final du chapitre 1 : Créer une application de prise de notes.

Organiser son projet

Pour nos projets, nous avons utilisé une architecture simple car nous n'avions pas beaucoup de fichiers, mais comment organiser la structure d'une application complexe ?

En règle générale, on retrouve à la racine d'une application React Native (utilisant expo) :

- Le fichier **App.js** (ou **App.tsx**)
- Le fichier **app.json**
- Le fichier **babel.config.js**
- Les fichiers **package.json** et **package-lock.json**
- Le dossier **node_modules**
- Un dossier **src**, qui contiendra les fichiers de développement
- Un dossier **scripts**, qui contient les scripts d'automatisation de build

Dans le dossier **src** :

- Un dossier **components**, qui contiendra les composants
- Le dossier **assets**, qui contiendra les images, audios, vidéos, polices, etc.
- Un dossier **shared**, qui contiendra les éléments partagés à toute l'application (énumérations, constantes, interfaces)
- Un dossier **database** qui contiendra les requêtes vers la base de données
- Un dossier **navigation** qui contiendra les éléments liés à la navigation dans l'application
- Un dossier **pages** ou **screens**, qui contiendra toutes les vues présentées à l'utilisateur

Création du projet noteApp

Nous allons créer une application de prise de notes, que nous nommerons **noteApp**.

Pour initialiser votre nouveau projet, entrez la commande :

```
npx create-expo-app noteApp
```

Note : Le dossier **Android**, qui serait installé lors d'une installation avec la CLI React Native, contient les fichiers de configuration **Gradle**, si vous développez une application complexe, certaines modifications manuelles pourront être appliquées, pour contourner des problèmes recensés avec certaines librairies.

Placez-vous dans le dossier de votre application :

```
cd noteApp
```

Entrez la commande suivante pour activer la visualisation **Expo** sur un navigateur :

```
npx expo install react-dom react-native-web
@expo/webpack-config
```

Installons les dépendances utiles à l'avance :

```
npm install @react-navigation/native@latest
@react-navigation/stack@latest
@react-native-community/masked-view@latest
react-native-screens@latest
react-native-safe-area-context@4.5.0
react-native-gesture-handler@latest
react-native-vector-icons @rneui/base @rneui/themed
```

Ouvrez votre serveur **Expo** et **Expo Go** pour vérifier l'installation.

Installation et mise en place de AsyncStorage

Pour notre application de prise de notes, nous aurons besoin de la fonctionnalité **AsyncStorage**, qui nous permettra de stocker des données sous la forme **Clé:Valeur**, en local sur le téléphone :

```
npx expo install
@react-native-async-storage/async-storage@1.17.11
```

Pour importer **AsyncStorage**, il faudra utiliser la ligne d'import suivante (dans le fichier **AsyncFunctions.js**) :

```
import AsyncStorage from
'@react-native-async-storage/async-storage';
```

AsyncStorage nous permettra uniquement de stocker des **strings**, pour stocker des données plus complexes, il faudra penser à utiliser **JSON.stringify()**, avant la sauvegarde des données et **JSON.parse()** à la lecture de ces données. La taille maximale par défaut est de **6 MB**.

Sauvegarder un **string** avec **AsyncStorage** :

```
const storeData = async (value) => {
    try {
      await AsyncStorage.setItem('@key', value)
    } catch (e) {// lance une erreur}
  }
```

Sauvegarder une donnée complexe avec **AsyncStorage** :

```
const storeData = async (value) => {
    try {
      const jsonValue = JSON.stringify(value)
      await AsyncStorage.setItem('@key, jsonValue)
    } catch (e) {
      // lance une erreur
    }
  }
```

Lire une donnée sous forme de **string** avec **AsyncStorage** :

```
const getData = async () => {
    try {
      const value = await AsyncStorage.getItem('@key)
      if(value !== null) {
        // donnée stocké
      }
    } catch(e) { // lance une erreur }
  }
```

Lire une donnée complexe avec **AsyncStorage** :

```
const getData = async () => {
    try {
      const jsonValue = await AsyncStorage.getItem('@key)
      return jsonValue != null ? JSON.parse(jsonValue) : null;
    } catch(e) {
      // lance une erreur
    }
  }
```

Pour ce projet, nous mettrons en place une architecture plus structurée.

Commencez par créer un dossier **src**, à la racine de votre projet, et transférez le dossier **assets** à l'intérieur.

Ouvrez le fichier **app.json** pour modifier les appels au dossier **assets** des paramètres suivants et pointer sur **./src/assets/** :

- **"icon"**
- **"splash" > "image"**
- **"android" > "foregroundImage"**
- **"web" > "favicon"**

Dans le dossier **src**, créez les dossiers suivants :

- **components**
- **screens**
- **shared**

Dans le dossier **shared**, créez le dossier **functions** et à l'intérieur du dossier

functions, créez un fichier nommé **AsyncFunctions.js**.

Ce fichier accueillera les fonctions liées à **AsyncStorage** et sera partagé avec tous les fichiers utilisant le stockage local.

Dans ce fichier, importez **AsyncStorage** :

```
import AsyncStorage from
"@react-native-async-storage/async-storage";
```

Nous allons créer deux fonctions qui nous permettront de tester l'implémentation de **AsyncStorage** et de voir un exemple de mise en place.

Le première fonction se nommera **initProfileName**, devra être asynchrone et exportée.

Elle prendra en paramètre **name**, qui sera **null** par défaut**.**

Le seconde fonction se nommera **getProfileName**, devra être asynchrone et exportée.

Elle ne prendra pas de paramètre.

Dans la fonction **initProfileName**, nous commencerons par mettre en place un **try/catch**, pour retourner les erreurs potentielles dans la console :

```
try {
```

```
} catch(e) {

    console.log('Erreur initProfileName

 : ' + e);

}
```

Dans le **try**, nous allons stocker, dans une constante nommée **jsonValue**, le retour asynchrone de **AsyncStorage.getItem()**, sur la clé **'@profile'** :

```
const jsonValue = await AsyncStorage.getItem('@profile')
```

Nous allons ensuite mettre en place une condition **if**, pour permettre à notre fonction si **jsonValue** est **null**, de mettre en place l'**item** ayant pour clé **'@profile'** en lui passant la valeur de **name** si elle n'est pas nulle ou la valeur **"Connected User"** par défaut et nous attendrons le retour de cette instruction :

```
if(jsonValue === null) {

            await AsyncStorage.setItem('@profile',
JSON.stringify(name !== null ? name : "Connected User"))

        }
```

Pour la fonction **getProfileName()**, nous utiliserons également un **try/catch** pour retourner les potentielles erreurs :

```
try {
```

```
} catch(e) {

    console.log('Erreur getProfileName : ' + e)

}
```

Dans le **try**, nous allons stocker, dans une constante nommée **value**, le retour asynchrone de **AsyncStorage.getItem()**, sur la clé '**@profile**' :

```
const value = await AsyncStorage.getItem('@profile')
```

Nous allons ensuite mettre en place une condition **if**, pour permettre à notre fonction si **value** n'est pas **null**, de retourner le contenu de la constante **value**, après avoir appliqué un **JSON.parse()** dessus. Sinon, on retourne la valeur "**Invité**" :

```
if(value !== null) {

          return JSON.parse(value);

        } else {

            return 'Invité';

        }
```

Vous devriez donc avoir le code suivant après ces étapes :

```
export async function initProfileName(name = null) {

    try {

       const jsonValue = await AsyncStorage.getItem('@profile')

       if(jsonValue === null) {
```

```
            await AsyncStorage.setItem('@profile',
JSON.stringify(name !== null ? name : "Connected User"))

        }

      } catch(e) {

          console.log('Erreur initProfileName : ' + e);

      }

}

export async function getProfileName() {

    try {

        const value = await AsyncStorage.getItem('@profile')

        if(value !== null) {

          return JSON.parse(value);

        } else {

            return 'Invité';

        }

      } catch(e) {

          console.log('Erreur getProfileName : ' + e)

      }

}
```

Exercice : Créez un screen de test

Nous allons créer un **screen** de test, qui aura pour but d'afficher la valeur

récupérée par notre fonction **getProfileName()** et de l'afficher dans un cadre avec un bouton pour simuler la connexion au profil et changer la valeur du state **connectedUser** afin de modifier l'affichage du nom.

Dans le dossier **screens**, créez le fichier **ProfileScreen.js** et intégrez les imports suivants :

- **React** et **useState**
- **View, Text, TouchableOpacity et StyleSheet**
- **{ initProfileName, getProfileName } from '../shared/functions/AsyncFunctions';**

Créez le composant **ProfileScreen** et exportez le par défaut.

Dans votre composant **ProfileScreen**, créez un **useState** qui servira à la gestion de **connectedUser** et **setConnectedUser**. Par défaut, il devra avoir la valeur **"Invité"**. Appelez la fonction **initProfileName()** pour initialiser le nom enregistré au chargement du composant.

Créez la méthode **loadProfile()** qui utilisera notre fonction **getProfileName()** et qui gèrera le **callback** renvoyé par cette fonction.

Si **getProfileName()** s'exécute correctement et envoie un retour, on veut le récupérer pour modifier le state de **connectedUser.**

Le composant **ProfileScreen** devra retourner une **vue**, qui contiendra un **Text**

affichant le message “**Bonjour,** ” suivi du nom stocké dans **connectedUser**.

Dans cette vue, vous devez également retourner un **TouchableOpacity**, qui enclenchera la méthode **loadProfile()** lors d’une pression sur la zone.

Ce **TouchableOpacity** contiendra un simple composant **Text**, qui contiendra le texte **“Charger le profil de test”**.

Pour les styles du composant, vous pouvez utiliser ces instructions, que vous devrez replacer au bon endroit :

```
const styles = StyleSheet.create({
    container: {
        margin:24,
        borderWidth:1,
        padding:32,
    },
    hello:{
        textAlign:"center",
    },
    buttonStyle: {
        borderWidth:2,
        borderColor:"#333",
```

```
        paddingLeft:15,
        paddingRight:15,
        paddingTop:5,
        paddingBottom:5,
        marginTop:24,
    },
    buttonText:{
        textAlign:"center",
    }
})
```

Pensez à créer votre fichier **index.js**, dans le dossier **screens** pour mettre en place votre liste d'**exports** de **screens**.

Pour tester votre **screen**, ouvrez le fichier **App.js** et importez **{ProfileScreen} from './src/screens'**, puis implémentez le composant **<ProfileScreen/>** dans la vue de **App**.

Exercice bonus pour les plus rapides

Si vous avez terminé l'exercice avant le reste du groupe, vous pouvez :

- Créer de nouvelles fonctions dans **AsyncFunctions.js** :
    - **setProfileName**, qui permettra d'ajouter une valeur sur la clé **'@profile'.**
    - **mergeProfileName**, qui permettra de fusionner la valeur de **'@profile'** avec une nouvelle valeur. Si **'@profile'** n'existe pas, il faudra déclencher **initProfileName()** avant.
    - **deleteProfileName**, qui permettra de supprimer l'**item '@profile'.**
    - **logProfileName**, qui permettra d'effectuer un **console.log** qui affichera la valeur de **'@profile'**, si elle existe.

Rendu de l'exercice

Pour l'exercice précédent, les critères qui seront étudiés lors de la correction sont les suivants :

- Organisation du code
- Respect des instructions
- Utilisation d'un useState
- Utilisation du callback de **getProfileName()** pour modifier la valeur de **connectedUser**
- Implémentation de **ProfileScreen**, dans le fichier **App.js**.

Création des dossiers et des fichiers nécessaires à l'application de prise de notes

Mettons en place la structure de notre application de prise de notes.

Nous allons créer à l'avance tous les fichiers et dossiers nécessaires au projet, puis nous les implémenterons au cas par cas pour avancer dans la création de notre application.

Pensez à implémenter votre export par défaut en créant les fichiers.

Nous aurons besoin de créer la structure de dossiers suivante :

- **src**
    - **assets**
    - **components**
    - **navigation**
    - **screens**
    - **shared**

Dans le dossier **components**, nous aurons besoin des fichiers suivants :

- **index.js**
- **Note.js**
- **NoteDetail.js**
- **NoteInputModal.js**
- **NotFound.js**
- **ReplacementView.js**
- **RoundIconBtn.js**
- **SearchBar.js**

Dans le dossier **navigation**, nous aurons besoin des fichiers suivants :

- **index.js**
- **NavigationTabs.js**

Dans le dossier **screens**, nous aurons besoin des fichiers suivants :

- **HomeScreen.js**
- **index.js**
- **NoteScreen.js**
- **NoteListScreen.js**
- **ProfileScreen.js**

Dans le dossier **shared**, nous aurons besoin des **dossiers** suivants :

- **constants**
- **context**
- **functions**
- **theme**

Dans le dossier **shared>constants**, nous aurons besoin des **fichiers** suivants :

- **ScreenSize.js**

Dans le dossier **shared>context**, nous aurons besoin des **fichiers** suivants :

- **index.js**
- **NoteProvider.js**
-

Dans le dossier **shared>functions**, nous aurons besoin des **fichiers** suivants :

- **index.js**
- **AsyncFunctions.js**
- **SortFunctions.js**

Dans le dossier **shared>theme**, nous aurons besoin des **fichiers** suivants :

- **colors.js**

Ouvrez le fichier **colors.js**, placé dans le dossier **shared>theme** et ajoutez le code suivant :

```
export default {
   BLACK: "#333",
   WHITE: "#fff",
   DARK: "#2A2F4F",
   ULTRALIGHT: "#fef5fb",
   LIGHT: "#FDE2F3",
   PRIMARY: "#917FB3",
   SECONDARY: "#E5BEEC",
   ERROR: "#C2241D"
}
```

Ce fichier contiendra une liste des couleurs que nous utiliserons pour concevoir l'application.

Dans le fichier **SortFunctions.js**, placé dans le dossier **shared>functions**, ajoutez le code suivant :

```
export function reverseIntDatas(data){

   return data.sort((a, b) => {

      const aInt = parseInt(a.time || a.id);

      const bInt = parseInt(b.time || b.id);

      if (aInt < bInt) return 1;

      if (aInt === bInt) return 0;

      if (aInt > bInt) return -1;

   });

}
```

Cette fonction de tri nous servira à trier les notes ajoutées par l'utilisateur, dans l'ordre décroissant, en se basant sur la valeur **time** ou **l'id** de la note.

Dans le fichier **AsyncFunctions.js**, placé dans le dossier **shared>functions,** nous allons avoir besoin de nouvelles fonctions :

```
export async function initUsername(name = null) {
  try {
    const jsonValue = await AsyncStorage.getItem('@username')
    if (jsonValue === null) {
      await AsyncStorage.setItem('@username', JSON.stringify(name !== null ? name : "NewUser"))
    }
  } catch (e) {
    console.log('Erreur initUsername : ' + e);
  }
}
```

```
export async function setUsername(username = null) {
  try {
    if (username !== null) {
      await AsyncStorage.setItem('@username', JSON.stringify(username));
    }
  } catch (e) {
    console.log('Erreur setUsername : ' + e);
  }
}
```

```
export async function mergeUsername(newValue = null) {
  try {
    if (newValue !== null) {
      const jsonValue = await AsyncStorage.getItem('@username')
      if (jsonValue === null) {
        initUsername();
      }
      await AsyncStorage.mergeItem('@username',
JSON.stringify(newValue));
    }
  } catch (e) {
  }
}
export async function getUsername() {
  try {
    const value = await AsyncStorage.getItem('@username');
    if (value !== null) {
      return JSON.parse(value);
    } else {
      return null;
    }
  } catch (e) {
    console.log('Erreur getUsername : ' + e)
```

```
  }
}
export async function deleteUsername() {
  try {
    await AsyncStorage.removeItem('@username');
  } catch (e) {
    console.log('Erreur deleteUsername : ' + e);
  }
}
```

```
export async function logUsername() {
  try {
    const jsonValue = await AsyncStorage.getItem('@username)
    if (jsonValue !== null) {
      console.log(`Username : ${jsonValue}`);
    } else {
      console.log("Username is null.")
    }
  } catch (e) {
    console.log('Erreur logUsername : ' + e);
  }
}
```

Ces fonctions vont nous permettre d'interagir avec le stockage local et de manipuler les données liées à la clé **@username**.

Dans le fichier **NoteProvider.js**, placé dans le dossier **shared>context**, ajoutez le code suivant :

```
import React, {createContext, useContext, useState, useEffect}
from 'react';

import AsyncStorage from
'@react-native-async-storage/async-storage';
```

Ce fichier aura le rôle de **Provider**, pour nos notes. Ce provider nous permettra de créer un contexte, qui sera utilisable par nos fichiers, pour gérer les données spécifiques au notes et pouvoir les modifier depuis n'importe quel composant.

```
const NoteContext = createContext();


const NoteProvider = ({children}) => {

   const [notes, setNotes] = useState([]);

   const findNotes = async () => {

      const result = await AsyncStorage.getItem("@notes");

      if (result !== null) {

         setNotes(JSON.parse(result));

      }

   }
```

```
    useEffect(() => {
        findNotes();
    }, []);

    return (
        <NoteContext.Provider value={{ notes, setNotes, findNotes
}}>
            {children}
        </NoteContext.Provider>
    );
}
export const useNotes = () => useContext(NoteContext);

export default NoteProvider;
```

Nous utiliserons donc **useNotes** dans les fichiers ayant besoin d'interagir avec les données liées aux notes et elles seront toutes conservées dans **NoteContext**.

Dans le fichier **ScreenSize.js**, placé dans le dossier **shared>constants**, ajoutez le code suivant :

```
import { Dimensions } from 'react-native';


export function getWidth(){
  const windowWidth = Dimensions.get('window').width;
  return windowWidth;
}


export function getHeight(){
  const windowHeight = Dimensions.get('window').height;
  return windowHeight;
}
```

Ces fonctions vont nous permettre de connaître la taille de l'écran utilisé.

Nous allons donc implémenter le code nécessaire au fonctionnement de notre application, en commençant par refactorer le fichier **App.js**.

Nous allons voir chaque fichier en détail par la suite et nous allons étudier le fonctionnement interne et les mises en lien pour que vous ayez toutes les cartes en main pour créer une nouvelle application utilisant ce système de stockage.

Refactorisation et intégration du nouveau code

App.js

Dans le fichier **App.js** :

```
import React, { useState, useEffect } from 'react';
import { getUsername } from "./src/shared/functions/AsyncFunctions";
import { NavigationContainer } from '@react-navigation/native';
import { NavigationTabs } from './src/navigation';
import {NoteProvider} from './src/shared/context';

export default function App() {

  const [userName, setUserName] = useState({});

  const findUser = async () => {
    const result = await getUsername('@username');
    if (result !== undefined && result !== null) {
      setUserName(JSON.parse(result));
    } else {
      setUserName(undefined);
    }
  }
```

```
  useEffect(() => {
    findUser();
  }, [])

  const modifyGlobalUsername = (newName) => {
    setUserName({ name: newName });
  }
  return (
    <NavigationContainer>
      <NoteProvider>
        <NavigationTabs userName={userName?.name}
modifyGlobalUsername={modifyGlobalUsername} />
      </NoteProvider>
    </NavigationContainer>
  );
}
```

Le fichier **App.js** va rester assez simple, il implémenter un **state** de **userName**, qui sera passé aux composants sans **Provider** pour vous présenter une autre manière de passer un **state** d'un composant **parent** à un composant **enfant**.

La fonction **findUser** n'est passée que dans le composant **App**, nous lui ajoutons donc cette méthode.

On utilise un **useEffect** pour exécuter la méthode **findUser()** au chargement de **App.js** uniquement.

La méthode **modifyGlobalUsername** est une méthode permettant de

modifier **userName**, qui sera fournie à certains composants enfants qui pourraient nécessiter la modification de **userName**.

App retournera donc :

- Un composant communautaire **NavigationContainer**, pour englober la navigation entre nos écrans.
- Un composant natif **NavigationTabs** qui effectuera la gestion de nos onglets de navigation (et des écrans cachés).

src > navigation > NavigationTabs.js

Ouvrez désormais le fichier **navigation>NavigationTabs.js** :

```
import React, { useState, useEffect } from 'react';

import { createBottomTabNavigator } from
'@react-navigation/bottom-tabs';

import HomeScreen from '../screens/HomeScreen';

import NoteScreen from '../screens/NoteScreen';

import NoteListScreen from '../screens/NoteListScreen';

import { Foundation as FoundationIcons } from
'react-native-vector-icons';

import { StyleSheet } from 'react-native';

import colors from '../shared/theme/colors';

import {getUsername} from '../shared/functions/AsyncFunctions';


const Tab = createBottomTabNavigator();
```

```
const NavigationTabs = ({ userName, modifyGlobalUsername }) => {
   const [navUserName, setNavUserName] = useState(null);
   const verifyUsername = async (username) => {
      if(username !== null && username !== undefined){
         const result = await
getUsername().then((existingUsername)=>{
            return JSON.parse(existingUsername)?.name;
         });
         if(result !== null && result !== undefined && typeof(result)
=== "string"){
            setNavUserName(result);
         }
      }
   }

   useEffect(() => {
      if (userName !== undefined && userName !== null) {
         setNavUserName(userName.name);
      }
      verifyUsername(userName);
   }, []);

   return (
      <Tab.Navigator
```

```
        initialRouteName="Home"
        backBehavior="history"
      >
        <Tab.Screen
          name="Home"
          style={styles.tabScreen}
          options={{
            tabBarLabel: "Accueil",
            tabBarActiveTintColor: "#917FB3",
            tabBarInactiveTintColor: "#E5BEEC",
            tabBarIcon: ({ color, size }) => (
              <FoundationIcons name="home" color={color}
size={size} />
            ),
            title: "Accueil",
          }}
        >
          {(props) => <HomeScreen {...props} userName={userName}
setNavUserName={setNavUserName}
modifyGlobalUsername={modifyGlobalUsername}/>}
        </Tab.Screen>

        <Tab.Screen
          name="NotesList"
          style={styles.tabScreen}
```

```
            listeners={{
               tabPress: e => {
                  verifyUsername(userName);
                  if (navUserName === null || navUserName ===
undefined) {
                     e.preventDefault();
                     alert("Vous devez entrer votre prénom pour
accéder à cette fonctionnalité.");
                  }
               }
            }}
            options={{
               tabBarLabel: "Liste de notes",
               tabBarActiveTintColor: "#917FB3",
               tabBarInactiveTintColor: "#E5BEEC",
               tabBarIcon: ({ color, size }) => (
                  <FoundationIcons name="clipboard-notes"
color={color} size={size} />
               ),
               title: `Liste de notes de ${userName || "Invité"}`,
            }}
         >
            {(props) => <NoteListScreen {...props}
userName={userName} />}
         </Tab.Screen>
```

```
        <Tab.Screen
           name="NoteScreen"
           style={styles.tabScreen}
           options={{
              /**
              * tabBarVisible + tabBarButton permettent de masquer
l'icône de la tapBar.
              */
              tabBarVisible: false,
              tabBarButton: (props) => null,
              tabBarLabel: `Note`,
              tabBarActiveTintColor: "#917FB3",
              tabBarInactiveTintColor: "#E5BEEC",
              tabBarIcon: ({ color, size }) => (
                 <FoundationIcons name="page-search" color={color}
size={size} />
              ),
              title: `Note de ${userName || "Invité"}`,
           }}
        >
           {(props) => <NoteScreen {...props} userName={userName}
/>}
        </Tab.Screen>
     </Tab.Navigator>
```

```
    )
}
const styles = StyleSheet.create({


tabScreen: {
      color: colors.LIGHT,
    }
})
export default NavigationTabs;
```

Tab contiendra la méthode **createBottomTabNavigator()**, nous permettant de créer une navigation sous forme de **TabBar**.

NavigationTabs prendra les **props** {**userName** et **modifyGlobalUsername**} pour pouvoir les passer à des écrans spécifiques par la suite.

On commence par implémenter le state **navUserName** qui nous servira d'élément de comparaison pour s'assurer que l'utilisateur a enregistré un nom avant d'essayer d'accéder à certaines fonctionnalités.

La méthode **verifyUsername** nous permettra d'effectuer cette vérification et d'inclure la potentielle donnée dans **navUserName**.

Dans le **useEffect**, nous initialisons **navUserName** avec les informations

disponibles.

NavigationTabs retournera un **Tab.Navigator**, ayant pour props **initialRouteName** et **backBehavior**.

initialRouteName nous permet de définir la route initiale de l'application, ici la route menant vers **Home** (donc **HomeScreen**).

backBehavior nous permet de spécifier le mode de retour, au clic sur le bouton de retour du téléphone.

Par défaut, **Tab.Navigator** envoie la page initiale, nous allons donc le configurer sur **history** pour renvoyer le dernier écran ouvert par l'utilisateur.

Contrairement aux **Tab.Screen** que nous avons pu implémenter dans les précédents projets, nous aurons besoin de communiquer des **props** aux **screens** dans cette application.

Tab.Screen sera donc utilisé en balise **ouvrante + fermante** et le composant sera englobé par ces balises.

À l'intérieur de nos **Tab.Screen**, nous allons implémenter nos composants et spécifier leurs **props**.

```
{(props) => <HomeScreen {...props} userName={userName}
setNavUserName={setNavUserName}
modifyGlobalUsername={modifyGlobalUsername}/>}
```

Pour **HomeScreen**, on passera des props globales et des props définies.

Les props globales seront récupérées avec le paramètre **props** et passées dans un **REST operator** et les props définies seront passées avec leur nom

définitif.

Pour **NoteListScreen**, nous aurons besoin du prop **userName** qui nous permettra d'afficher le nom de l'utilisateur à certains endroits de la page.

Nous implémenterons également une vérification dans **Tab.Screen**, affichant un message d'alerte si l'utilisateur n'a pas encore défini de nom et l'accès à la page sera bloqué tant que le nom ne sera pas défini.

```
{(props) => <NoteListScreen {...props} userName={userName} />}
```

NoteScreen utilisera également le prop **userName**.

```
{(props) => <NoteScreen {...props} userName={userName} />}
```

src > navigation > index.js

Nous allons créer le fichier **index.js**, dans le dossier **navigation** pour y implémenter la liste d'export de nos composants de navigation.

Créez le fichier **index.js** et ajoutez l'import, puis l'export de **NavigationTabs** :

```
import NavigationTabs from "./NavigationTabs";



export{

   NavigationTabs,

}
```

src > shared > constants > ScreenSize.js

Nous allons désormais implémenter les fonctionnalités partagées de l'application, qui se situeront dans le dossier **shared**.

Dans le dossier **constants**, créez le fichier **ScreenSize.js**.

Ce fichier contiendra deux méthodes exportables qui nous serviront à connaître la largeur et la hauteur de l'écran, en se basant sur les données fournies par le composant **Dimensions** de React Native.

```
import { Dimensions } from 'react-native';


export function getWidth(){
  const windowWidth = Dimensions.get('window').width;
  return windowWidth;
}


export function getHeight(){
  const windowHeight = Dimensions.get('window').height;
  return windowHeight;
}
```

src > shared > context > NoteProvider.js

Nous allons désormais implémenter les fonctionnalités partagées de l'application, qui se situeront dans le dossier **shared**.

Dans le dossier **context**, créez le fichier **NoteProvider.js**.

Ce fichier gèrera l'état global des notes de l'application et sera en capacité de renvoyer un **provider** qui nous permettra d'avoir accès à cet état global.

```
import React, {createContext, useContext, useState, useEffect}
from 'react';

import AsyncStorage from
'@react-native-async-storage/async-storage';


const NoteContext = createContext();


const NoteProvider = ({children}) => {


   const [notes, setNotes] = useState([]);


   const findNotes = async () => {

      const result = await AsyncStorage.getItem("@notes");

      if (result !== null) {

         setNotes(JSON.parse(result));

      }
```

```
    }

    useEffect(() => {
        findNotes();
    }, []);

    return (
        <NoteContext.Provider value={{ notes, setNotes, findNotes
}}>
            {children}
        </NoteContext.Provider>
    );
}

export const useNotes = () => useContext(NoteContext);

export default NoteProvider;
```

src > shared > context > index.js

Créez le fichier **index.js** dans le dossier **context** pour permettre l'export du **provider** en ciblant le dossier **src > shared > context**.

```
import NoteProvider from './NoteProvider';



export{

   NoteProvider,

}
```

src > shared > theme > colors.js

Créez le fichier **colors.js** dans le dossier **theme.** Ce fichier nous permettra de définir les couleurs que nous utiliserons pour l'application, dans une liste d'export.

```
export default {

   BLACK: "#333",

   WHITE: "#fff",

   DARK: "#2A2F4F",

   ULTRALIGHT: "#fef5fb",

   LIGHT: "#FDE2F3",

   PRIMARY: "#917FB3",

   SECONDARY: "#E5BEEC",

   ERROR: "#C2241D"

}
```

Implémentation des écrans

screens > HomeScreen.js

Créez le fichier **HomeScreen.js**, dans le dossier **screens**.

Le composant **HomeScreen** renverra l'écran d'accueil.

Nous aurons besoin des dépendances suivantes :

```
import React, { useState, useEffect } from 'react';

import { View, Text, StyleSheet, TextInput, Dimensions } from
'react-native';

import colors from '../shared/theme/colors';

import { getWidth, getHeight } from
'../shared/constants/ScreenSize';

import { RoundIconBtn, ReplacementView } from '../components/';

import { setUsername, getUsername } from
"../shared/functions/AsyncFunctions";
```

Avant l'ouverture de notre composant, nous aurons à implémenter 3 constantes :

```
const screenWidth = getWidth();

const screenHeight = getHeight();

const debug = false;
```

screenWidth et **screenHeight** contiendront les méthodes fournies par les méthodes implémentées dans notre fichier **ScreenSize.js** et **debug** nous permettra de spécifier si le mode **debug** de l'écran est actif.

Si le mode debug est actif, nous afficherons la valeur de **name**, qui sera un état de notre composant **HomeScreen**, à l'intérieur d'un composant **Text**.

```
const HomeScreen = ({ userName, modifyGlobalUsername, setNavUserName,
navigation }) => {

return (

      <View style={styles.container}>

         <Text style={styles.inputTitle}>

            Entrez le prénom qui sera associé aux notes

         </Text>

         <TextInput

            style={styles.textInput}

            placeholder="Exemple : Jean"

            value={name}

            onChangeText={handleTextChange}

         />

         {debug ? <Text style={styles.userText}> [ {name} ] </Text> :
null}

         {name?.trim()?.length > 0 ? <RoundIconBtn
iconName="arrow-right" iconType="foundation" color={colors.WHITE}
size={24} onPress={handleSubmit} style={{ borderRadius:12,
padding:16,}} /> : <ReplacementView width="100%" padding={40} />}

      </View>

   )

}
```

Notre composant **HomeScreen** utilisera les **props** suivants :

```
{ userName, modifyGlobalUsername, setNavUserName, navigation }
```

Ce composant aura besoin de gérer l'état de **name**, nous allons donc lui implémenter un **useState** ayant pour valeur initiale une chaîne de caractères vide :

```
const [name, setName] = useState("");
```

Ce composant utilisera un **useEffect** pour effectuer des actions lors de son rendu. Ce useEffect utilisera une méthode, pensez à bien implémenter cette méthode **avant** le useEffect.

```
const initUsername = async () => {

      getUsername()

         .then((newUser) => {

            if(newUser !== null){

               setName(JSON.parse(newUser).name);

            }

         })

   }

   useEffect(() => {

      if (name === "") {

         initUsername();

      }
```

```
      if (userName !== "" && name !== userName) {
         setName(userName);
      }
   }, []);
```

Au rendu du composant, on spécifiera donc avec le useEffect que nous souhaitons effectuer une vérification des données passées et modifier l'état de **name** si **userName** contient une valeur n'étant pas identique à l'état initial de **name**.

Notre composant utilisera également plusieurs méthodes (que l'on implémentera entre l'ouverture du composant et l'ouverture du retour).

```
const handleTextChange = (text) => {
      setName(text.trim());
   };
const handleSubmit = async () => {
      const user = { name: name };
      await setUsername(JSON.stringify(user));
      setNavUserName(name);
      modifyGlobalUsername(name);
      navigation.navigate("NotesList");
}
```

Notre composant utilisera ces instructions de style :

```
const styles = StyleSheet.create({
   container: {
      flex: 1,
      backgroundColor: colors.ULTRALIGHT,
      alignItems: "center",
      justifyContent: "center",
   },
   inputTitle: {
      alignSelf: "flex-start",
      textAlign: "center",
      paddingLeft: 24,
      marginTop: 3,
      marginBottom: 3,
      opacity: 0.5,
   },
   textInput: {
      width: (screenWidth - 50),
      height: 48,
      borderWidth: 1,
      borderColor: colors.DARK,
      borderRadius: 6,
```

```
      paddingLeft: 12,
      fontSize: 24,
      marginTop: 12,
      marginBottom: 12,
      color: colors.PRIMARY,
   },
   userText: {
      textAlign: "center",
      color: colors.DARK,
      fontSize: 24,
   },
});
```

N'oubliez pas d'implémenter l'**export default** du composant, à la fin de votre fichier.

```
export default HomeScreen;
```

screens > NoteListScreen.js

Créez le fichier **NoteListScreen.js**, dans le dossier **screens**.

Cet écran sera chargé de l'affichage de toutes les notes enregistrées.

Pour le mettre en place, nous aurons besoin des imports suivants :

```
import React, { useState, useEffect, useCallback } from 'react';

import { RefreshControl, View, StyleSheet, Text, Alert, FlatList
} from 'react-native';

import colors from '../shared/theme/colors';

import { Note, SearchBar, RoundIconBtn, NoteInputModal, NotFound
} from '../components';

import AsyncStorage from
"@react-native-async-storage/async-storage";

import { useNotes } from "../shared/context/NoteProvider";

import {reverseIntDatas} from
"../shared/functions/SortFunctions";
```

Le composant **NoteListScreen** utilisera les props **userName** et **navigation** :

```
const NoteListScreen = ({ userName, navigation }) => {}
```

Ce composant utilisera les états suivants :

```
const [name, setName] = useState("");

const [modalVisible, setModalVisible] = useState(false);

const [searchQuery, setSearchQuery] = useState("");

const { notes, setNotes, findNotes } = useNotes();

const [resultNotFound, setResultNotFound] = useState(false);
```

Nous lui implémenterons également la possibilité de rafraîchir la vue :

```
const [refreshing, setRefreshing] = useState(false);

const onRefresh = useCallback(() => {

      setRefreshing(true);

      setTimeout(() => {

         findNotes();

         setRefreshing(false);

      }, 325);

   }, []);
```

Et nous lui imposerons l'exécution de certaines actions au rendu avec un **useEffect** :

```
useEffect(() => {

      if (userName !== "" && name !== "" && userName !== name) {

         setName(userName);

      }

      findNotes();

   }, []);
```

Nous souhaitons que l'affichage des notes se fasse dans l'ordre décroissant, nous utiliserons donc notre méthode **reverseIntDatas** :

```
const reverseNotes = reverseIntDatas(notes);
```

NoteListScreen aura besoin de méthodes supplémentaires pour gérer les interactions avec l'écran :

```
const toggleModal = () => {
      setModalVisible(!modalVisible);
   };
   const handleOnSubmit = async (title, description) => {
      let author = (userName !== undefined && userName !== null) ?
userName : null;
     const actualTime = Date.now();
      const note = {
         id: actualTime ,
         title: title,
         description: description,
         author: author,
         created_at: actualTime,
      }
      const updatedNotes = [...notes, note];
      setNotes(updatedNotes);
      await AsyncStorage.setItem("@notes",
JSON.stringify(updatedNotes));
   }
```

```
  const handleOnSearchInput = async (text) => {
    setSearchQuery(text);
    if (!text.trim()) {
      setSearchQuery("");
      setResultNotFound(false);
      return await findNotes();
    }
    const filteredNotes = notes.filter(note => {
      if
(note.title.toLowerCase().includes(text.toLowerCase())) {
        return note;
      }
    });
    if (filteredNotes.length > 0) {
      setNotes([...filteredNotes]);
    } else {
      setResultNotFound(true);
    }
  }
  const handleOnClearSearchInput = async () => {
    setSearchQuery("");
    setResultNotFound(false);
```

```
    return await findNotes();
  }
```

Le composant retournera donc :

```
return (
    <View style={styles.container}>
      {notes.length ?
        <SearchBar
          value={searchQuery}
          onChangeText={handleOnSearchInput}
          onClear={handleOnClearSearchInput}
          containerStyle={{
            marginVertical: 15,
          }}
        />
        : null}

      {resultNotFound ?
        (<NotFound navigation={navigation}/>)
        : (<FlatList
          data={reverseNotes}
          numColumns={2}
```

```
                refreshing={refreshing}
                onRefresh={onRefresh}
                columnWrapperStyle={{
                   justifyContent: "space-between",
                   marginBottom: 15,
                }}
                keyExtractor={item => item.id.toString()}
                renderItem={
                   ({ item }) =>
                      <Note
                         item={item}
                         onPress={() =>
navigation.navigate("NoteScreen",
                            {
                               note: item,
                               noteId: item.id,
                               noteTitle: item.title,
                               noteDescription: item.description,
                               noteAuthor: item.author,
                            }
                         )
                         }
```

```
                    />
                }
            />)}


        {!notes.length ? <View
style={[StyleSheet.absoluteFillObject,
styles.emptyHeaderContainer]}>
            <Text style={styles.emptyHeader}>
                Add Notes
            </Text>
        </View> : null}
        <View style={styles.addBtnContainer}>
            <RoundIconBtn
                iconName="clipboard-pencil"
                iconType="foundation"
                style={styles.addBtn}
                color={colors.WHITE}
                onPress={toggleModal}
            />
        </View>
        <NoteInputModal
```

```
            isEdit={false}
            visible={modalVisible}
            toggleModal={toggleModal}
            modalRequestClose={() => {
               Alert.alert("Quitter", "Souhaitez-vous quitter
l'ajout de note ?", [
                  {
                     text: 'Non',
                  },
                  {
                     text: 'Oui',
                     onPress: () =>
                        toggleModal()
                  },
               ]);
            }}
            onSubmit={handleOnSubmit} />
      </View>
   )
}
```

Nous lui appliquerons ce style :

```
const styles = StyleSheet.create({
   container: {
      flex: 1,
      backgroundColor: colors.ULTRALIGHT,
      justifyContent: "flex-start",
      paddingVertical: 6,
      paddingHorizontal: 12,
      zIndex: 1,
   },
   emptyHeaderContainer: {
      flex: 1,
      justifyContent: "center",
      alignItems: "center",
      zIndex: -1,
   },
   emptyHeader: {
      fontSize: 30,
      textTransform: "uppercase",
      fontWeight: "700",
      opacity: 0.2,
   },
   addBtnContainer: {
```

```
      width: null,
      height: null,
      position: 'absolute',
      right: 15,
      bottom: 15,
      zIndex: 3,
   },
   addBtn: {
      padding: 12,
      borderRadius: 12,
   },
})
```

Et nous n'oublierons pas de le rendre exportable :

```
export default NoteListScreen;
```

screens > NoteScreen.js

Créez le fichier **NoteScreen.js**, dans le dossier **screens**.

Cet écran nous permettra de gérer la visualisation d'une note entrée dans l'application et stockée dans le stockage local du téléphone.

Pour cet écran, nous aurons besoin des imports suivants :

```
import React, { useState, useEffect } from 'react';

import { View, StyleSheet, Alert } from 'react-native';

import colors from '../shared/theme/colors';

import { NoteDetails, RoundIconBtn, NoteInputModal } from
"../components";

import AsyncStorage from '@react-native-async-storage/async-storage';

import { useNotes } from '../shared/context/NoteProvider';
```

Notre écran prendra en compte les propriété suivantes : **userName**, **route** et **navigation** :

```
const NoteScreen = ({ userName, route, navigation }) => {



}
```

Cet écran aura besoin d'une gestion d'état sur des paramètres passés via la propriété **route**.

Les paramètres nécessaires à l'écran seront définis lors de la redirection, à l'appel de la méthode **navigation.navigate()**, pour permettre la récupération des informations sur la note actuelle et prendra un état initial se basant sur la valeur de **route.params.note.**

```
const { setNotes } = useNotes();

const [note, setNote] = useState(route.params.note);

const [isEdit, setIsEdit] = useState(false);

const [showModal, setShowModal] = useState(false);

const [name, setName] = useState("");
```

setNotes utilisera notre méthode **useNotes()** (crée dans le **NoteProvider**) pour définir le nouveau contexte de l'écran.

Ce contexte permettra à l'écran **NoteScreen** d'utiliser le composant global **NoteProvider** et nous permettra d'avoir accès aux méthodes et aux requêtes liées aux notes.

Nous effectuerons une vérification au rendu de l'écran pour initialiser le nom de l'utilisateur.

```
useEffect(() => {

     if (userName !== "" && name !== "" && userName !== name) {

        setName(userName);

     }

  }, []);
```

Au niveau des méthodes liées à cet écran, nous souhaitons :

- Pouvoir supprimer une note du stockage local

- Afficher une alerte de sécurité à l'utilisateur pour éviter une suppression de la note par erreur
- Gérer la mise à jour d'une note
- Utiliser une modal pour afficher l'écran d'édition d'une note
- Pouvoir appeler une modal en modifiant l'état de **isEdit**, pour gérer les manipulations qui seront effectuées lors de la modification d'une note

```
const deleteNote = async () => {
      const result = await AsyncStorage.getItem("@notes");
      let notes = [];
      if (result !== null) {
         notes = JSON.parse(result);
      }

      const newNotes = notes.filter(item => item.id !== note.id);
      await AsyncStorage.setItem("@notes", JSON.stringify(newNotes));
      setNotes(newNotes);
      navigation.goBack();
   }
```

La méthode **deleteNote** nous permettra de supprimer une note du stockage local, en ciblant l'objet **@notes** et nous mettrons à jour les données du stockage en filtrant les données n'ayant pas le même **id** que la note venant d'être supprimée.

```
const displayDeleteAlert = () => {
      Alert.alert("Supprimer la note", "Souhaitez-vous supprimer cette
```

```
note ?", [
        {
           text: "Annuler",
        },
        {
           text: "Supprimer",
           onPress: deleteNote
        }
     ],
        {
           cancelable: true,
        }
     );
  }
```

La méthode **displayDeleteAlert** nous permettra d'afficher un message de confirmation à la suppression d'une note, pour éviter toute erreur de la part de l'utilisateur.

```
const handleUpdate = async (title, description, time) => {
     const result = await AsyncStorage.getItem("@notes");
```

```
    let notes = [];
    if(result !== null){
       notes = JSON.parse(result);
    }
    const newNotes = notes.filter(thisNote => {
       if(thisNote.id === note.id){
          thisNote.title = title;
          thisNote.description = description;
          thisNote.isUpdated = true;
          thisNote.time = time;
          setNote(thisNote);
       }
       return thisNote;
    });
    setNotes(newNotes);
    await AsyncStorage.setItem("@notes", JSON.stringify(newNotes));
  }
```

La méthode **handleUpdate** nous servira à gérer la mise à jour d'une note, en nous basant sur l'**id** de la note.

```
const toggleModal = () => {
    setShowModal(!showModal);
```

```
  };
```

La méthode **toggleModal** nous permettra de gérer l'état de **showModal**.

```
const openEditModal = () => {
      setIsEdit(true);
      toggleModal();
  }
```

La méthode **openEditModal** nous permettra de spécifier l'état de **isEdit**, pour passer en mode **Édition de note**.

Nous allons désormais implémenter le retour de notre composant, en englobant ce retour dans un composant **View** :

```
return (
      <View style={styles.container}>


      </View>
 );
```

Sur cet écran, nous commencerons par implémenter le composant **NoteDetails**, qui aura besoin de la prop **note** :

```
<NoteDetails
    note={note}
/>
```

Nous allons ensuite implémenter une vue, qui contiendra nos composants **RoundIconBtn**, qui auront pour rôle de permettre à l'utilisateur d'effectuer des actions concernant la note :

```
<View style={styles.btnsContainer}>
            <RoundIconBtn
               iconName="arrow-left"
               iconType="foundation"
               size={24}
               color={colors.WHITE}
               style={styles.returnBtn}
               onPress={() => navigation.goBack()}
            />
            <RoundIconBtn
               iconName="trash"
               iconType="foundation"
               size={36}
               color={colors.WHITE}
               style={styles.deleteBtn}
```

```
            onPress={() => displayDeleteAlert()}
          />
          <RoundIconBtn
            iconName="pencil"
            iconType="foundation"
            size={24}
            color={colors.WHITE}
            style={styles.modifyBtn}
            onPress={openEditModal}
          />
        </View>
```

Nous aurons ainsi : un bouton permettant de quitter la vue de la note, un bouton permettant de supprimer la note (en affichant un avertissement de sécurité, pour éviter la suppression par erreur de cette note) et un bouton permettant d'ouvrir une **Modal** en mode **Édition** pour modifier cette note.

Nous implémenterons en dessous de la vue contenant ces boutons, le composant **Modal** permettant l'édition de la note, sous condition d'une modification de l'état **showModal**.

```
{showModal ? <NoteInputModal
            isEdit={isEdit}
```

```
            note={note}
            toggleModal={toggleModal}
            modalRequestClose={() => {
               Alert.alert("Quitter", "Souhaitez-vous quitter l'ajout
de note ?", [
                  {
                     text: 'Non',
                  },
                  {
                     text: 'Oui',
                     onPress: () =>
                        toggleModal()
                  },
               ]);
            }}
            onSubmit={handleUpdate}
            visible={showModal}

         /> : null}
```

Nous implémenterons ensuite les styles de notre vue :

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

```
  container: {
     flex: 1,
     backgroundColor: colors.ULTRALIGHT,
     justifyContent: 'center',
  },
  header: {
     fontSize: 25,
     fontWeight: "900",
  },
  btnsContainer: {
     width: "100%",
     maxHeight: 84,
     overflow: "hidden",
     flexDirection: "row",
     justifyContent: "space-between",
     alignItems: "center",
     paddingHorizontal: 12,
     position: "absolute",
     bottom: 0,
     right: 0,
     zIndex: 0,
  },
```

```
  checkBtn: {
     padding: 12,
  },
  returnBtn: {
     padding: 9,
     backgroundColor: colors.PRIMARY,
  },
  deleteBtn: {
     padding: 9,
     backgroundColor: colors.ERROR,
  },
  modifyBtn: {
     padding: 9,
     backgroundColor: colors.PRIMARY,
  }
})
```

Sans oublier l'export par défaut de notre composant :

```
export default NoteScreen;
```

screens > index.js

Nos écrans sont désormais prêts à accueillir nos futurs composants personnalisés, nous n'avons plus qu'à créer la liste d'export dans le dossier **screens**.

```
import HomeScreen from './HomeScreen';

import ProfileScreen from './ProfileScreen';

import NoteListScreen from './NoteListScreen';

import NoteScreen from './NoteScreen';

import NoteEdit from './NoteEdit';


export{

   HomeScreen,

   ProfileScreen,

   NoteListScreen,

   NoteScreen,

   NoteEdit,

}
```

components > Note.js

Le composant **Note** nous permettra de gérer la mise en forme des données et le retour d'une note. Ce composant aura besoin des imports suivants :

```
import React from 'react';

import { Text, StyleSheet, TouchableOpacity } from 'react-native';

import colors from "../shared/theme/colors";

import { getWidth } from "../shared/constants/ScreenSize"
```

Le composant **Note** aura besoin de deux props : **item** et **onPress.**

```
const Note = ({ item, onPress }) => {

const { title, description, author } = item;



}
```

On utilisera la notation de **déstructuration** pour stocker les données de la propriété **item** dans trois variables : **title, description** et **author.**

```
   const { title, description, author } = item;
```

Ce composant retournera une zone **TouchableOpacity**, contenant des composants **Text**, qui accueilleront nos valeurs :

```
return (

        <TouchableOpacity

           style={styles.container}

           onPress={onPress}
```

```
        >
            <Text numberOfLines={2} style={styles.title}>
               {title}
            </Text>
            <Text numberOfLines={2} style={styles.description}>
               {description}
            </Text>
            {author !== null ? <Text numberOfLines={2}
style={styles.author}>
               DE : {author}
            </Text> : null}
         </TouchableOpacity>
   );
```

On utilisera la propriété **numberOfLines** sur les composants **Text** pour limiter l'affichage du texte contenu à **2 lignes maximum**.

On stylisera ce composant avec les instructions suivantes :

```
const styles = StyleSheet.create({
   container: {
      width: (getWidth() / 2) - 50,
      alignSelf: "center",
      paddingVertical: 6,
```

```
    paddingHorizontal: 12,
    backgroundColor: colors.SECONDARY,
    margin: 12,
    borderRadius: 6,
  },
  title: {
    fontSize: 24,
    fontWeight: "bold",
    marginBottom: 12,
    color: colors.BLACK,
  },
  description: {
    minHeight: 48,
    fontSize: 18,
    color: colors.BLACK,
  },
  author: {
    minHeight: 20,
    fontSize: 12,
    color: colors.BLACK,
    opacity: 0.5,
  }
```

```
})
```

Et on ajoutera l'export par défaut du composant, s'il n'a pas été créé en même temps que le composant :

```
export default Note;
```

components > NoteDetails.js

Le composant **NoteDetails** nous permettra de gérer la mise en forme des données d'une note sélectionnée sur **NoteListScreen** et affichera la note complète, avec une information sur la date de création de la note.

Ce composant renverra uniquement l'affichage des informations de la note complète, mise en forme.

Nous utiliserons les imports suivants :

```
import React from 'react';

import { View, ScrollView, Text, StyleSheet } from 'react-native';

import colors from '../shared/theme/colors';
```

Le composant aura besoin de la propriété **note**, que l'on utilisera pour lui passer les informations complètes d'une note :

```
const NoteDetails = ({ note }) => {
```

```
}
```

On utilisera l'**id** de la note pour mettre en forme sa date de création (si la valeur **time** n'existe pas encore sur la note).

```
const getCreationDate = new Date(note.id);

const createdAt = `Ajoutée le : ${getCreationDate.getDay() < 10 ? '0'
+ getCreationDate.getDay() :
getCreationDate.getDay()}/${getCreationDate.getMonth() < 10 ? '0' +
getCreationDate.getMonth() :
getCreationDate.getMonth()}/${getCreationDate.getFullYear()}`;


const getUpdatedDate = new Date(note?.time || note.id);

const updatedAt = `Modifiée le : ${getUpdatedDate.getDay() < 10 ? '0'
+ getUpdatedDate.getDay() :
getUpdatedDate.getDay()}/${getUpdatedDate.getMonth() < 10 ? '0' +
getUpdatedDate.getMonth() :
getUpdatedDate.getMonth()}/${getUpdatedDate.getFullYear()}`;
```

Il retournera un composant **View**, qui englobera deux **View** et une **ScrollView.**

La première vue contiendra les informations sur l'auteur de la note et la date de création (ou de modification) de cette note.

La deuxième vue contiendra le titre de la note.

La **ScrollView** contiendra le contenu de la note.

```
return (
   <View style={styles.container}>
      <View style={styles.infosContainer}>
         <Text style={styles.author}>
            Ajoutée par {note.author}
         </Text>
         <Text style={styles.createdAt}>
            {note.isUpdated ? updatedAt : createdAt}
         </Text>
      </View>
      <View style={styles.titleContainer}>
         <Text style={styles.title}>
            {note.title}
         </Text>
      </View>
      <ScrollView style={styles.descriptionContainer}>
         <Text style={styles.description}>
            {note.description}
         </Text>
      </ScrollView>
   </View>
```

```
);

const styles = StyleSheet.create({
   container: {
      width: "100%",
      flex: 1,
      justifyContent: "flex-start",
      backgroundColor: colors.WHITE,
      zIndex: -1,
   },
   infosContainer: {
      flexDirection: "row",
      justifyContent: "space-between",
      flexWrap: "wrap",
      paddingHorizontal: 6,
      paddingVertical: 12,
      backgroundColor: colors.WHITE,
   },
   author: {
      fontSize: 12,
      fontWeight: "500",
   },
   createdAt: {
      fontSize: 12,
```

```
    fontWeight: "500",
  },
  titleContainer: {
    width: "100%",
    flexDirection: "row",
    flexWrap: "wrap",
    paddingHorizontal: 12,
    paddingVertical: 6,
  },
  title: {
    fontSize: 36,
    color: colors.DARK,
  },
  descriptionContainer: {
    width: "100%",
    paddingHorizontal: 12,
    paddingVertical: 6,
    marginBottom: 84,
  },
  description: {
    fontSize: 21,
    color: colors.BLACK,
  }
})
```

Et pensez à l'export du composant :

```
export default NoteDetails;
```

components > NoteInputModal.js

Le composant **NoteInputModal** nous permettra d'intégrer une **Modal** de **React Native Elements**, qui sera utilisée sur notre application pour l'ajout de notes et l'édition de notes.

Nous aurons besoin des imports suivants pour ce composant :

```
import React, { useState, useEffect } from 'react';

import { ScrollView, View, TextInput, StyleSheet, Alert, Keyboard }
from 'react-native';

import Modal from "react-native-modal";

import colors from "../shared/theme/colors";

import { TouchableWithoutFeedback } from
'react-native-gesture-handler';

import { getHeight } from '../shared/constants/ScreenSize';

import RoundIconBtn from "./RoundIconBtn";
```

Il faudra installer la dépendance **react-native-modal** :

```
npm i react-native-modal
```

Le composant **Modal** de React Native Elements ajoute une surcouche au composant core **Modal** (de React Native).

Cette surcouche nous permet de résoudre les bugs remontés sur la version de React Native actuelle (bugs qui seront corrigés lors de la montée de version en **0.72**).

Notre composant **NoteInputModal** utilisera les propriétés suivantes :

- **visible :** pour spécifier à la propriété **visible** du composant **Modal** si la modal est visible ou non (valeur booléenne).
- **toggleModal :** pour passer une méthode permettant d'afficher ou de masquer la modal, selon les besoin du composant parent.
- **modalRequestClose :** pour passer une méthode permettant de gérer l'état qui sera lié à l'affichage d'une modal, dans le composant parent.
- **onSubmit :** pour passer une méthode dédiée à la gestion des données à la soumission de données, utilisées par le composant parent, et entrées par l'utilisateur dans notre modal.
- **note :** pour passer les informations de la note.
- **isEdit :** pour spécifier si la modal est en mode édition ou non (valeur booléenne).

```
const NoteInputModal = ({ visible, toggleModal, modalRequestClose,
onSubmit, note, isEdit }) => {



}
```

Nous utiliserons **useState** pour gérer l'état du **titre** et de la **description** pouvant être implémentés dans la modal.

```
  const [title, setTitle] = useState("");

  const [description, setDescription] = useState("");
```

Au chargement du composant et à la modification de la valeur de **isEdit**, nous déclencherons un **useEffect** qui permettra de modifier l'état du **titre** et de la **description**.

On pensera à fournir **isEdit** en second paramètre, dans un **tableau**, pour la vérification s'effectue à chaque changement d'état de **isEdit**.

```
useEffect(() => {
      if (isEdit) {
         setTitle(note.title);
         setDescription(note.description);
      }
   }, [isEdit]);
```

Nous aurons besoin d'une méthode **handleOnChangeText()**, qui prendra les paramètres **text** et **valueFor**, et effectuera la gestion des données entrées par l'utilisateur dans les composants **TextInput** que nous utiliserons à l'intérieur du composant **Modal** :

```
const handleOnChangeText = (text, valueFor) => {
      if (valueFor === "title") {
         setTitle(text);
      }
```

```
    if (valueFor === "description") {
       setDescription(text);
    }
  }
```

Nous implémenterons une méthode **closeModal** pour gérer la fermeture de la modal et demander une confirmation à l'utilisateur sur l'action de fermeture de la modal

```
const closeModal = () => {
    Alert.alert("Quitter", "Voulez-vous quitter l'ajout de note ?",
    [
       { text: 'Non' },
       {
          text: 'Oui',
          onPress: () => {
             if (!isEdit) {
                setTitle("");
                setDescription("");
             }
             toggleModal();
          }
       },
    ]);
```

```
  }
```

Nous implémenterons une dernière méthode, **handleAddNewNote**, pour modifier l'état de **title** et de **description**, et pour gérer l'ajout d'une nouvelle note par l'utilisateur. Nous effectuerons également des vérifications sur le contenu fourni par l'utilisateur :

```
const handleAddNewNote = () => {
      if (title !== "" && description !== "") {
         if (isEdit) {
            onSubmit(title, description, Date.now());
         } else {
            onSubmit(title, description);
            setTitle("");
            setDescription("");
         }
         toggleModal();
      } else {
         if (title === "") {
            Alert.alert("Titre manquant", "Vous devez donner un titre
à la note.",
```

```
            [
               { text: 'OK' },
            ]);
         }


          if (title !== "" && description === "") {
            Alert.alert("Description manquante", "Vous devez ajouter
du contenu à la note avant de l'ajouter",
            [
               { text: 'OK' },
            ]);
         }
      }
   }
```

Concernant le retour de notre composant, il renverra un composant **Modal**, qui englobera :

- Un composant **View** contenant :
  - un **TextInput**
  - une **ScrollView** contenant :
    - un **TextInput**
  - un composant **View** contenant :
    - **2** composants **RoundIconBtn**
- Un composant **TouchableWithoutFeedback** contenant :

- un composant **View**

On commencera donc par implémenter la modal, qui utilisera les propriétés suivantes :

- **visible={visible}**
- **animationType=”slide”**
- **style={{ margin: 0 }}**
- **backgroundTransitionOutTiming={0}**
- **onRequestClose={modalRequestClose}**

```
return (
      <Modal
         visible={visible}
         animationType="slide"
         style={{ margin: 0 }}
         backgroundTransitionOutTiming={0}
         onRequestClose={modalRequestClose}
      >


    </Modal>
);
```

```
```

Dans cette modal, on implémenter les composants suivants :

```
<View

style={styles.container}

>



</View>

<TouchableWithoutFeedback

style={[styles.modalBackgroundContainer,StyleSheet.absoluteFillObject]
}

>



</TouchableWithoutFeedback>
```

Notre composant **View** aura uniquement une propriété de **style** (styles.container).

Notre composant **TouchableWithoutFeedback** aura une propriété de style, utilisant plusieurs styles différents :

- **styles.modalBackgroundContainer**, que nous implémenterons dans

notre **StyleSheet**.

- **StyleSheet.absoluteFillObject**, qui est une méthode du composant **StyleSheet** permettant de créer une superposition en position absolue et en position **0** (top, left, right, bottom) et nous permet d'implémenter des ajustements supplémentaires dans notre feuille de style.

À l'intérieur de notre composant **View**, nous allons implémenter :

Un composant **TextInput**, qui sera utilisé pour gérer le **titre** d'une note (affichage et modification) :

```
<TextInput
    value={title}
    placeholder="Titre"
    style={[styles.input, styles.title]}
    onChangeText={(text) => handleOnChangeText(text, "title")}
/>
```

Une **ScrollView** contenant un **TextInput** qui sera chargé d'accueillir la **description** et de gérer son affichage et sa modification :

```
<ScrollView style={styles.scrollView}>
    <TextInput
        value={description}
        multiline
        placeholder="Contenu de la note"
        style={[styles.input, styles.description]}
       onChangeText={(text) => handleOnChangeText(text,
"description")}
```

```
    />
</ScrollView>
```

Nous lui implémenterons également une nouvelle vue, contenant deux composants **RoundIconBtn**, qui nous permettront d'afficher un bouton permettant à l'utilisateur de fermer la modal et un bouton permettant la validation de l'ajout ou des modifications de la note :

```
<View style={styles.checkBtnContainer}>
    <RoundIconBtn
        iconName="x"
        iconType="foundation"
        size={24}
        color={colors.WHITE}
        style={styles.closeModalBtn}
       onPress={closeModal}
    />
    <RoundIconBtn
        iconName="plus"
        iconType="foundation"
        size={24}
        color={colors.WHITE}
        style={styles.checkBtn}
```

```
        onPress={handleAddNewNote}
    />
</View>
```

À l'intérieur du composant **TouchableWithoutFeedback**, nous implémenterons un composant **View :**

```
<View
    style={[styles.modalBackground, StyleSheet.absoluteFillObject]}
>


</View>
```

Cette vue viendra se placer en position absolue et nous permettra d'ajouter, dans la feuille de style, des instructions supplémentaires pour lui implémenter un **backgroundColor** et une position sur l'axe Z à -1 avec **zIndex**.

Nous allons donc spécifier les styles utilisés par notre composant, en commençant par initialiser notre constante **styles** :

```
const styles = StyleSheet.create({


})
```

Nous donnerons à notre container une disposition flexible, un espacement entre les éléments internes de **0**, une marge interne verticale de **12,** une position sur l'axe Z de **1** et une couleur de fond utilisant notre fichier **colors.js** pour implémenter la couleur **WHITE** :

```
container: {
      flex: 1,
      gap: 0,
      paddingVertical: 12,
      zIndex: 1,
      backgroundColor: colors.WHITE,
   },
```

Nous donnerons à notre **scrollView** une marge externe vers le bas de **84** :

```
   scrollView: {
      marginBottom: 84,
   },
```

Nous donnerons à notre **input** une bordure en bas d'une épaisseur de **1**, une couleur de bordure utilisant notre couleur **PRIMARY**, une taille de police de **24**px et une couleur de police utilisant notre couleur **DARK** :

```
input: {
      borderBottomWidth: 1,
      borderBottomColor: colors.PRIMARY,
      fontSize: 24,
      color: colors.DARK,
   },
```

Nous donnerons à notre **title** une largeur de **100%**, une hauteur correspondant à une fraction de la hauteur de notre écran (en utilisant **getHeight()** pour connaître la hauteur de l'écran et en divisant la hauteur par **10**), une graisse de police d'une valeur de **700**, un alignement de texte à gauche et une marge interne sur l'axe horizontal d'une valeur de **12** :

```
title: {
      width: "100%",
      height: (getHeight() / 10),
      fontWeight: "700",
      textAlign: "left",
```

```
    paddingHorizontal: 12,
  },
```

Nous donnerons à notre **description** une hauteur minimale (correspondant à (la hauteur maximale de notre écran - (la hauteur maximale de notre écran divisé par **10**) - **300**), un alignement de texte à gauche, une bordure en bas d'une épaisseur de **0**, une marge interne d'une valeur de **12** :

```
description: {
    minHeight: (getHeight() - (getHeight() / 10) - 300),
    textAlign: "left",
    borderBottomWidth: 0,
    padding: 12,
  },
```

Nous donnerons à notre **checkBtnContainer** une largeur de **100%**, une hauteur maximale de **84**, un overflow **hidden** (pour masquer les éléments susceptible de déborder du composant), une direction de la disposition flexible en ligne, une justification du contenu d'une valeur **space-between**, une marge interne sur l'axe horizontal d'une valeur de **12**, une position absolue avec un espacement du bas de l'écran d'une valeur de **15** et un espacement à droite d'une valeur de **0** et une position sur l'axe Z d'une valeur de **2** :

```
checkBtnContainer: {
    width: "100%",
    maxHeight: 84,
```

```
    overflow: "hidden",
    flexDirection: "row",
    justifyContent: "space-between",
    paddingHorizontal: 12,
    position: "absolute",
    bottom: 15,
    right: 0,
    zIndex: 2,
  },
```

Nous donnerons à notre **checkBtn** une marge interne d'une valeur de **12 :**

```
checkBtn: {
    padding: 12,
  },
```

Nous donnerons à notre **closeModalBtn** une marge interne d'une valeur de **12** et une couleur de fond utilisant notre couleur **ERROR** :

```
closeModalBtn: {
    padding: 12,
    backgroundColor: colors.ERROR,
  },
```

Nous penserons également à exporter le composant **NoteInputModal** par défaut :

```
export default NoteInputModal;
```

components > NotFound.js

Le composant **NotFound** nous permettra d'afficher un retour visuel si aucun contenu n'est trouvé lors d'une recherche avec notre composant **SearchBar**.

Ce composant aura besoin des imports suivants :

```
import React from 'react';

import { View, Text, StyleSheet } from 'react-native';

import RoundIconBtn from "./RoundIconBtn";

import colors from "../shared/theme/colors";
```

Ce composant ne comportera pas de paramètres ou de méthodes, il retournera uniquement ses composants :

```
const NotFound = () => {

     return (



     )

}
```

Il retournera une vue :

```
<View

     style={[StyleSheet.absoluteFillObject, styles.container]}

>



</View>
```

Cette vue englobera une nouvelle vue, contenant un **RoundIconBtn** qui prendra comme propriétés :

- **iconName="x"**
- **iconType="foundation"**
- **size={48}**
- **color={colors.BLACK}**
- **style={styles.icon}**

```
<View style={styles.iconContainer}>

    <RoundIconBtn

        iconName="x"

        iconType="foundation"

        size={48}
```

```
        color={colors.BLACK}
        style={styles.icon}
      />
</View>
```

À la suite de cette vue, la vue principale de notre composant englobera également un composant **Text** :

```
<Text
style={{ marginTop: 20, fontSize: 24, alignSelf: "center", }}
>
      Aucun résultat
</Text>
```

Nous initialiserons ensuite la constante **styles** pour créer une feuille de style avec **StyleSheet** :

```
const styles = StyleSheet.create({


})
```

Notre **container** utilisera ces instructions de style :

```
container: {
      flex: 1,
      justifyContent: "center",
      alignItems: "center",
```

```
    opacity: 0.5,
    zIndex: -1,
    paddingTop: 84,
  },
```

Notre **iconContainer** utilisera ces instructions de style :

```
  iconContainer: {
    borderRadius: 256,
    alignSelf: "center",
  },
```

Notre **icon** utilisera ces instructions de style :

```
icon: {
    padding: 6,
    backgroundColor: "transparent",
    shadowColor: "transparent",
    alignSelf: "center",
  }
```

N'oubliez pas d'exporter votre composant :

```
export default NotFound;
```

components > ReplacementView.js

Le composant **ReplacementView** nous servira à réserver un espace minimal sur l'écran, si une icône doit être utilisée sous condition.

Il aura besoin des imports suivants :

```
import React from 'react';

import { View, StyleSheet } from 'react-native';
```

Il aura besoin des propriétés **width**, **height** et **padding** :

```
const ReplacementView = ({ width, height, padding }) => {

   return (

      <View style={[styles.replacementView, {

         width: width,

         height: height,

         padding:padding || 48,

      }]}></View>

   );
```

```
}
```

Il retournera uniquement une vue à la taille spécifiée, qui bloquera l'espace à l'écran avant le remplacement par notre composant **RoundIconBtn**.

Il aura besoin des styles suivants :

```
const styles = StyleSheet.create({

   replacementView: {

      backgroundColor: "transparent",

      alignSelf:"center",

   }

})
```

Et nous penserons à l'exporter par défaut :

```
export default ReplacementView;
```

components > RoundIconBtn.js

Le composant **RoundIconBtn** nous permettra d'implémenter des icônes, fournies par **React Native Elements,** rapidement sur les fichiers de notre application.

Il aura besoin des imports suivants :

```
import React from 'react';

import { StyleSheet } from 'react-native';

import { Icon } from '@rneui/base';

import colors from '../shared/theme/colors';
```

Il aura besoin des propriétés **iconName**, **iconType, size, color style** et **onPress** :

```
const RoundIconBtn = ({ iconName, iconType, size, color, style,
onPress }) => {

     return (


     )
```

```
}
```

Il retournera un composant **Icon** uniquement :

```
<Icon

    type={iconType || "antdesign"}

    name={iconName}

    size={size || 24}

    color={color || colors.LIGHT}

    style={[styles.icon, {...style}]}

    onPress={onPress}

/>
```

Ce composant **Icon** aura besoin d'un type pour définir le type de bibliothèque d'icônes à utiliser, un nom d'icône, une taille d'icône, une couleur d'icône et nous lui donnerons des styles supplémentaires ainsi que la possibilité de réagir à une pression de l'utilisateur.

Nous lui donnerons ce style :

```
const styles = StyleSheet.create({

   icon:{

      backgroundColor: colors.PRIMARY,
```

```
      padding:24,
      borderRadius: 50,
      alignSelf:"center",
      elevation:2,
      shadowRadius: 50,
      shadowColor: colors.DARK,
      margin:12,
      aspectRatio: 1/1,
      zIndex:3,
   }
})
```

Puis nous exporterons le composant par défaut :

```
export default RoundIconBtn;
```

components > SearchBar.js

Le composant **SearchBar** nous permettra d'implémenter une barre de recherche pour la liste de notes.

Il aura besoin des imports suivants :

```
import React, {useState} from 'react';

import { View, StyleSheet, TextInput } from 'react-native';

import { getWidth, getHeight } from "../shared/constants/ScreenSize";

import colors from "../shared/theme/colors";

import RoundIconBtn from "./RoundIconBtn";
```

Il utilisera les propriétés **containerStyle, value, onChangeText** et **onClear** :

```
const SearchBar = ({ containerStyle, value, onChangeText, onClear })
=> {

   return (



   )

}
```

Il retournera dans un premier temps, une vue qui englobera le reste des composants :

```
<View
    style={[styles.container, { ...containerStyle }]}
>


</View>
```

Nous implémenterons dans cette vue un **TextInput** qui représentera la barre de recherche et qui utilisera les propriétés héritées du composant parent **value** et **onChangeText** :

```
<TextInput
    value={value}
    onChangeText={onChangeText}
    style={styles.searchBar}
    placeholder="Entrez votre recherche"
/>
```

Nous implémenterons en dessous de ce **TextInput**, une nouvelle vue implémentant conditionnellement un composant **RoundIconBtn** (si **value** n'est pas **null** ou **undefined** ou **vide**) :

```
<View style={styles.iconContainer}>
            {value ?
               <RoundIconBtn
                  iconName="x"
                  iconType="foundation"
                  size={18}
                  color={colors.WHITE}
                  style={styles.icon}
                  onPress={onClear}
               />
               : null}
         </View>
```

Nous initialiserons la feuille de style :

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

```
})
```

Et nous utiliserons les styles suivants :

```
container: {
      flexDirection: "row",
      justifyContent: "center",
      alignItems: "center",
      width: "100%",
      height: (getHeight() / 8),
      minHeight: 50,
      maxHeight: 50,
      padding: 5,
      marginBottom: 24,
   },
   searchBar: {
      width: (getWidth() - 40),
      borderWidth: 0.5,
      borderColor: colors.PRIMARY,
      height: 40,
      borderRadius: 40,
```

```
    paddingLeft: 12,
    paddingRight: 12,
    fontSize: 20,
    color: colors.PRIMARY,
  },
  iconContainer:{
    position:"absolute",
    right:6,
    zIndex:2,
  },
  icon: {
    padding: 6,
  }
```

Avant d'exporter notre composant par défaut :

```
export default SearchBar;
```

components > index.js

Nous allons désormais pouvoir remplir notre liste d'exportation, dans le fichier **index.js** :

```
import RoundIconBtn from "./RoundIconBtn";
import ReplacementView from "./ReplacementView";
import SearchBar from "./SearchBar";
import NoteInputModal from "./NoteInputModal";
import Note from "./Note";
import NoteDetails from "./NoteDetails";
import NotFound from "./NotFound";

export{
   RoundIconBtn,
   ReplacementView,
   SearchBar,
   NoteInputModal,
   Note,
   NoteDetails,
```

```
    NotFound,

}
```