UP projet - Développer une application mobile avec React Native

Table des matières

I. Initialisation du projet

Durée : 1 h 30

Environnement de travail : avoir installé sur son ordinateur VS Code, Node.js[1], Expo et des simulateurs de téléphone mobile.

Pré-requis : avoir les bases de JavaScript nécessaires à la compréhension de React et React Native, savoir lancer un projet React Native avec expo, et une première expérience avec react-navigation[2] serait un plus.

Contexte

Vous en avez très probablement déjà utilisé sur papier ou sous forme d'application, la to-do list est un classique de l'apprentissage de tout-e développeur-euse.

Nous allons dans ce projet guidé coder notre propre application de to-do list en React Native. Ce projet sera l'occasion de voir comment il est possible d'avoir des données persistantes dans une application en utilisant le stockage local des téléphones mobiles.

Objectifs

Initialiser l'application avec expo

Contexte

Expo est une plateforme de développement permettant de créer des applications mobiles cross-platform en React Native. Même si son usage n'est pas recommandé dans certains projets ayant des fonctionnalités particulières, il sera plus que suffisant pour l'application que l'on va dans ce cours.

La première commande à exécuter est « *expo init [project_name]* ». Expo va nous proposer plusieurs templates. Nous choisirons le managed workflow en JavaScript.

Une fois les packages installés, lancer la commande « *npm start* » ou « *yarn start* » pour lancer le projet, puis il suffit d'ouvrir un navigateur.

Pour vérifier que le serveur de développement fonctionne bien, nous pouvons modifier le texte indiqué dans le fichier « *App.js* » et voir si les changements sont bien effectifs sur notre simulateur.

Complément

Expo[3]

II. Structure du projet et babel-plugin-module-resolver

Objectifs

- Créer la structure de base des dossiers de notre application
- Installer et paramétrer babel-plugin-module-resolver

---

1 https://nodejs.org/en/
2 https://reactnavigation.org/
3 https://expo.io/

Contexte

Comme dans tout projet informatique, la structure est cruciale. Afin de ne pas se perdre dans les imports de fichiers lorsque l'on travaille avec des dossiers complexes, il est préférable d'utiliser des moyens de configurer ses chemins. Nous utiliserons pour cela babel-plugin-module-resolver.

Dans cette vidéo je vous présenterai la structure de base que je mets en place de chacun de mes projets.

Expo utilisant déjà babel, il y a déjà un fichier de configuration dans notre dossier : « *babel.config.js* ». Une fois le package babel-plugin-module-resolver, il ne nous reste plus qu'à éditer ce fichier pour paramétrer nos routes.

Exemple **Code**

Configuration babel

```
module.exports = function (api) {
  api.cache(true);
  return {
    presets: ["babel-preset-expo"],
    plugins: [
      [
        "module-resolver",
        {
          root: ["."],
          alias: {
            "@components": "./app/components",
            "@hooks": "./app/hooks",
            "@screens": "./app/screens",
            "@navigation": "./app/navigation",
          },
        },
      ],
    ],
  };
};
```

Complément

NPM : babel-plugin-module-resolver[1]

III. Mise en place de la navigation

Objectifs

- Installer react-navigation
- Créer nos pages « *Home.js* » et « *ToDoCreate.js* »
- Mettre en place la navigation de base

Contexte

Comme vous avez dû le voir, la navigation sur un projet React Native diffère d'un projet web. Nous allons devoir mettre en place une ou plusieurs stacks de navigation qui donneront accès aux différents écrans de l'application.

Dans un contexte réel, on aura souvent plusieurs niveaux de profondeur de stack. Dans le cas de ce projet guidé, nous allons garder les choses simples avec un seul niveau et deux écrans.

Tout d'abord, nous allons installer le package « *react-navigation* » dans notre projet. Il y a un certain nombre de dépendances à installer pour que le package fonctionne bien. Etant donné que nous sommes sur un projet Expo, il faut bien penser à les installer avec la commande « *expo install [noms des packages]* ».

Pour le moment, dans notre dossier « *screens* », nous allons créer deux composants « *minimalistes* » pour la « *Home* » et la « *ToDoCreate* ». Nous leur apporterons du contenu plus tard.

Nous allons, après cela, pouvoir créer la Navigation dans le dossier dédié et l'importer au niveau de notre App.js. À la base de ce fichier, nous allons devoir mettre un composant « *NavigationContainer* ».

Le navigateur que nous allons utiliser dans ce projet va être créé par le « *createStackNavigator* ».

Une fois le navigateur créé, nous allons pouvoir placer notre Router et nos Screens auxquels nous passerons les composants préparés plus tôt. Nous allons vouloir que notre navigation soit en mode « *modal* », c'est-à-dire que l'écran d'ajout de to-do devra apparaître depuis le bas.

Chacun des composants « *screen* » recevra automatiquement une prop « *navigation* ». Cet élément expose une méthode « *navigate* » (entre autres) qui nous permet de passer d'un écran à l'autre.

1 https://www.npmjs.com/package/babel-plugin-module-resolver

Exemple **Code**

Navigation

```
import React from "react";
import { NavigationContainer } from "@react-navigation/native";
import { createStackNavigator } from "@react-navigation/stack";

import Home from "@screens/Home";
import ToDoCreate from "@screens/ToDoCreate";

export default function Navigation() {
  const RootStack = createStackNavigator();

  return (
    <NavigationContainer>
      <RootStack.Navigator mode="modal">
        <RootStack.Screen
          name="Home"
          component={Home}
          options={{ headerShown: false }}
        />
        <RootStack.Screen
          name="ToDoCreate"
          component={ToDoCreate}
          options={{ headerShown: false }}
        />
      </RootStack.Navigator>
    </NavigationContainer>
  );
}

```

Complément

React navigation[1]

1 https://reactnavigation.org/

IV. Composant UI : le bouton

Objectifs

- Créer le composant Button
- L'appliquer au bouton de navigation

Contexte

L'intérêt d'utiliser des technologies comme React ou React Native est de pouvoir réutiliser son code. Dans tous mes projets, je dédie toujours un dossier « *UI* » à l'intérieur du dossier « *components* » où je place tous mes composants généraux, c'est-à-dire ceux qui sont amenés à être réutilisés dans plusieurs pages.

Afin d'être conforme avec les standards du développement mobile, nous allons nous baser sur un kit UI iOS disponible gratuitement dans la partie communauté de Figma (excellente source de designs et d'inspiration).

En nous basant sur un des boutons du kit UI iOS, nous allons voir dans cette vidéo comment créer notre propre composant Button.

Complément

Fichier figma (gratuit)[1]

V. Composant UI : l'input

Objectifs

- Créer le composant Button
- L'appliquer au bouton de navigation

Contexte

Même contexte que l'élément précédent.

En nous basant sur un des boutons du kit UI iOS, nous allons voir dans cette vidéo comment créer notre propre composant Input.

VI. Contrôler le formulaire avec un hook personnalisé

Objectifs

- Créer dans un hook dédié les states relatifs à notre formulaire
- Passer ce state à notre formulaire
- Créer une fonction pour gérer la soumission du formulaire et la passer à ce dernier

1 https://www.figma.com/community/file/809487622678629513

Contexte

Les Hooks React permettent de séparer toute la logique de nos composants visuels. Nous ne sommes plus contraints d'utiliser des composants de type Class pour pouvoir utiliser des choses comme le state.

Pour notre popup de création de tâche, nous allons donc créer un hook dédié dont la fonction sera de gérer les valeurs entrées et l'action de soumission du formulaire.

Dans un nouveau fichier dans le dossier « *hooks* », importer React, et déclarer 2 états grâce au hook React.useState() : le titre et la description de notre tâche.

Puis, plutôt que de les exporter un par un et de les passer manuellement à nos composants Input, nous verrons qu'il est plus simple de créer un tableau « *fields* » dans notre hook.

Chaque élément de ce tableau sera dédié à un champ différent, avec toutes les informations nécessaires pour s'interfacer avec nos éléments visuels.

Étant donné que l'on a maintenant un tableau, il ne nous reste qu'à utiliser la méthode .map() pour pouvoir rendre un Input pour chacun des éléments.

Exemple **Code**

Hook pour gérer un formulaire

```
import React from "react";

export default function useNewToDoManage() {
  const [title, titleChange] = React.useState("");
  const [description, descriptionChange] = React.useState("");

  const fieldsTab = [
    {
      value: title,
      onChange: (e) => titleChange(e),
      placeholder: "Name of the task to do",
    },
    {
      value: description,
      onChange: (e) => descriptionChange(e),
      placeholder: "Description (optionnal)",
    },
  ];

  async function handleSubmit() {
    if (title.length === 0) {
      return alert("The task requires a  title");
    }
  }

  return { fieldsTab, handleSubmit };
}

```

Complément

Vidéo d'introduction des React hooks[1]

VII. Interagir avec le AsyncStorage : Read et Create

1 https://www.youtube.com/watch?v=dpw9EHDh2bM

Objectifs

- Installer le package AsyncStorage
- Créer un hook dédié à la gestion du stockage
- Créer les actions de lecture et de création dans la base de données locale

Contexte

Afin qu'à l'allumage de notre application on puisse retrouver des tâches précédemment créées, nous allons devoir mettre en place un système de stockage.

Étant donné que notre besoin est relativement simple, nous allons pouvoir utiliser la librairie AsyncStorage pour enregistrer au niveau des téléphones mobiles de nos utilisateur-trices les données.

Pour ajouter le package à notre projet expo, nous devons utiliser la commande « *expo install @react-native-async-storage/async-storage* ». Cette librairie fonctionne de manière classique sur le modèle .set(@key, value), .get(key).

Dans notre AsyncStorage, nous n'allons pouvoir stocker que des valeurs sous forme de chaînes de caractère. Or, nous voulons y stocker des objets. Pour cela, nous allons devoir convertir en chaîne nos données en entrée avec JSON.stringify() et les convertir en objets en sortie avec JSON.parse().

Pour pouvoir charger les données stockées dans notre petite BDD à l'ouverture de notre app, nous allons devoir utiliser le hook React useEffect(). Il chargera les données initiales et les stockera dans un state.

Puis, nous allons écrire notre action de création de task. Pour cela, il s'agira de pousser dans notre tableau la nouvelle valeur, puis de charger le nouveau tableau créé dans notre state et notre AsyncStorage.

Complément

Expo : AsyncStorage[1]

VIII. Interagir avec le AsyncStorage : Update et Delete

Objectifs

Créer les fonctions Update et Delete

Contexte

Les 4 opérations principales que l'on va effectuer sur une base de données se résument à l'acronyme CRUD : Create, Read, Update, et Delete.

Dans la vidéo précédente, nous avons vu ensemble les actions React et Create. Pour complémenter notre hook, il s'agira de mettre en place l'update et le delete.

Cette vidéo est en continuité directe de la précédente. Nous allons créer deux nouvelles fonctions pour modifier et supprimer nos éléments.

Dans ces deux cas, nous allons avoir besoin de récupérer l'index des éléments que nous voulons modifier/supprimer, c'est-à-dire leur position dans le tableau.

Puis, une fois que l'on a identifié l'élément en question, il faut utiliser la méthode de tableau .splice().

1 https://docs.expo.io/versions/latest/sdk/async-storage/

Complément

W 3 schools : JavaScript Array splice() Method[1]

IX. Créer des tasks et afficher la liste des to-do

Objectifs

- Créer une task à la soumission de notre formulaire
- Afficher notre liste de to-do

Contexte

Maintenant que nous avons nos fonctions pour interagir avec notre petite base de données locale. Il ne nous reste plus qu'à la brancher à notre formulaire pour commencer à ajouter des tâches à notre liste.

Nous avions dans notre précédente vidéo créé notre hook pour gérer notre formulaire. Il va falloir lui passer notre hook de gestion de l'AsyncStorage, et plus spécifiquement la fonction de création d'une task.

Dans notre hook de gestion de formulaire, nous avions créé une fonction pour gérer la soumission du formulaire. C'est à cet endroit qu'il faut maintenant créer nos tâches. Pour cela, il faut passer à notre fonction de création un objet ayant les propriétés « *title* » et « *description* ». Une fois la fonction appelée, nous pouvons maintenant mettre à zéro nos valeurs « *title* » et « *description* ».

Une fois nos premières tâches créées, il va falloir les afficher. Pour cela, nous allons appeler le hook de gestion de l'AsyncStorage dans l'écran d'accueil, et cette fois-ci nous allons importer dans un premier temps la liste des tâches. Cette liste étant un tableau, il suffit d'utiliser la méthode .map() pour afficher la to-do list.

X. Créer un state global

Objectifs

- Créer un state global grâce à l'API Context
- Afficher notre liste de to-do

Contexte

À la fin de la vidéo précédente, nous avons rencontré un problème... En effet, à la création d'une nouvelle tâche, la liste ne s'actualisait pas. C'est parce que nous appelions à deux endroits différents notre hook de gestion de l'AsyncStorage, et que ce dernier a un state local.

Afin de pouvoir partager de l'information partout dans une application, il est souvent indispensable de mettre en place un state global. Une des solutions très populaire est Redux. Cependant, il n'est pas nécessaire d'installer une librairie pour mettre en place un state global, car React propose une API Context très puissante permettant de se passer de Redux.

1 https://www.w3schools.com/jsref/jsref_splice.asp

Tout d'abord, il va falloir créer un état initial. Dans notre cas, nous allons seulement y initialiser un tableau vide pour notre liste de tâches.

Puis nous allons créer un store avec la création de contexte fournie par React. De ce store, nous allons pouvoir extraire un composant Provider qui nous permettra d'englober toute notre application et de lui passer ce state général.

`Context` va nous permettre de partager des données entre nos composants ; mais pas seulement entre parents et enfants.À partir d'un composant Provider, nous allons pouvoir créer un "contexte" de données que tous les sous-composants (pas uniquement les enfants directs) vont pouvoir utiliser.

Sans l'utilisation de l'API `Context`, les informations sont passées de parents à enfants. Avec le composant Provider, on va pouvoir consommer la valeur définie avec value={} dans le composant App et faire redescendre les informations dans les composants utilisant le contexte

Le composant AppStateProvider que nous allons créer va justement rendre cet élément auquel nous passerons les fonctions d'un reducer. Une fonction reducer permet d'interagir avec un état qu'elle stocke (qu'on doit initialiser) via des actions. Le hook useReducer() retourne deux éléments : le state qu'elle gère, et la fonction dispatch qui permet de passer des actions à ce state. Nous passerons ce state et ce dispatch au Provider fourni par notre création de contexte afin que, partout dans l'application, on puisse les utiliser.

Puis, il faudra passer ce composant AppStateProvider à notre application générale au niveau de « *App.js* ». Par soucis de praticité, nous allons créer un hook « *useAppContext.js* » qui permettra de directement pouvoir accéder aux éléments state et dispatch.

Finalement, nous verrons comment utiliser cela au sein de notre hook de gestion de l'AsyncStorage, en passant d'un state local à un state global.

Exemple **Code**

Création du contexte

```
1  import React, { createContext, useReducer } from "react";
2
3  const initialState = { toDoList: [] };
4
5  const store = createContext(initialState);
6  const { Provider } = store;
7
8  function AppStateProvider({ children }) {
9    const [state, dispatch] = useReducer((prevState, action) => {
10     switch (action.type) {
11       case "TO_DO_LIST_CHANGE":
12         return { ...prevState, toDoList: action.toDoList };
13       default:
14         return prevState;
15     }
16   }, initialState);
17   return <Provider value={{ state, dispatch }}>{children}</Provider>;
18 }
19
20 export { store, AppStateProvider };
21
```

Hook de gestion de l'AsyncStorage utilisant le state global

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

import useAppContext from "./useAppContext";

export default function useAsyncStorageCRUD() {
  const {
    state: { toDoList },
    dispatch,
  } = useAppContext();

  function toDoListChange(newList) {
    dispatch({ type: "TO_DO_LIST_CHANGE", toDoList: newList });
  }

  React.useEffect(() => {
    AsyncStorage.getItem("todolist").then((savedToDoList) => {
      if (savedToDoList) {
        let parsedList = JSON.parse(savedToDoList);
        toDoListChange(parsedList);
      }
    });
  }, []);

  async function toDoCreate(toDo) {
    const newToDoList = [...toDoList];
    newToDoList.push(toDo);
    const jsonValue = JSON.stringify(newToDoList);
    await AsyncStorage.setItem("todolist", jsonValue);
    toDoListChange(newToDoList);
  }

  async function toDoUpdate(index, toDo) {
    const newToDoList = [...toDoList];
    newToDoList.splice(index, 1, toDo);
    const jsonValue = JSON.stringify(newToDoList);
    await AsyncStorage.setItem("todolist", jsonValue);
    toDoListChange(newToDoList);
  }

  async function toDoDelete(index) {
    const newToDoList = [...toDoList];
    newToDoList.splice(index, 1);
    const jsonValue = JSON.stringify(newToDoList);
    await AsyncStorage.setItem("todolist", jsonValue);
    toDoListChange(newToDoList);
  }

  return { toDoList, toDoCreate, toDoUpdate, toDoDelete };
}

```

Complément

React : context[1]

Log rocket[2]

XI. Style des cartes « to-do »

Objectifs

Styliser les cartes de tâches

Contexte

Nous avons maintenant bien notre liste qui s'actualise lorsque nous ajoutons de nouveaux éléments. Il s'agira maintenant de donner un peu de style à nos éléments.

Dans cette vidéo, nous styliserons nos cartes de tâches, en nous inspirant encore une fois du fichier Figma de guidelines UI iOS.

XII. Supprimer une to-do

Objectifs

- Créer le bouton de suppression par carte
- Passer la fonction de suppression à nos éléments

Contexte

Afin que la liste de tâches ne s'encombre pas, il va falloir permettre aux utilisateurs-trices de supprimer des tâches. Cela tombe bien, car nous avons déjà préparé la fonction pour effectuer cette action.

L'idée ici est de passer notre fonction de suppression à notre composant. En premier lieu, il faut créer un bouton au niveau de chacune de nos cartes.

Comme nous l'avions vu plus tôt, notre fonction de suppression prend en paramètres la position de l'élément que l'on veut supprimer dans le tableau. Pour cela, nous pouvons carrément créer la fonction de gestion de la suppression à l'intérieur de la méthode .map().

On appellera ici la fonction de suppression de la BDD exposée par le hook « *useAsyncStorageCRUD* » en lui passant en paramètres l'index qui nous est fourni par la méthode .map().

XIII. Cocher / décocher une task

1 https://reactjs.org/docs/context.html
2 https://blog.logrocket.com/guide-to-react-usereducer-hook/

Objectifs

- Créer une checkbox dont le contenu change selon une prop
- Passer la propriété « *checked* » aux tâches au moment de leur création
- Lui passer une fonction pour pouvoir cocher / décocher les tâches

Contexte

Sans avoir à supprimer des éléments, nous voulons donner la possibilité d'indiquer qu'une tâche est réalisée. Pour cela, nous allons créer une case à cocher au sein de chaque carte et la faire interagir avec notre base de données.

Dans un premier temps, nous allons créer l'élément visuel. Pour cela, nous allons voir comment utiliser des icônes vectorielles exportées par la bibliothèque Ionicons. L'affichage de l'icône « *cochée* » à l'intérieur de notre case devra être conditionné à une prop.

Pour cela, nous allons devoir ajouter une propriété aux tâches.

Finalement, il nous reste seulement à créer une fonction de mise à jour de nos éléments. Nous avions déjà créé une fonction « *toDoUpdate* ». Nous devons l'importer du hook « *useAsyncStorageCRUD* » et la passer à l'intérieur de notre liste pour pouvoir la déclencher lorsque l'on appuie sur notre case.

Complément

Icons[1]

XIV. Touche de style finale

Objectifs

- Rendre notre conteneur scrollable
- Augmenter les marges de l'élément « *button* »
- Ajouter des titres
- Changer la couleur de fond de l'écran de création de tâche

Contexte

Maintenant que la partie fonctionnelle de notre application est prête, je souhaiterais que l'on passe un peu de temps pour améliorer l'apparence générale de notre projet pour conclure.

Au-delà de la partie esthétique, il faut commencer par se demander ce qui se passera s'il y a beaucoup de tâches dans notre liste... Elles dépasseraient sans doute les limites de notre écran. Pour résoudre ce problème, nous allons mettre en plus une ScrollView sur notre conteneur.

Puis, nous améliorerons le style de notre application avec quelques petits détails qui feront la différence, et ajouterons des titres à nos pages.

XV. Pour aller plus loin...

1 https://icons.expo.fyi/

Nous arrivons à la fin de ce projet. Nous avons maintenant une petite app permettant de créer, modifier, et supprimer des tâches. Vous pouvez dès maintenant l'utiliser dans votre vie personnelle.

Le projet que nous avons réalisé n'est que la première étape de ce qui pourrait être une application de productivité plus poussée... Libre à votre motivation et à votre créativité pour aller plus loin !

Pistes pour aller plus loin :

- **Classifier les tâches** : donner des paramètres supplémentaires aux tâches, comme des tags par exemple, et changer la couleur des cartes selon ces tags.
- **Créer une fonctionnalité de Pomodoro** : il s'agirait de déclencher un minuteur pour réaliser les tâches en s'inspirant de la méthode de travail Pomodoro[1].

1 https://en.wikipedia.org/wiki/Pomodoro_Technique