BTS SIO 2022 Corentin JACQUIER

Documentation Technique

Petility

SUP ALTA
CAMPUS SACRÉ-CŒUR DES CHARTREUX

Table des matières

I - Introduction **2**

II - Base de données **2**

A - Modélisation 2

B - Mise en place 3

B - Configuration 5

III - Routeur **6**

IV - Authentification **7**

A - Création du compte 7

B - Connexion au compte 7

C - Déconnexion du compte 8

D - Suppression du compte 9

V - Interface **11**

A - Page d'accueil 11

B - Graphiques 14

I - Introduction

Nous allons étudier le fonctionnement de l'application mobile Petility qui est une application de gestion et au suivi de la santé de vos animaux. L'application est codée en *React Native* un framework produit par FaceBook écrit en Javascript, Java, C++, Objective C et Python. Contrairement à la bibliothèque React, React Native s'exécute dans un arrière-plan sans manipuler le DOM (Documentation Object Model) grâce au DOM virtuel. Le code de l'application est disponible sur *Github* et les différents documents sont disponibles sur mon *portfolio*.

Ce dernier est ainsi conçu pour développer des applications responsives sur les plateformes Android et iOS, ainsi de nombreuses applications utilisent React Native notamment les applications de Facebook, Discord, Skype mais aussi des jeux.

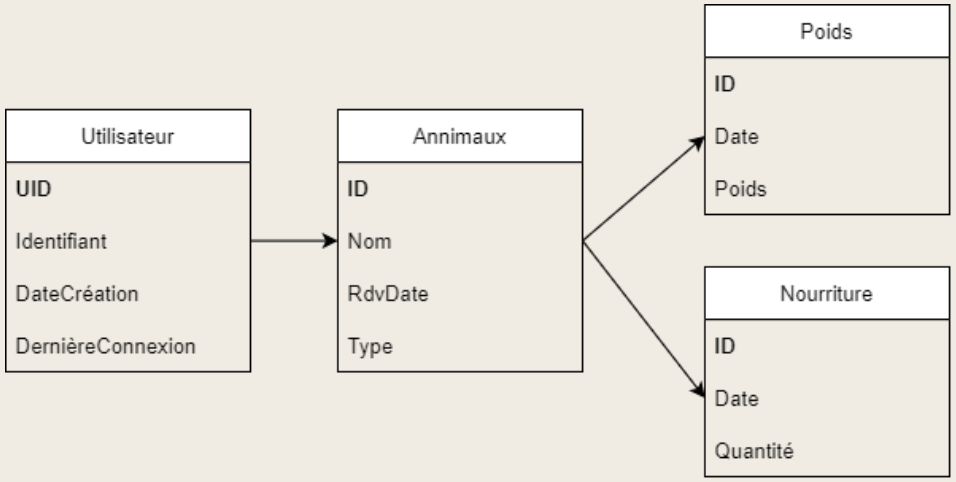
Enfin, concernant la base de données, celle-ci est en place avec *Firebase* de Google. En effet, ce service de stockage en ligne créé par Google permet de gérer l'authentification aisément et concernant l'application Petility, elle utilise la base de données Firestore Cloud. Alors, ses données sont externalisées sur les serveurs de Google sur une interface NoSQL.

II - Base de données

La base données est donc hébergée sur la structure Firestore de Google.

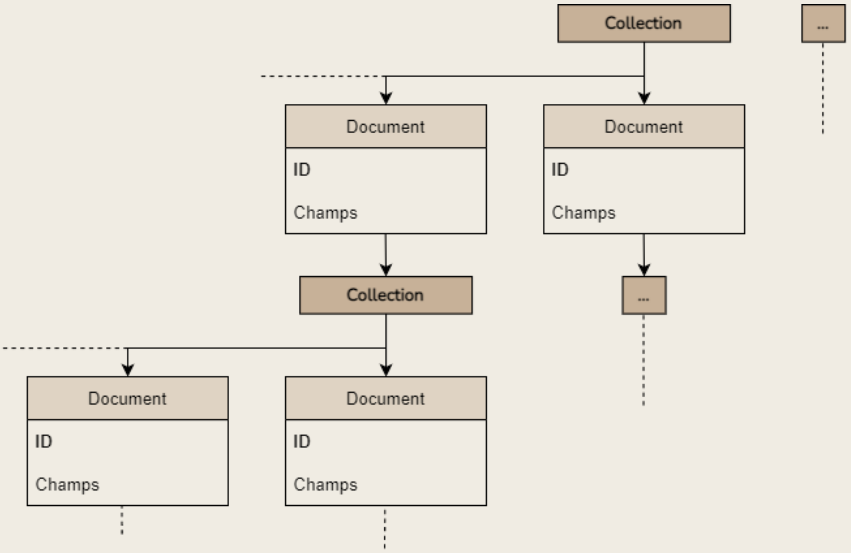
A - Modélisation

Voici un schéma simplifié de la base de données de l'application :



B - Mise en place

La base de données est sous forme de collections qui contiennent des documents qui eux même possède des champs, comme ceci :



Ainsi, pour l'application Petility on a une première collection qui va contenir tous les animaux de l'utilisateur. Alors un document sera attribué à un animal dont les données générales seront enregistrées dans les champs de ce document. Ensuite, les détails de l'animal seront enregistrés avec des sous-collections dans les champs des sous documents.

Donc, voici la fonction pour créer un animal par exemple :

```
const addAnimal = () => {
  if (newname !== "" && newtype !== "" && newrdv !== "") {
    nbAni += 1;
    setDoc(doc(database, uid, nbAni.toString()), {
      name: newname,
      type: newtype,
      rdv: newrdv
    });
    navigation.navigate("Menu");
  } else {
    Alert.alert("Veuillez remplir tous les champs");
  }
};
```

Dans cette fonction, on vérifie d'abord si tous les champs sont bien remplis et on peut alors ajouter 1 à l'index de l'animal (on les récupère précédemment), soit le numéro du document (alors on peut avoir plusieurs animaux d'index 1, 2, 3..).

Ensuite, la fonction setDoc prend en paramètre un document (doc) qui lui prend en paramètre la base de données (*getFirestore()*), l'uid de l'utilisateur concerné, et le nom du document en String.

Puis, on ajoute les données des champs (name, type et rdv). Et on retourne sur le menu principal de l'application.

Le fonctionnement est le même pour l'ajout des détails des animaux (nourriture et poids), mais le paramètre du document dans *setDoc* inclut alors le chemin vers la sous-collection utilisée :

```
const updateAnimal = () => {
  if (addType == "food") {
    nbFood += 1;
    var newModDetails = newDetails.replace(/,/g, ".");
    if (newdate !== null && newModDetails !== null) {
      setDoc(doc(database, uid, id, "food", nbFood.toString()), {
        date: newdate,
        quantity: newModDetails
      });} else {
      Alert.alert("Veuillez remplir les champs !");
    }}
  if (addType == "weight") {
    nbWeight += 1;
    var newModDetails = newDetails.replace(/,/g, ".");
    if (newdate !== null && newModDetails !== null) {
      setDoc(doc(database, uid, id, "weight", nbWeight.toString()), {
        date: newdate,
        weight: newModDetails
      });} else {
      Alert.alert("Veuillez remplir les champs !");
    }}
  navigation.navigate("Détail", { id });
};
```

B - Configuration

C'est dans le fichier *firebase/config.js* dans le dossier *src/firebase* que l'on trouve le fichier de configuration de connexion à la base de données.

```
import { initializeApp } from 'firebase/app';
import { getAuth } from 'firebase/auth';
import { getFirestore } from 'firebase/firestore';

const firebaseConfig = {
    apiKey: "******************",
    authDomain: "******************.firebaseapp.com",
    databaseURL: "https://****************.firebasedatabase.app",
    projectId: "****************",
    storageBucket: "****************.appspot.com",
    messagingSenderId: "*****************",
    appId: "****************",
    measurementId: "*****************"
};

// initialize firebase
initializeApp(firebaseConfig);

export const auth = getAuth();
export const database = getFirestore();
```

C'est dans ce fichier que l'on importe différents module lié à la Firebase tel que l'authentification et Firestore. Ensuite, on entre les identifiants de connexion et on initialise la base de données Firebase.

Et enfin, on exporte les modules (auth et database).

III - Routeur

C'est à la racine du projet que le fichier *App.js* possède le rôle de routeur. En effet, après avoir importé les modules nécessaires, on définit le nom de toutes les pages en fonction de leurs chemins d'accès.

Puis on ajoute les différents stacks (pages) généraux (navigator) et de connexion (auth). Et la gestion de connexion aux pages.

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

import { auth } from './src/firebase/config';

//import des pages
import Menu from './src/screens/HomeScreen';
import Login from './src/screens/CnxScreen/LoginScreen';
[...]

const Stack = createStackNavigator();

//Pages de l'application
function MenuStack() {
 return (
  <Stack.Navigator>
   <Stack.Screen name='Menu' component={Menu} />
          [...]
  </Stack.Navigator>
 );}

//Pages de connexion
function AuthStack() {
 return (
  <Stack.Navigator screenOptions={{ headerShown: false }}>
   <Stack.Screen name='Login' component={Login} />
          [...]
          </Stack.Navigator>
 );}
```

IV - Authentification

La gestion de l'authentification de l'application se fait avec Firebase. On a une page de création de compte, de connexion et de gestion du compte de l'utilisateur connecté. Sur la page de gestion du compte utilisateur, celui-ci peut se déconnecter ou supprimer son compte.

A - Création du compte

Concernant la page de création du compte (RegistrationScreen), c'est après avoir fait les imports nécessaires et faire appel aux hooks (comme useState) que l'on peut ensuite créer les fonctions et écrire le code de la view (affichage de la page).

C'est la fonction *onHandleSignup* qui vérifie que les champs ne sont pas vide après que l'utilisateur clique sur le bouton pour créer un compte. C'est grâce à la fonction *createUserWithEmailAndPassword* qui permet de créer un compte dans la base de données Firebase.

```
const onHandleSignup = () => {
    if (email !== '' && password !== '') {
      var modemail = email.replace(/ /g, "");
      createUserWithEmailAndPassword(auth, modemail, password)
        .then(() => console.log('Signup success'))
        .catch(err => Alert.alert(`${err.replace('Firebase:','')}`));
    }
  };
```

De plus, pour réduire la taille du texte si une erreur se produit, on remplace tous les "Firebase:" avec un seul "Firebase:".

B - Connexion au compte

Le fonctionnement du fichier de connexion au compte (LoginScreen) est le même que pour la création de compte mais avec la fonction *onHandleLogin*.

Cette fonction vérifie également que les champs ne soient pas vides lorsque l'utilisateur clique sur le bouton pour se connecter. Mais ici, on utilise la fonction *signInWithEmailAndPassword* qui prend en paramètre les identifiants pour se connecter à l'application.

```
const onHandleLogin = () => {
  if (email !== '' && password !== '') {
    var modemail = email.replace(/ /g, "");
    signInWithEmailAndPassword(auth, modemail, password)
      .then(() => console.log('Login success'))
      .catch(err => Alert.alert(`Erreur : ${err}`));
  }
};
```

Ici aussi, pour réduire la taille du texte si une erreur se produit, on remplace tous les "Firebase:" avec un seul "Firebase:".

C - Déconnexion du compte

L'utilisateur peut se déconnecter dans sur la page de gestion de son compte (userScreen).

Après avoir cliqué sur le bouton de déconnexion, c'est la fonction *onSignOut* qui est appelée, qui utilise le module de déconnexion de l'authentification de Firebase *signOut*.

```
const onSignOut = () => {
  signOut(auth).catch(error => console.log('Erreur de déconnexion: ', error));
};
```

Alors, après avoir cliqué sur se déconnecter l'utilisateur sera de retour sur la page de connexion et devra entrer de nouveau ses identifiants pour se connecter à l'application.

D - Suppression du compte

Le fonctionnement est similaire pour la suppression du compte. Or, afin de vérifier si ce n'est pas une mégarde de l'utilisateur, on ouvre un menu contextuel afin de vérifier si l'utilisateur souhaite réellement supprimer son compte.

Alors on importe les modules nécessaires :

```
import { Text, TouchableOpacity, Alert, Modal, StyleSheet, Pressable, View } from "react-native";
[...]
```

Puis dans la fonction UserScreen après avoir récupérer les données de l'utilisateur on ajoute les fonctions *onSignOut* et *onDeleteUser* :

```
const onDeleteUser = () => {

  //suppression des animaux
  const unsubscribe = onSnapshot(
    collection(database, uid),
    (querySnapshot) => { querySnapshot.forEach(
        (data) => {
          deleteDoc(doc(database, uid, data.id));
        },
        function (error) { console.log(error); }
      );
    }
  );

  //suppression des détails des animaux de l'utilisateur
 deleteUser(user).then(()=>
 {navigation.navigate("Signup")}).catch(error => console.log('Erreur : ', error));
 return () => unsubscribe();
};
```

On définit les fonctions qui concernent le modal :

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

Puis dans la vue on ajoute le modal :

```
<View style={styles.container}>
      <Modal
      animationType="slide"
      transparent={true}
      visible={modalVisible}
      onRequestClose={() => {
        Alert.alert("Modal has been closed.");
        setModalVisible(!modalVisible);
      }}
    >
      <View style={styles.container}>
        <View style={styles.modalView}>
            <Text style={styles.modalText}>Êtes-vous sur de vouloir supprimer votre compte, vos données enregistrées sur ce compte seront perdues ?</Text>
          <Pressable
            style={[styles.button, styles.modalButtonClose]}
            onPress={() => setModalVisible(!modalVisible)}
          >
            <Text style={styles.textStyle}>Non, c'est une érreur</Text>
          </Pressable>
          <Pressable
            style={[styles.button, styles.modalButtonAccept]}
            onPress={onDeleteUser}
          >
            <Text style={styles.textStyle}>Oui, je veux le supprimer</Text>
          </Pressable>
        </View>
      </View>
    </Modal>
```

Et enfin, on ajoute dans cette même vue le bouton qui ouvre le modal :

```
<TouchableOpacity
      style={styles.buttonDanger}
      onPress={() => setModalVisible(true)}
    >
<Text style={styles.buttonTitle}>Supprimer de compte</Text>
    </TouchableOpacity>
```

Donc, quand l'utilisateur va supprimer son compte (après avoir accepté le pop up), son compte et ses données seront définitivement supprimées. Et il sera de nouveau sur la page de création de compte de l'application.

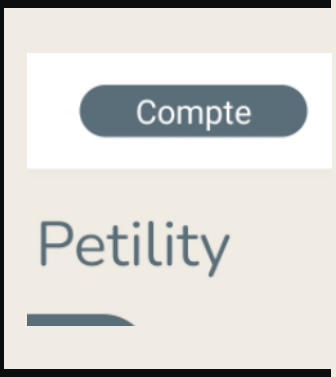
V - Interface

Nous allons voir comment les données sont affichées sur différentes pages tel que sur la page d'accueil et la page du détail avec les graphiques.

A - Page d'accueil

Le code de la page d'accueil est dans le fichier *screens/ HomeScreen.js*. Après avoir fait les imports et déclarer les différentes variables, on peut placer le bouton pour accéder au compte de l'utilisateur en haut à droite :

```
useLayoutEffect(() => {
  navigation.setOptions({
    headerRight: () => (
      <TouchableOpacity
        style={styles.topButton}
        onPress={() => navigation.navigate("Utilisateur")}>
        <Text style={styles.buttonTopTitle}>Compte</Text>
      </TouchableOpacity>
    )});
 }, [navigation]);
```



Ensuite, on affiche le nom de l'application et un gros bouton pour ajouter des animaux et la liste des animaux (FlastList). En plus d'un bouton en bas pour afficher la page d'aide, le tout dans la view :

```
if (!fontsLoaded) {
  return <AppLoading />;
 } else {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Application Petility</Text>

      <TouchableOpacity
        style={styles.buttonHome}
        onPress={() => navigation.navigate("Ajouter")}>
        <Text style={styles.buttonTitle}> Ajouter un animal </Text>
      </TouchableOpacity>
      <FlatList data={ani} renderItem={renderItem} />
      <TouchableOpacity
        style={styles.buttonRound}
        onPress={() => navigation.navigate("Astuces")}>
        <Text style={styles.buttonTitle}>Aide</Text>
      </TouchableOpacity>
    </View>
  );
 }
```

Pour afficher la liste des animaux sur cette page, on récupère d'abord les données de chaques élément depuis la base de données Firestore (id, nom et date de rdv) ;

```
useEffect(() => {
  const unsubscribe = onSnapshot(
    collection(database, uid),
    (querySnapshot) => {
      const ani = [];
      querySnapshot.forEach(
        (doc) => {
```

```
        ani.push({
          id: doc.id,
          name: doc.data().name,
          rdv: doc.data().rdv
        }); },
      function (error) {
        console.log(error);
      } );
    setAni(ani);
    setLoading(false);
  } );
 return () => unsubscribe();
}, []);
```

Puis, on affiche ces données dans des boutons à la suite sur la page d'accueil :

```
function Item({ id, name, rdv }) {
 const handlePress = () => {
  navigation.navigate("Détail", { id });
 };
 return (
  <TouchableOpacity onPress={() => handlePress()}>
   <View style={styles.itemHome}>
    <View style={styles.itemTop}>
     <Text style={styles.name}>{name}</Text>
     <View style={styles.itemDate}>
      <Text style={styles.itemDateTitle}>Prochain RDV le {rdv + "/" + month}</Text>
     </View>
    </View>
   </View>
  </TouchableOpacity>
 );
}
const renderItem = ({ item }) => (
 <Item id={item.id} name={item.name} rdv={item.rdv} />
);
```

Donc, les données seront affichées dans une FlatList composé de boutons avec différentes informations sur chacun d'entre eux pour chaque animal (nom animal et date de rendez-vous).

Et quand l'utilisateur clique sur l'un des boutons, alors il ouvrira la page de détail de l'animal en question.

B - Graphiques

C'est la page de détail des animaux qui contient les deux graphiques qui s'adaptent en fonction des données entrées. On a deux graphiques : nourriture et poids de l'animal. Les graphiques sont issus du kit *react-native-chart-kit* qui utilise *react-native-svg*.

Ici, on va s'occuper de celui du poids mais c'est le même principe pour le graphique de nourriture.

Le code de la page de détail est contenu dans le fichier *screens/ShowScreens.js*, après importer les différents modules dont LineChart et SVG :

```
import { LineChart } from "react-native-chart-kit";
import { Rect, Text as TextSVG, Svg } from "react-native-svg";
```

On peut déclarer les variables dont celles qui vont prendre les coordonnées des points du graphique :

```
let [posWeight, setPosWeight] = useState({
  x: 0,
  y: 0,
  visible: false,
  value: 0
});
```

Après avoir récupérer le nom de l'animal sélectionné par l'utilisateur, on récupère le données relatives au poids de l'animal dans deux variables *aniWeightDate* (pour la date) et *aniWeight* (pour le poids) :

```
useEffect(() => {
  const unsubscribe = onSnapshot(
    collection(database, uid, id, "weight"),
    (querySnapshot) => {
      const aniWeightDate = [];
      const aniWeight = [];
      querySnapshot.forEach(
        (doc) => {
          aniWeightDate.push(doc.data().date);
          aniWeight.push(doc.data().weight);
        },
        function (error) {
          console.log(error);
        } );
      setLoading(false);
      setAniWeight(aniWeight);
      setAniWeightDate(aniWeightDate);
    } );
  return () => unsubscribe();
}, []);
```

Ensuite, on vérifie si on a au moins deux coordonnées de point pour le graphique. En effet, le graphique doit avoir au minimum deux valeurs par défaut :

```
if (aniWeightDate.length >= 2 && aniWeight.length >= 2) {
  var weight = {
    labels: aniWeightDate,
    datasets: [
      {
        data: aniWeight,
        color: (opacity = 1) => `rgba(89, 110, 121, ${opacity})`
      } ] };
```

```
} else {
  var weight = {
    labels: ["Ajouter", "2", "valeurs"],
    datasets: [
      {
        data: ["1", "2", "3"]
      } ] };
}
```

Puis, on définit le style des graphiques :

```
const chartConfig = {
  backgroundGradientFrom: "#DFD3C3",
  backgroundGradientTo: "#DFD3C3",
  color: (opacity = 1) => `rgba(0, 0, 0, ${opacity})`
};
```

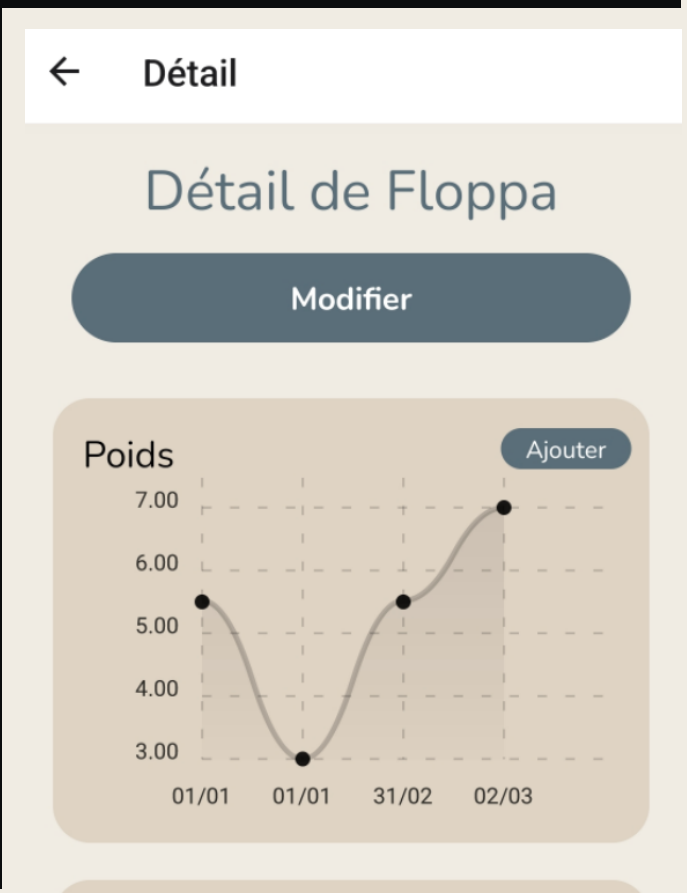
Enfin, dans la vue on affiche le nom de l'animal et un bouton pour éditer ses données générales. Et également les graphiques :

```
if (!fontsLoaded) {
  return <AppLoading />;
} else {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Détail de {PetName}</Text>
      <TouchableOpacity
        style={styles.button}
        onPress={() => navigation.navigate("Modifier", { id })}
      >
        <Text style={styles.buttonTitle}>Modifier</Text>
      </TouchableOpacity>
      <View style={styles.item}>
        <View style={styles.itemTop}>
          <Text style={styles.itemName}>Poids </Text>
          <TouchableOpacity
```

```
        style={styles.itemButtonWeight}
        onPress={() =>
          navigation.navigate("Historique", { id, addType: "weight" })
        }
      >
        <Text style={styles.itemButtonTitle}>Ajouter</Text>
      </TouchableOpacity>
    </View>
    <LineChart
      data={weight}
      width={280}
      height={180}
      chartConfig={chartConfig}
      bezier
      style={{
        fontFamily: "Nunito_400Regular"
      }}
      yAxisSuffix={" kg"}
      decorator={() => {
        return posWeight.visible ? (
          <View>
            <Svg>
              <Rect
                x={posWeight.x - 18}
                y={posWeight.y - 18}
                width="35"
                height="25"
                rx={10}
                fill="#596E79"
              />
              <TextSVG
                x={posWeight.x}
                y={posWeight.y}
                fill="white"
                fontSize="16"
                fontWeight="bold"
```

```
            textAnchor="middle"
          >
            {posWeight.value}
          </TextSVG>
        </Svg>
      </View>
    ) : null;
  }}
  onDataPointClick={(data) => {
    let isSamePoint =
      posWeight.x === data.x && posWeight.y === data.y;

    isSamePoint
      ? setPosWeight((previousState) => {
          return {
            ...previousState,
            value: data.value,
            visible: !previousState.visible
          };
        })
      : setPosWeight({
          x: data.x,
          value: data.value,
          y: data.y,
          visible: true
        });
  }}
 />
 </View>
 </View>
 );
}
```



C’est lors de l'affichage du graphique que l’on définit ses paramètres (comme la position et le taille du graphique).