SmartList

MANUEL DE MAINTENANCE

Contents

1 Introduction

Ce document est destiné aux intervenants dans la maintenantce ou l'amélioration du projet intitulé **SmartList**. Il décrit le fonctionnement technique du code. Les thématiques abordées sont: l'éxécution, la configuration et le build.

2 Commandes de démarrage

2.1 Démarrage de l'interface du serveur expo

Pour démarrer le projet du côté client il faut suivre ces étapes:

- `expo install ou  npm install ou yarn install`
- `expo start ou npm start ou yarn start`

Si le démarrage se fait avec succès, une page web du serveur **Metro Bundler expo** s'affiche sur le navigateur par défaut.

2.2 Build en mode développement

Pour faire un build dans le cadre du développemnt, il faut aller sur la fenêtre de la page web du serveur expo, s'assurer que le mode de production est éteint, puis choisir son mode de build:

- Tunnel ssh
- LAN
- Local

Pour le mode LAN, il est nécessaire que le téléphone soit sur le même réseau que l'ordinateur. Le problème ne se pose pas dans le cas d'une machine virtuelle android.

2.3 Build en mode production

Il y a deux possibilités:

- Cocher le mode production dans l'interface du serveur expo.
- Lancer la commande expo build:android.

La commande expo build:android permet de produire une application téléchargeable sous forme de fichier apk.

3 Back-end

Le fichier **package.json** contient toutes les dépendances à installer pour faire fonctionner le projet. Parmis les principales dépendances sur lesquelles s'appuie notre projet, on trouve firebase, firesore,redux, jest et jsdoc.

3.1 Dépendance firebase

C'est une API qui fournit plusieurs fonctionnalité cloud comme le service d'authentification. Les étapes suivantes sont nécessaires pour la configuration du service:

- expo install firebase
- aller sur le site web de firebase
- créer un compte puis un projet web
- copier la configuration proposée

```
1   const firebaseConfig = {
2     PersistenceEnabled: true,
3     apiKey: "AIzaSyAF_rY_VHwjw_sHV-XTwQtxyrx-L1r1XoE",
4     authDomain: "shoppinglist-6cc7b.firebaseapp.com",
5     projectId: "shoppinglist-6cc7b",
6     storageBucket: "shoppinglist-6cc7b.appspot.com",
7     messagingSenderId: "899980789044",
8     appId: "1:899980789044:web:437a9d1df1252be5abd161",
9     measurementId: "G-F5XJDKGDKF"
10  };
```

- coller la configuration dans le fichier App.js dans le dépôt

3.2 Dépendance firestore

Pour procéder à l'installation de cette dépendance qui gère le stockage dans le cloud, lancez la commande:

```
- expo add @react-native-firebase/firestore
```

Pour accéder à firesore:

```
import firebase from 'firebase'

firebase.firestore()
```

Pour accéder à une collection dans la base de donnée:

```
import firebase from 'firebase'

firebase.firestore().collection("users")
```

Pour accéder à un document dans une collection:

```
import firebase from 'firebase'

firebase.firestore().collection("users").doc(id)
// remplacer par firebase.auth().currentUser.uid pour acceder
     aux donnees de l'utilisateur
```

Pour effectuer un update de la base de donnée:

```
- sélectionner un document
- doc("document").update({state: newState})
```

Pour lire les données depuis le cloud:

```
- sélectionner un document
- appliquer la fonction get()
- effectuer un snapshot (état de la base en un instant)
- effectuer les actions à transmettre au code pour propager les données
```

Exemple de code: ./src/redux/actions/listActions.js

```
firebase
.firestore()
.collection('users')
.doc(firebase.auth().currentUser.uid)
.collection('User')
.doc('user')
.get()
.then((snapshot) => {
  if (snapshot.exists) {
    console.log("dispaaaatching");
    dispatch({ type: 'LOAD_LISTS_CLOUD', payload: snapshot.
        data() });
  }
  else {
    console.log('does not exist');
  }
```

L'update instantané des donées en cas de changement local, faisait ralentire considérablement l'application. Afin de palier à ce problème de performance et pour économiser les flux de lectures/écritures dans la base, le choix de lire les donées uniquement si l'application est en train de charger a été fait. Quand à l'écriture, elle se fait une fois les listes de courses sont modifiées dans le contexte de la page d'accueil. Cela veut dire que l'ajout des produits dans le contenu de la liste de course n'envoie pas de requête au cloud. C'est au moment du retour vers la page d'accueil que la sauvegarde se fait.

La fonction saveToCloud est responsable de l'écriture, tandis que loadFromCloud récupère les données du cloud.

Ce choix signifie qu'un utilisateur ne peut pas être parfaitement synchrone en étant connecté sur deux appareils en même temps. Par contre, en utilisant un seul appareil, l'utilisateur a la garantie que ses données soient synchronisées.

3.3 Dépendance redux

Redux est un module externe qui permet de contrôler le flux et la gestion des informations dans l'application. Celui-ci s'avère important pour communiquer entre les components. Ces 3 dossiers permettent de mettre en place cet outil:

```
- actions
- reducers
- store
```

Le dossier actions permet de définir des actions de type *string* à accomplir puis les transmets dans le canal de *redux*.

Exemple de code:

```
1    addToLists: (newItem) =>
2      dispatch({
3        type: 'ADD_TO_LISTS',
4        payload: newItem,
5      }),
```

Le dossier reducers permet de traduire les actions de type string en fonctions qui finissent par effectuer un update du state du reducer.

Exemple de code:

```
1  export default function listReducer(state = initialState, action) {
2
3    switch (action.type) {
4
5      case 'SET_ID_SELECTED':
6        return {
7          lists: [...state.lists],
8          lastIdSelected: action.payload,
9        }
10
11     case 'ADD_TO_LISTS': {
12       return {
13         lists: [action.payload, ...state.lists],
```

```
        lastIdSelected: state.lastIdSelected

      }
    }
}
```

Afin de connecter le reducer à notre component on doit ajouter le code suivant:

```
import {mapStateToProps, mapDispatchToProps} from '../redux/actions
    /listesActions';

const ListesScreen = (props) => {
...
}
export default connect(mapStateToProps, mapDispatchToProps)
(ListesScreen);
```

Le dossier store permet de réeunir tous les reducers en un seul en utilisant le fichier store.js:

```
import rootReducer from '../reducers/index'

export const store = createStore(rootReducer, applyMiddleware(thunk
    ));
```

Pour la mise en place d'un listener pour redux:

```
store.subscribe(() => saveToLocalStorage(store.getState()));
```

Il permet d'appeler le service de redux à chaque fois qu'une action est propagée. De plus, cette ligne de code permet de sauvegarder localement les changement effectués suite à une action.

Afin de connecter le store à notre application, il faut encapsuler notre navigation dans un component *Provider* propre à *react-redux*.

Dans le fichier App.js:

```
import { Provider } from 'react-redux'
.
.
.
 <Provider store={store}>
    <RootNavigation />
 </Provider>);
```

3.4 Dépendance JsDoc

Pour générer la documentation de notre code, nous avons utilisé JsDoc. La documentation est fournie avec le rendu du projet mais dans le cas de la reprise du code pour le migrer vers un autre projet cela peut être utile.

Installation:

- expo add jsdoc
- ajout du script: "docs": "node_modules/.bin/jsdoc -c jsdoc.json" dans package.json dans la section scripts

- `créeation d'un fichier jsdoc.json et ajout d'une configuration`

Configuration actuelle du fichier jsdoc.json:

```
{
  "source": {
    "include": [
      "src","./App.js"
    ],
    "includePattern": ".js$",
    "excludePattern": "(node_modules/|docs)"
  },
  "plugins": [
    "plugins/markdown",
    "node_modules/better-docs/typescript"
  ],
  "templates": {
    "cleverLinks": true,
    "monospaceLinks": true
  },
  "opts": {
    "recurse": true,
    "template": "node_modules/better-docs",
    "destination": "./docs/",
    "readme": "./README.md"
  }
}
```

Pour compiler la documentation:

- `yarn docs`

Pour afficher la documentation:

- `ouvrir le fichier index.html dans le répertoire docs`

Résultat:

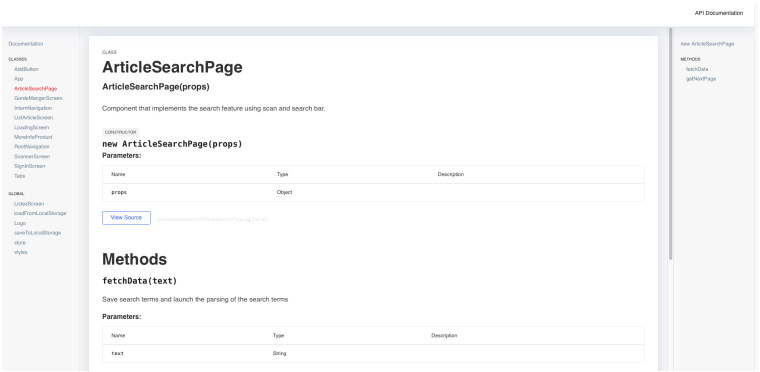

Figure 1: Génération de la documentation avec JsDoc

3.5 Navigation

L'entrée de l'application se fait à travers le fichier App.js. Ensuite on définit une première navigation appelée **RootNavigation** qui nous permet d'assurer une bonne gestion de la connection de l'utilisateur. Cette navigation fait appel à **TabsNavigation** qui s'occupe de garder en mémoire les deux principales fenêtres de l'application (liste et garde-manger). Celle-ci est connecté à **InternNavigation** qui permet de naviguer vers toutes les pages internes. La figure ci-dessous résume l'achitecture de l'application:

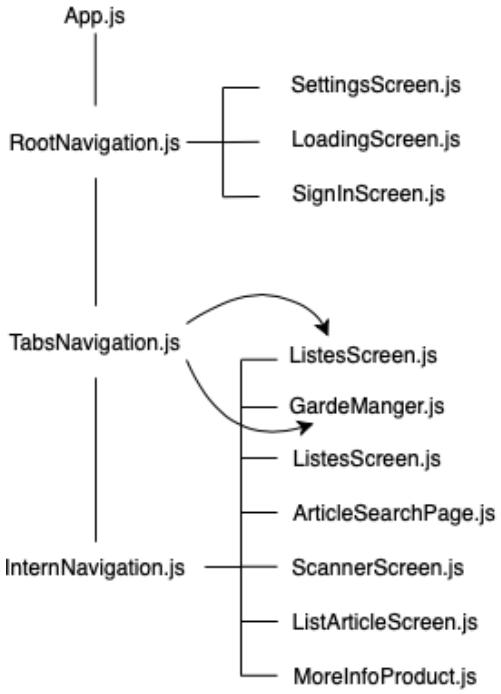

Figure 2: Arbre de navigation du projet

4 Front-end

4.1 Structure du front-end

Le front-end est géré par ces fichiers/dossiers:

- `app.json`
- `components/`
- `screens/`
- `assets/`

4.2 Le fichier app.json

Ce fichier permet de configurer le nom, le logo et la page de chargement. La configuration actuelle est la suivante:

```
{
  "expo": {
    "name": "ShoppingList",
    "slug": "ShoppingList",
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./assets/icon.png",
    "splash": {
      "image": "./src/assets/logo.png",
      "resizeMode": "contain",
      "backgroundColor": "#FFFFFF"
    },
    "updates": {
      "fallbackToCacheTimeout": 0
    },
    "assetBundlePatterns": [
      "**/*"
    ],
    "ios": {
      "supportsTablet": true
    },
    "android": {
      "adaptiveIcon": {
        "foregroundImage": "./assets/icon.png",
        "backgroundColor": "#FFFFFF"
      },
      "package": "com.imd.ShoppingList"
    },
    "web": {
      "favicon": "./assets/icon.png"
    }
  }
}
```

4.3 Dossier components

Ce dossier contient tous les components partagés entre les différents pages.

4.4 Dossier screens

Ce dossier contient tous les components qui forment une page de navigation dans l'application.

4.5 Dossier assets

Ce dossier contient les images utilisées dans le développement.