

React Native

<https://reactnative.dev/>

01

Introduction

Contexte hybride et framework
cross-plateformes

Développement hybride VS natif

A decorative line with circles starts at the top right, goes left, then down to a larger circle, then up and right to another circle, and finally up and right to a third circle. Ellipses are placed at the end of the line segments.

Hybride (+)

- Portabilité cross-platform : un seul code pour plusieurs store (app store, google play)
 - Coût de développement : plus rapide, profiles dev plus fréquents (javascript vs swift et kotlin)
 - Montée en compétence plus rapide
 - Maintenabilité et flow de mise en production
- 
- A decorative line with circles starts from the left, goes right to a larger circle, then right to another circle, and finally right to a third circle. Ellipses are placed at the beginning of the line.

Natif (+)

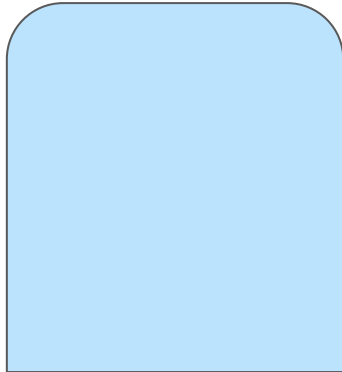
- Meilleures performances : un framework cross-platform n'aura pas les mêmes performances qu'un langage natif.
- Fonctionnalités du téléphone accessible directement
- Meilleure expérience utilisateur (animations, nav, ...)
- Sécurité du langage : ce sont des langages créés par Apple / Android et sans doute plus fiables dans le temps

Utilisation des frameworks cross-plateformes 2019 - 2020



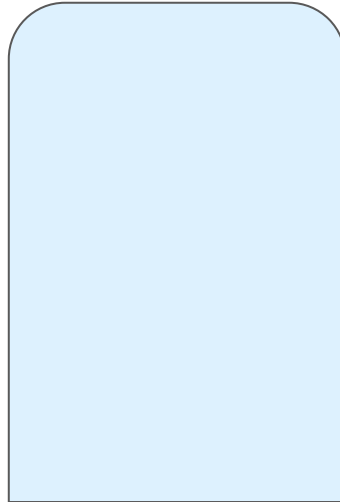
Flutter

2019 - 30%
2020 - 39%



React Native

2019 - 42%
2020 - 42%



Cordova

2019 - 29%
2020 - 18%



Comparatif des frameworks


	Rendering	Gratuit	Communauté	Montée en compétence
React Native	Natif	✓	Forte présence	Moyenne
Flutter	Natif	✓	Communauté grandissante	Difficile
Ionic	Non natif	✓	En baisse	Moyenne
Titanium	Non natif	✗	Basse	Difficile



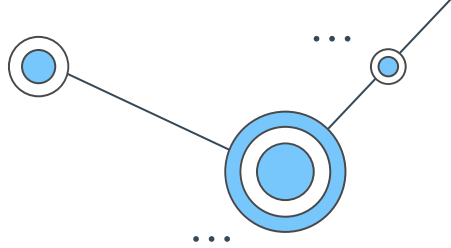
02

React Native

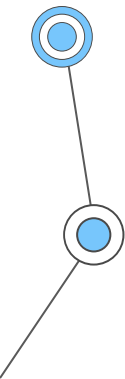
Fonctionnement et principe de base



Qu'est-ce que React Native ?



1. Framework qui permet de créer des applications mobiles **cross-plateformes**.
2. **Open source**, gratuit, créé par Facebook en 2015.
3. Basé sur **React**, et donc **Javascript**.
4. Entreprises connues du paysage tech utilisant React Native dans leurs projets : Facebook, Instagram, Discord, Skype, Pinterest, Tesla, Walmart, Airbnb...



Comment fonctionne React Native ?

01
...

React

Votre code source React.

02
...

Javascript

Il est ensuite interprété,
comme le web, en js.

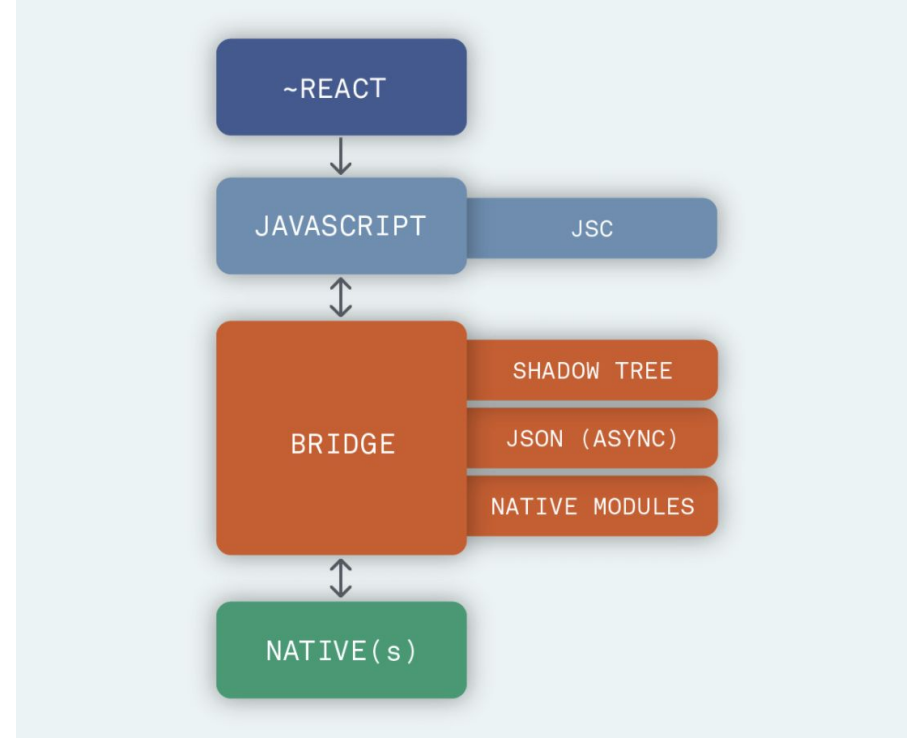
03
...

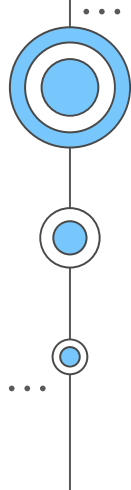
Bridge

Groupe de tâches asynchrones
pour communiquer entre le js
et le natif via JSON.

04
...

Le natif

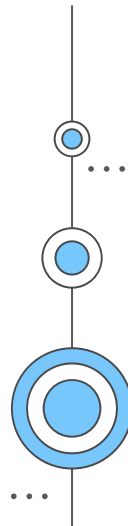




03

Getting Started

Environnement, Pré-requis et options
de développement.





Choix de la méthode



01

Expo CLI

<https://expo.io/tools>

Expo est une suite d'outils qui va vous faciliter la vie durant tout le développement de votre application React Native, ainsi que durant sa mise en production et son maintien.

Prérequis

- Node (> v12)

02

React Native CLI

<https://reactnative.dev/docs>

React Native CLI est utile si vous souhaitez, en plus du code React, créer vous-même vos propres plugins natifs.

Prérequis

- Node (> v12)
- Xcode si macOS, ou Android studio (macOS, Windows, Linux)

Set up expo

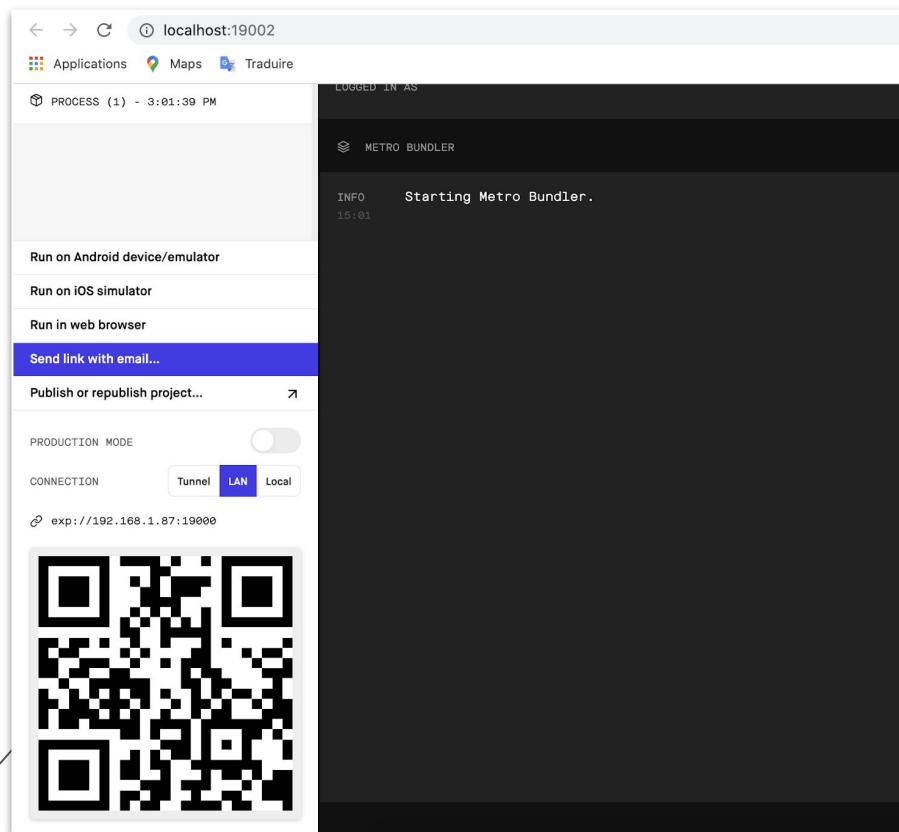
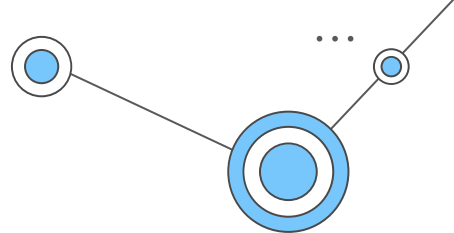
Nous utiliserons donc expo au cours de cette formation. C'est très simple, une fois que node js est installé, il vous suffit de setup expo de manière globale :

```
$ npm install -g expo-cli
```

Vous aurez ensuite accès aux commandes “expo” depuis votre poste. Nous allons donc pouvoir créer notre première application React Native très rapidement via le terminal

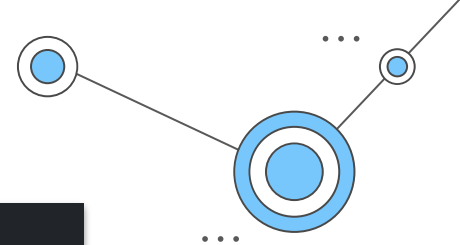
```
$ cd www  
// Commande pour init un nouveau projet  
$ expo init pocReactNative  
// Commande pour lancer et expo et l'environnement de développement  
$ expo start
```

L'interface expo et ses options

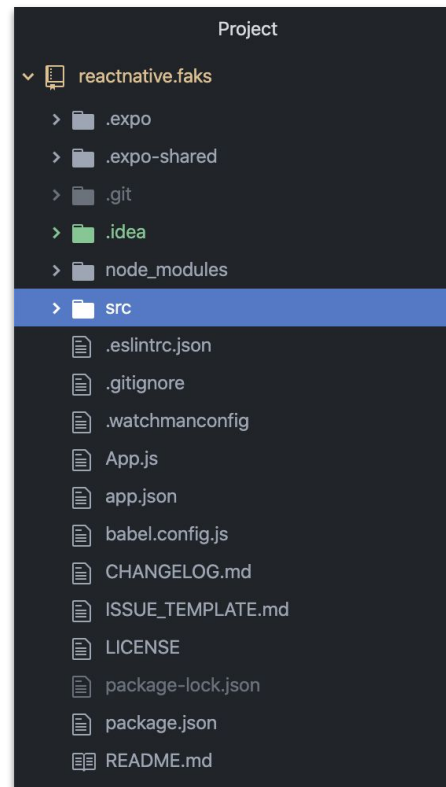
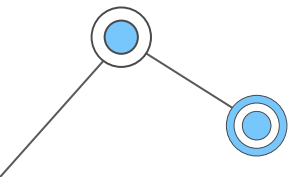


1. **Run on Android device / emulator**
Téléchargez un émulateur pour tester votre App.
2. **Run on iOS simulator**
Téléchargez un émulateur pour tester votre App.
3. **Run in web browser**
Pas d'intérêt dans notre cadre.
4. **Send link**
Partagez votre projet avec votre client / Project manager.
5. **Publish project**
Poussez une nouvelle release si votre app est déjà sur le store.
6. **Production mode**
7. **QR code**
Flashez le code pour tester votre app sur real device

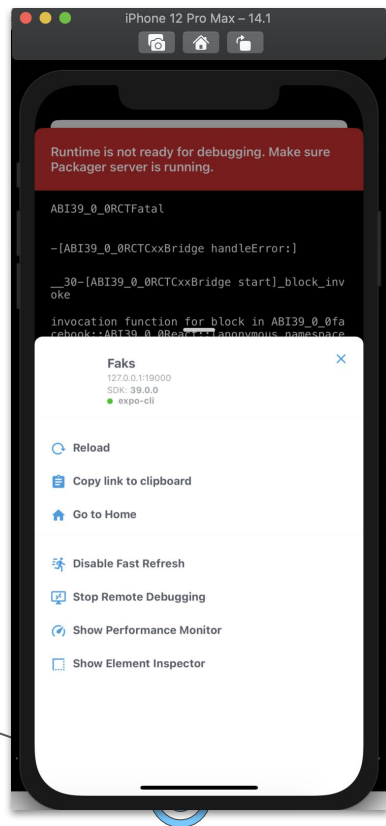
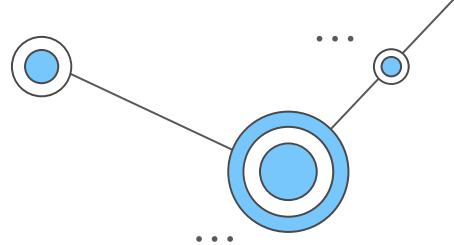
L'architecture générée par expo



1. **src/**
Le code source de votre application.
2. **node_modules/**
Les plugins nodes installées via npm install.
3. **App.js**
Le premier component appelé, il est la racine de votre app.
4. **app.json**
Fichier contenant la configuration de votre application (nom, version d'expo, version de code, ...).
5. **package.json**
Contient la liste des dépendances de votre app, la version souhaitée, incluant react-native lui-même.



Le simulateur / le device



On ne va pas tout passer en revue ici, je vous laisserai découvrir le simulateur - si vous ne le connaissez pas déjà - au long de vos développements.

Simplement, sachez qu'il existe deux trois options qui vont être utiles :

- Remote debugging, qui va permettre d'ouvrir un onglet chrome pour ouvrir la console et ainsi voir les console.log de votre codes et vos erreurs / warnings.
- Fast refresh, qui va refresh l'app automatiquement à chaque fois que vous sauvegardez votre code.
- Show element inspector, un peu moins utile, mais qui peut vous débloquer si vous êtes en difficulté sur le développement de votre interface.

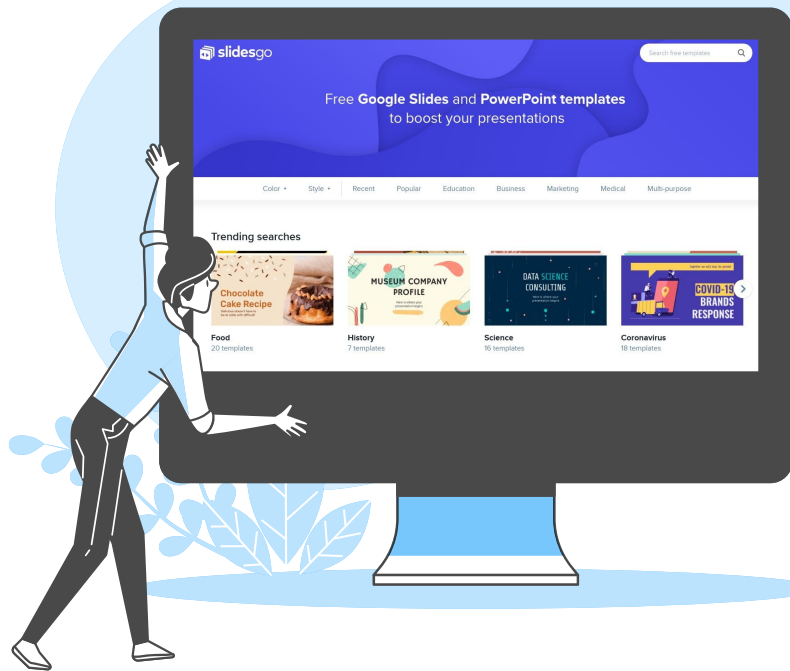
04

React.js

Les principes de bases utilisés dans
React Native (composants, props et
state)

Le premier principe de React

“ Une bibliothèque
JavaScript pour créer des
interfaces utilisateurs à
base de composants
autonomes ”

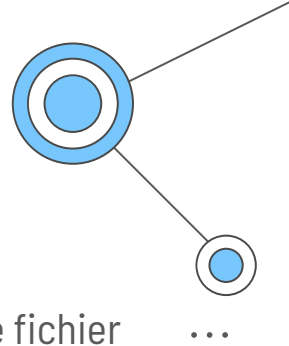




Structure d'un composant simple

(1/3) Les imports

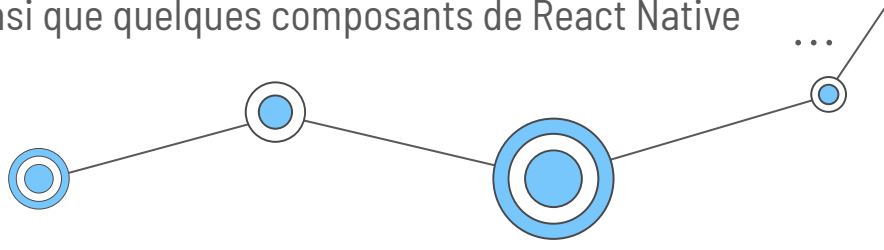
Ouvrez votre fichier App.js, à la racine de votre code. Tout sous React est composant, votre fichier App, qui est le point d'entrée de votre application, n'échappe pas à cette règle.



App.js

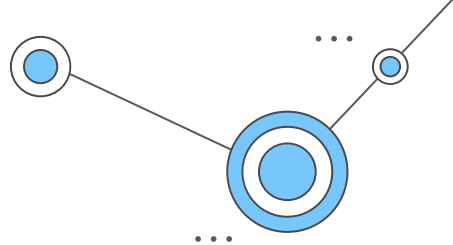
```
1 import { StatusBar } from 'expo-status-bar';
2 import React from 'react';
3 import { StyleSheet, Text, View } from 'react-native';
4
```

La première partie d'un composant sont les imports des dépendances, ou fonctions dont il a besoin pour être généré. De base on retrouve React, ainsi que quelques composants de React Native tels que **Text** et **View**.



Structure d'un composant simple

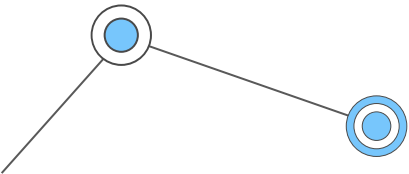
(1 / 3) Les imports



```
2 import React from 'react';  
3 import { StyleSheet, Text, View } from 'react-native';
```

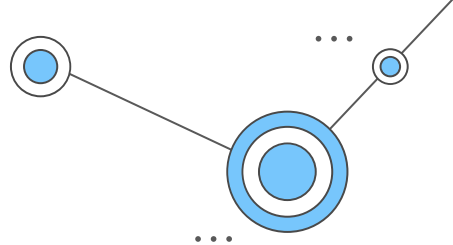
Il est important de noter qu'il existe plusieurs types d'imports. Les “**{ imports }**” et les “**imports**”. Concrètement, il n'y a pas de différence, **cela vient simplement de la manière dont sont exportés ces éléments**.

En effet, via les imports/exports, on peut partager toutes sortes de choses en JavaScript : une librairie entière, une simple variable, un composant, ... Certains vont être exportés via le mot clé **default**, d'autres non, il vous faut alors adapter l'import également. Nous verrons cette notion un peu plus en détail plus tard, quoi qu'il en soit, si vous faites une erreur sur la manière d'importer une librairie, vous le verrez vite, donc pas d'inquiétudes.



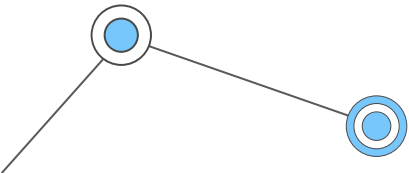
Structure d'un composant simple

(2 / 3) Le corp du composant : en JSX



```
4 |
5 | export default function App() {
6 |   return (
7 |     <View style={styles.container}>
8 |       <Text>Open up App.js to start working on your app!</Text>
9 |       <StatusBar style="auto" />
10 |     </View>
11 |   );
12 | }
13 |
```

Cette syntaxe est du **JSX**, c'est une **extension de JavaScript** créé par Facebook, inspiré du HTML & XML. Vous l'aurez compris, **cela reste donc du js**, ce qui signifie que vous pouvez utiliser du js pure dans un composant si vous le souhaitez. Concernant le JSX, **il sera lui même compilé en objets JavaScript**.



Structure d'un composant simple

(2 / 3) Le corp du composant : en JSX

```
<MyComponent />
```

Une balise JSX commence toujours par une majuscule

```
<MyComponent  
  text="toto" />
```

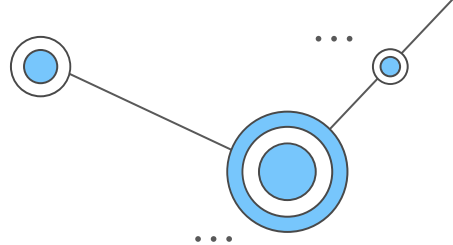
Ici, text est ce que l'on appelle une **propriété (= props)**, qui est une notion fondamentale permettant de personnaliser un composant ou lui faire passer des données.

```
<MyComponent  
  text="toto"  
  onPress={}  
/>
```

Une props peut très bien être une valeur, mais aussi une fonction, ou un évènement.

Structure d'un composant simple

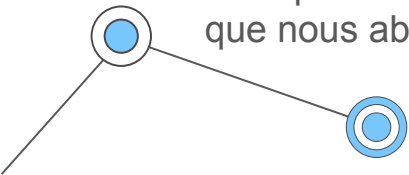
(3 / 3) Styles et exports



```
14  const styles = StyleSheet.create({
15    container: {
16      flex: 1,
17      backgroundColor: '#fff',
18      alignItems: 'center',
19      justifyContent: 'center',
20    },
21  });
```

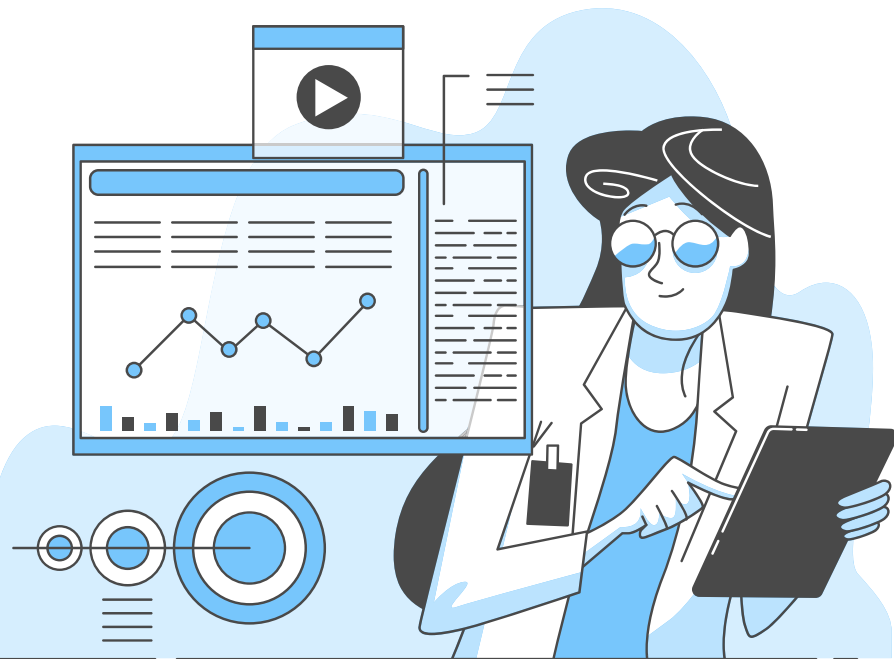
À la fin de notre composant, on retrouve généralement au moins deux choses : le **style css** ainsi que **l'export**. Pour le css, c'est le même que sur le web, la différence est qu'il est camel case.

Il est possible d'avoir d'autres sections, mais elles concernent des notions plus complexes que nous aborderons plus tard, dans la partie **redux**.



A vous de jouer !

Créez votre premier composant



01

...

[src/components](#)

Créez vos dossiers, afin d'y placer les composants

02

...

[HelloWorld.js](#)

Créez le fichier de votre composant

03

...

[Return Hello World](#)

Faites en sorte que votre composant retourne la string "Hello World"

04

...

[App.js](#)

Importez votre composant, afin que votre application affiche son rendu



Aller plus loin

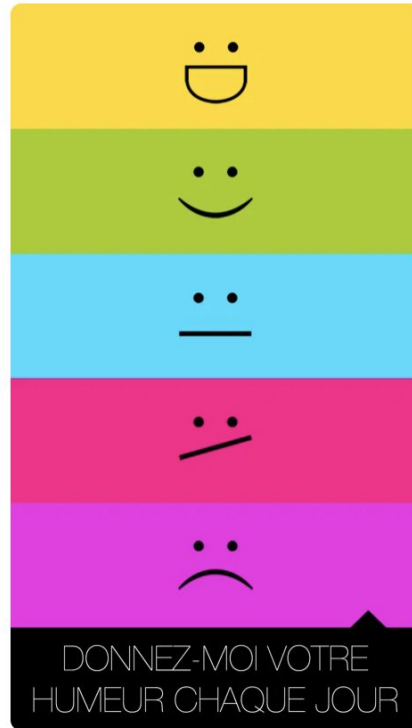
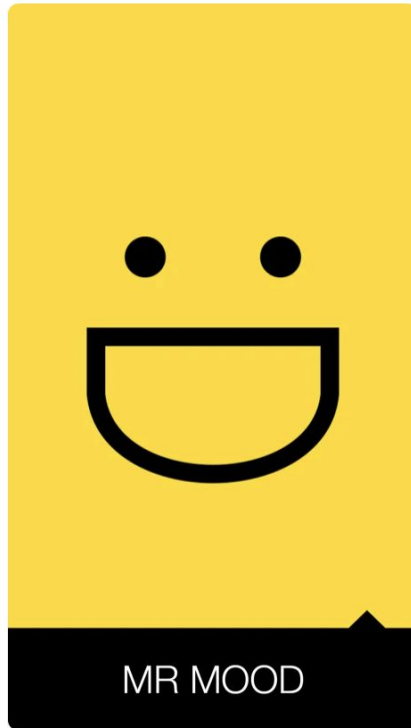


A partir de ce point, nous avons la base : on sait initialiser une app, et créer des composants autonomes simples, qui auront du css.

C'est bien, mais la réalité est beaucoup plus complexe et doit répondre à d'avantages de besoins. Par exemple, **une application se doit d'être interactive, et de réagir aux événements** de l'utilisateur. Pour ça, React inclus les notions de **propriétés** et d'**états**, plus communément appelé **props et state**.

Nous allons construire un mini projet à partir de maintenant afin d'amener ces notions pas à pas. On va reprendre le concept simple de l'application "Mr Mood" qui sert à garder en tête vos humeurs.

Mr Mood





La structure de notre application



01

Créez `src/components` & `src/screens`

`src/screens` contiendra également des composants, simplement, on les appellera **screens** s'ils sont des **parents** (i.e : les screens sont souvent ceux appelés au clic sur un menu tab, et ils ont pour vocation d'encapsuler tous les autres.

02

Créez `screens/List/index.js` et `screens/Info/index.js`

Ce seront nos deux composants dit "screens", pour vulgariser, ils vont correspondre à nos pages. List, est ce que l'on a dès qu'on arrive sur l'application : la liste de nos humeurs déjà présentes. Infos sera une page de statistiques de nos humeurs.

Commencez par créer ces deux screens, et faites en sorte que chacun retourne Hello World, comme vu précédemment.



03

La structure de notre application



Créez `src/config/store.js`

Souvent, on retrouve **un dossier config qui va contenir quelques données clés**. Pour le moment on va créer ce fichier js qui va simplement renvoyer un tableau en dur que l'on va interpréter.

Vous avez vu comment exporter un component et l'importer dans un autre, et, comme on a dit plus tôt, il est possible de faire de même avec n'importe quoi : une class, une variable, une lib... Essayez donc d'exporter un array à partir de `store.js` et de l'importer dans votre component `List`.

Pour le moment, faisons simple, et gardons un array avec une structure du type : **[{id: 1, mood: 5, title: "Parfait"}, {id: 2, mood: 3, title: "Normal"}]**.

Dans votre component `List`, trouvez le meilleur moyen de boucler sur votre array et affichez la note `mood` et le `title`.



La structure de notre application



```
8
9  {moods.map((mood) => (
10    <Text>{mood.title}</Text>
11  )}}
```

Pour ce point 03, la meilleure solution était d'utiliser **.map**. Petit point syntaxe, on fonctionne ici (et la plupart du temps) avec **ES6 (= ECMAScript 6)** qui permet de faciliter beaucoup de choses.

Typiquement, dans ce cas ci :

- (mood) => correspond à fonction (mood)
- => (...) correspond à { return (...) }

Il faut prendre un peu de temps pour s'adapter à cette syntaxe, mais on se rend vite compte qu'elle est un atout majeur niveau lisibilité et organisation du code.

store.js

```
1 export default MOODS = [  
2   {  
3     id: 1,  
4     mood: 5,  
5     title: "Parfait",  
6   },  
7   {  
8     id: 2,  
9     mood: 3,  
10    title: "Normal",  
11  },  
12 ];  
13
```

index.js

```
1 import React from "react";  
2 import { View, Text } from "react-native";  
3 import MOODS from "@config/store";  
4  
5 const List = (props) => {  
6   return (  
7     <View>  
8       {MOODS.map((mood) => (  
9         <Text>{mood.title}</Text>  
10       ))}  
11     </View>  
12   );  
13 }  
14  
15 export default List;  
16
```

04

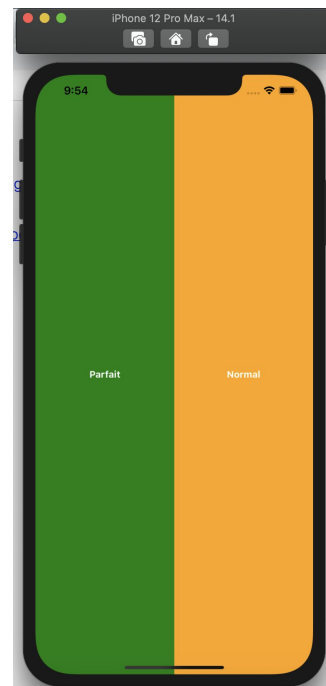
components/Mood.js

Ok ok, on a nos résultats en home désormais, mais ce qu'on voulait voir à la base, ce sont les props. Comme créé dans notre tableau, on a différents types d'humeurs qui correspondent à une note et un texte. L'idée, ça va être de donner un peu de style css en fonction de la note, et en utilisant un composant enfant.

Essayez de trouver le meilleur moyen de :

- Créer un composant Mood.js
- Importer `<Mood />` dans votre boucle à la place de juste afficher la valeur title
- Lui faire passer les props title et mood et les afficher dans le component Mood.js plutôt que List.
- En fonction de la props mood, donner un background différent

Les props



```
index.js — screens/List × App.js
1 import React from "react";
2 import { StyleSheet, View } from "react-native";
3 import MOODS from "@config/store";
4 import Mood from "@components/Mood"
5
6 const List = (props) => {
7   return (
8     <View style={styles.moods}>
9       {MOODS.map((mood) => (
10         <Mood
11           mood={mood.mood}
12           title={mood.title}
13           key={mood.id.toString()}
14         />
15       ))}
16     </View>
17   );
18 }
19
20 const styles = StyleSheet.create({
21   moods: {
22     flex: 1,
23     flexDirection: "row",
24   },
25 });
26
27 export default List;
```

```
index.js — components/Mood index.js — screens/List ×
1 import React from "react";
2 import { StyleSheet, View } from "react-native";
3
4 const Mood = (props) => {
5   const { title, mood } = props;
6
7   const _getStyle = (mood) => {
8     const colors = ["purple", "red", "orange", "yellow", "green"];
9
10    return colors[mood - 1];
11  };
12
13  return (
14    <View style={[styles.mood, { backgroundColor: _getStyle(mood) }]}>
15      <Text style={styles.moodText}>{title}</Text>
16    </View>
17  );
18 }
19
20 const styles = StyleSheet.create({
21   mood: {
22     flex: 1,
23     justifyContent: "center",
24   },
25   moodText: {
26     textAlign: "center",
27     color: "#fff",
28     fontWeight: "bold"
29   },
30 });
31
32 export default Mood;
```

A decorative network diagram with blue nodes and lines. The nodes are represented by concentric circles, with some having a solid blue center and others being hollow. They are connected by thin grey lines. There are three main paths: one in the top right corner, one in the bottom left corner, and one in the bottom center. Ellipses (...) are used to indicate that the network continues beyond the visible nodes.

Point oral sur les composants de base

Le state



“ Pour contrôler un composant, il y a deux types de données, les props et le state. Les props sont générées par le parent et ne peuvent être modifiées par l’enfant. ”

En résumé, le state va servir à changer des valeurs d’un composant, ou encore même son rendu.

Exemple : Au clic sur un `<Button />` on veut pouvoir ajouter un Mood dans le store, et que notre composant l’affiche.

Initialiser le state

```
6  class List extends React.Component {
7    constructor(props) {
8      super(props)
9
10     this.state = {
11       moods: MOODS,
12     }
13   }
14
15   render () {
16     const { moods } = this.state;
17
18     return (
19       <View style={styles.moods}>
20         {moods.map((mood) => (
21           <Mood
22             mood={mood.mood}
23             title={mood.title}
24             key={mood.id.toString()}
25           />
26         ))}
27       </View>
28     );
29   }
30 }
```

Changeons notre composant List en le faisant devenir une class, avec un constructeur et une méthode render.

Nous reviendrons sur ce point plus tard lorsqu'on abordera Redux, mais pour le moment utilisons la méthode de base. Cela permettra aussi de mieux comprendre le fonctionnement d'un composant et de connaître l'historique des méthodes de développement.

On initialise le state dans le constructeur via :
`this.state = { key: value };`

Value peut être n'importe quel type de variable (objet, tableau, string, int, ...).

Comprendre le state et le modifier

```
this.setState();
```

C'est cette fonction qui va nous permettre de **modifier le state**. Par exemple, admettons que l'on a `displayToto: false`, dans notre state. Pour le faire passer à `true`, on ferait :

```
this.setState({ displayToto: true });
```

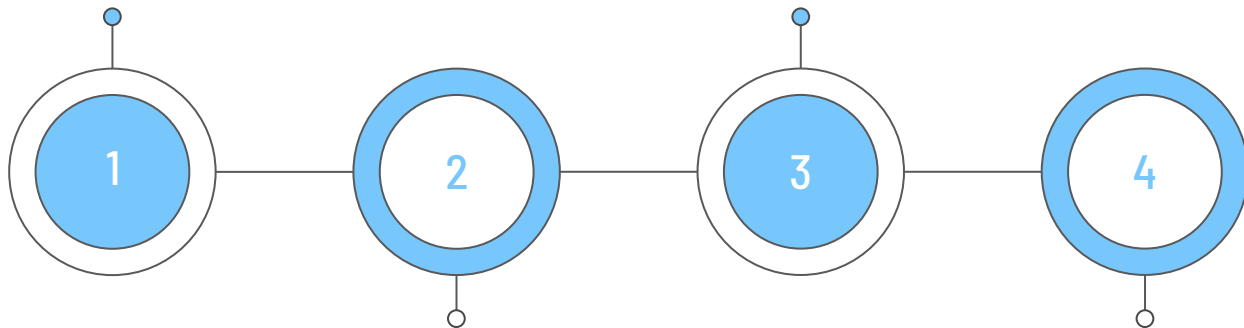
Comme vous le voyez c'est plutôt simple niveau syntaxe. **Ensuite, tout se passe dans le render**. Imaginons que ce booléen `displayToto` sert à afficher `<Text>toto</Text>` lorsqu'il est à `true`, alors on aura :

```
{displayToto && (  
  <Text>toto</Text>  
)}
```

Comprendre le state et le modifier

On initialise le state du
composant List

React détecte un
changement via setState et
demande au composant List
de se re-render



L'utilisateur trigger une
action qui modifie le state

Le composant List se
rend avec le nouveau
state

Le state

05

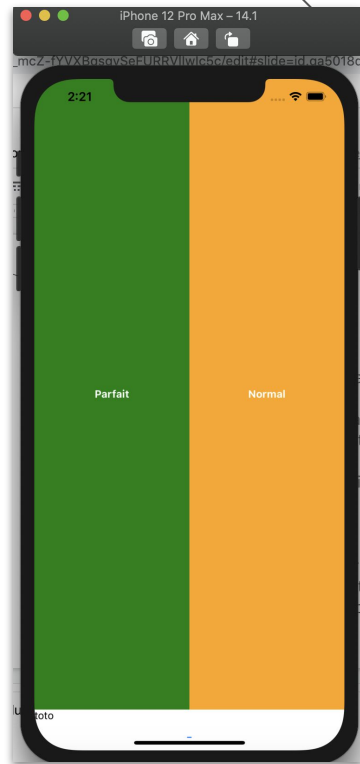
addIsOpen

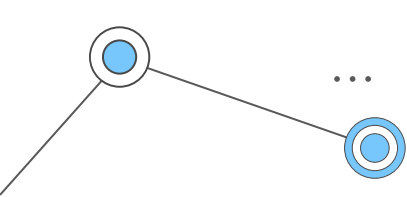
Maintenant que l'on a vu le principe de base, on va rendre possible l'ajout d'humeurs. Dans un premier temps, essayez de :

- Ajouter le booléen "displayAdd: false" au state
- Dans le composant, sous les humeurs, ajouter un `<Button />` qui aura pour titre "+".
- Ajouter une props "onPress" à ce bouton qui va venir modifier la valeur de displayAdd via `setState`.

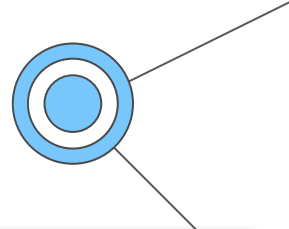
Critères d'acceptations :

- Si `displayAdd = false`, alors le bouton a pour titre "+"
- Si `displayAdd = true`, alors le bouton a pour titre "-", et une nouvelle `<View />` au dessus du bouton apparaît affichant "`<Text>toto</Text>`".





Le state



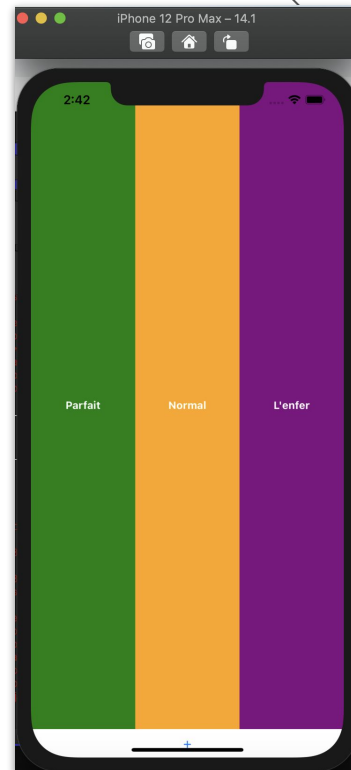
06

useState: moods

Tout est en place. Il ne reste plus qu'à pouvoir ajouter une humeur. Pour ce faire, à la place de `<Text>toto</Text>`, faites en sorte de :

- Retourner un bouton par niveau d'humeur, avec un texte différent, par exemple :D, :), :|, :/, :(.
- Ajouter un nouvel item à moods, via le useState, au clic sur un des boutons.
- Constater que votre composant se re-rend, et que votre nouvelle humeur est ajoutée.

(!!!!) Pensez à factoriser et optimiser au maximum votre code. Un composant doit rester léger, évitez d'avoir trop de code métier inutile.





Aller plus loin

07

Exercice libre

Prenez un peu de temps pour donner du style et de la cohérence à cette première partie de l'application, allez un peu plus loin que ce qui est proposé. Jouez avec les composants de base et, si vous le souhaitez, ajoutez même des fonctionnalités.

TODO :

- Trouvez un moyen pour que les humeurs aient toujours la même largeur, et que l'on obtienne un scroll horizontal si on dépasse la largeur du device
- Donnez des hauteurs aux humeurs en fonction de leur note mood.
- Faites en sorte que le bouton "+" soit rond et en overlay du reste
- Idem avec les boutons d'ajout d'humeur
- Bonus si vous finissez tôt : Ajouter des icônes smileys (cela vous fera découvrir la partie assets) sur ces boutons, ainsi que sur les barres d'humeurs au dessus du texte.

05

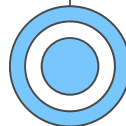
React navigation

Créer une navigation au sein de votre
application

...
...
...
Nous avons vu les trois grosses notions amenées par React. Ce ne sont évidemment pas les seules, mais les fondamentales.

Outre les principes de React.js, il est nous reste encore beaucoup de choses à voir. L'une d'entre elle est la navigation.

Elle est portée par le plugin React Navigation, créé par la communauté. Le plugin va permettre d'utiliser différents types de navigation.



Les différents types de navigation

Stack

- Gère une pile de screens
- Communique avec le natif et fait en sorte d'avoir le chevron "précédent" actif
- Permet de slide entre les screens de la même stack

Drawer

- Créé un menu burger
- Renvoie en général vers des stacks navigator

Tab

- Créé des tabs, comme on le voit fréquemment (ex: fb, insta) souvent utilisé en menu principal.
- Renvoi en général vers des stacks navigator

C'est ce que l'on utilisera dans notre cas.

Navigation – plugin et imports

01

Installez les plugins

```
$ npm install @react-navigation/stack @react-navigation/bottom-tabs  
@react-navigation/native
```

02

Créez src/navigation/MainNavigation.js

Ce fichier contiendra votre navigation, et comme prévu le premier navigator. Pour cela, on doit importer les dépendances comme dans tout composant.

```
1  import React from "react";  
2  import { StatusBar } from "react-native";  
3  import { NavigationContainer } from "@react-navigation/native";  
4  import { createBottomTabNavigator } from "@react-navigation/bottom-tabs";  
5  import { createStackNavigator } from "@react-navigation/stack";  
6  import List from "@screens/List";  
7  import Infos from "@screens/Infos";  
8  
9  const RootStack = createStackNavigator();  
10 const Tabs = createBottomTabNavigator();  
11
```

Navigation – NavigationContainer

03

Créez votre root / parent : le NavigationContainer

Tout passera par lui désormais, il appellera les différents Navigator de votre app. StatusBar ici représente la zone avec l'indicateur batterie, l'heure, etc. Ensuite vient notre RootStack.Navigator qui appelle notre TabsNavigator.

```
54  const MainNavigation = () => (  
55    <NavigationContainer>  
56      <StatusBar  
57        hidden={false}  
58        backgroundColor="transparent"  
59        barStyle="dark-content"  
60      />  
61      <RootStack.Navigator initialRouteName="MainStackScreen">  
62        <RootStack.Screen  
63          name="Main"  
64          component={TabsNavigation}  
65          options={{ headerShown: false }}  
66        />  
67      </RootStack.Navigator>  
68    </NavigationContainer>  
69  );  
70  
71  export default MainNavigation;  
72
```

Navigation – Tabs

04

Créez vos tabs

Comme vu dans le slide précédent, on appelle le TabsNavigation. On garde **en général tous nos navigators dans un même fichier**, à moins d'une architecture vraiment complexe.

Ici le **parent est TabsNavigator**, qui représente la bottom bar, avec quelques props et options de style.

A l'intérieur, chacun de nos screens sont listés, avec leurs propres informations. Il est possible de donner une icône, des comportements au clic, de rajouter des listeners et du style css, etc.

(!) Vous remarquerez l'important "navigation.navigate("ScreenName"), qui vous sera utile pour naviguer d'un screen à l'autre n'importe où dans votre app.

```
MainNavigation.js
12 const TabsNavigation = () => {
13   <Tabs.Navigator
14     initialRouteName="Main"
15     tabBarOptions={{
16       activeBackgroundColor: "#fff",
17       inactiveBackgroundColor: "#fff",
18       showLabel: true,
19       showIcon: true,
20       activeTintColor: "green",
21       inactiveTintColor: "#828282",
22     }}
23     tabStyle={{
24       flexDirection: "row",
25     }}
26   >
27     <Tabs.Screen
28       name="Main"
29       component={List}
30       options={{
31         title: "Moods",
32       }}
33       listeners={({ navigation }) => ({
34         tabPress: () => {
35           navigation.navigate("Main");
36         },
37       })}
38     />
39     <Tabs.Screen
40       name="Infos"
41       component={Infos}
42       options={{
43         title: "Infos",
44       }}
45       listeners={({ navigation }) => ({
46         tabPress: () => {
47           navigation.navigate("Infos");
48         },
49       })}
50     />
51   </Tabs.Navigator>
```

06

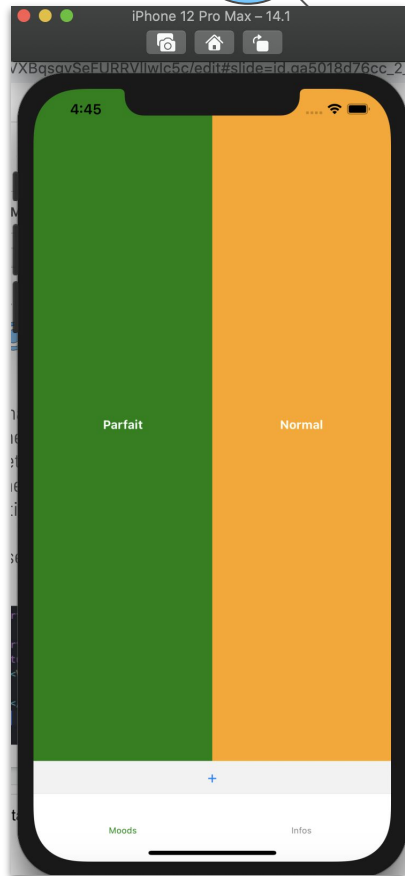
App.js

Désormais, vous avez défini une navigation au sein de l'application. Seulement, il faut que l'app la considère, et qu'elle passe par ça. Il faut donc retourner à la racine : App.js et supprimer le composant "List". Souvenez vous, vous l'avez appelé en route par défaut dans votre navigation.

L'idée sera donc de remplacer List, par votre navigation et le tour est joué

```
4 import MainNavigation from '@navigation/MainNavigation';
5
6 ~ export default function App() {
7 ~   return (
8 ~     <View style={styles.container}>
9       <MainNavigation />
10    </View>
11  );
12 }
```

Navigation – App.js





Aller plus loin

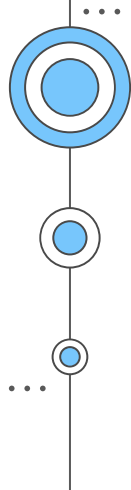


07

Exercice - drawer et header

Maintenant que vous avez saisi, à vous de jouer :

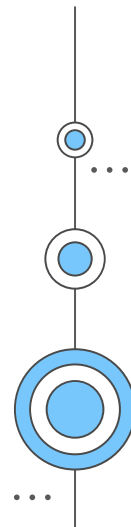
- Ajouter un header à l'application, avec un texte "MrMood" à droite, et une icône cliquable représentant un "profile" à gauche.
- Au clic sur cet icône profil, ouvrez un drawer navigator.
- Placez un lien dedans renvoyant vers un nouveau screen "Profile", renvoyez Hello World, comme au départ.
- Dans ce drawer toujours, affichez un second lien, tout en bas: "se déconnecter".



06

Persistence des données

Les différentes méthodes



Persistence des données



Notre application commence doucement à ressembler à quelque chose, mais, vous l'aurez remarqué en redémarrant l'app, les données ne persistent pas.

C'est normal, jusque là, nous avons simplement écrit un tableau en dur que l'on ajoute à l'initialisation du state. Ce dernier est modifié par la suite, mais rien de plus.

Nous allons ici étudier les méthodes les plus répandues pour persister ses données, et tester la plus simple d'entre elles.

Firestore

Firestore est un indispensable. Il n'est pas utile uniquement avec React Native, et pas utile uniquement pour de la donnée non plus. Il est bien plus vaste que ça. On peut y retrouver un système de notifications push, un suivi des crashes, un analytics tracker, un mode de distribution d'app, etc.

Dans notre cas, firestore met à disposition une base de donnée NoSQL en temps réel, qui regorge de fonctionnalités intéressantes. Par exemple, il est possible de stocker les actions de l'utilisateur s'il n'a plus de réseau et de tout pousser d'un coup par la suite, sans perdre de données.



Realm

Realm est très connu du monde mobile. C'est un système de base de données géré par MongoDB, donc encore en NoSQL. C'est un outil open-source utilisé par des centaines d'entreprises aujourd'hui.

Comme firebase, il va fonctionner sous forme de tableau JSON et permet le offline. Il offre également une application desktop afin de visualiser le contenu de votre base de données.

Autres exemples du type Realm / Firebase : SQLite, MongoDB (directement), PouchDB, ... il en existe beaucoup, mais les deux plus intéressants restent Firebase et Realm.



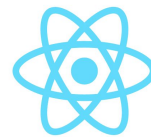
L'AsyncStorage

Il est le point commun entre tous. En effet, chacune des solutions précédentes va héberger votre database, mais au final, elle ira l'écrire dans votre AsyncStorage.

L'AsyncStorage c'est un système de stockage de données en clair, sous forme de clé:valeur. Pour faire simple, on stocke des tableaux de données JSON, comme dans les exemples avant.

Il est très simple d'utilisation, et surtout, tant que notre base de données ne devient pas complexe, il sera amplement suffisant. Nous allons donc l'utiliser pour stocker nos moods.

(!) Attention, dans le cadre d'une architecture plus complexe, il est à mon sens impératif de passer par Realm ou Firebase afin de maîtriser sa donnée.



AsyncStorage – Setup

01

Installer le plugin

```
$ npm install @react-native-community/async-storage
```

02

store.js

Retournez dans votre fichier store, et supprimez votre tableau en dur pour appeler l'AsyncStorage à la place. Ici, on décide d'appeler les valeurs ayant pour key "moods", au sein de notre async storage. Tout est en JSON, donc pensez à parse votre retour de data.

```
const values = await AsyncStorage.getItem("moods");  
const jsonValues = JSON.parse(values);  
  
return jsonValues;
```

(!) C'est quoi **await** ?

AsyncStorage – getItem

03

Préparer le code pour la suite

Pour l'instant, on ne récupère rien, c'est normal, notre AsyncStorage est vide. Avant d'avancer sur ce point, voici un exemple de fonction un peu plus générique que ce que l'on vient de produire :

```
store.js                                     index.js
1  import { AsyncStorage } from "react-native";
2
3  export const getItem = async (key) => {
4    try {
5      const values = await AsyncStorage.getItem("moods");
6      const jsonValues = JSON.parse(values);
7
8      return jsonValues;
9    } catch (e) {
10     console.log('ERROR STORE.JS getItem - ', e.toString());
11   }
12 }
13
```

AsyncStorage – setItem

04

setItem

De la même manière, on va créer la fonction qui vient modifier l'AsyncStorage sur une clé donnée.

```
14 export const setItems = async (key, values) => {  
15   try {  
16     const jsonValues = JSON.stringify(values)  
17     await AsyncStorage.setItem(key, jsonValues);  
18   } catch (e) {  
19     console.log('ERROR STORE.JS setItems - ', e.toString());  
20   }  
21 }  
22
```

AsyncStorage – appel dans le component

05

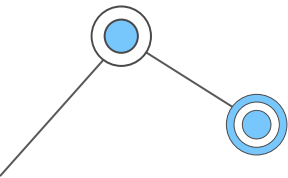
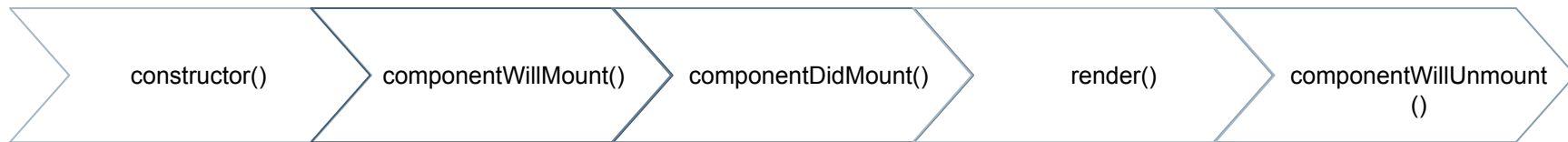
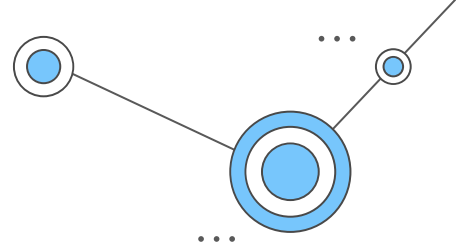
Mise en oeuvre côté composant

componentDidMount est une fonction dite “lifecycle”. Elle est introduite par React. Concrètement, il existe plusieurs méthodes pour le “cycle de vie” d’un composant que l’on verra sur le slide suivant. Celui-ci sera appelé lorsque le composant viendra d’être monté.

*(!) Point sur la structure **...moods***

```
6 class List extends React.Component {
7   constructor(props) {
8     super(props)
9
10    this.state = {
11      addIsOpen: false,
12      moods: [],
13    }
14  }
15
16  async componentDidMount() {
17    const storedMoods = await getItem("@moods");
18
19    this.setState({
20      moods: storedMoods
21    });
22  }
23
24  _addMood() {
25    const { moods } = this.state;
26    const updatedMoods = [
27      ...moods,
28      {
29        id: moods.length + 1,
30        mood: 1,
31        title: "L'enfer"
32      }
33    ];
34
35    setItem("@moods", updatedMoods);
36    this.setState({
37      addIsOpen: false,
38      moods: updatedMoods,
39    });
40  }
41
42  render() {
43    const { moods, addIsOpen } = this.state;
44  }
```

Cycle de vie d'un composant



AsyncStorage – à vous de jouer

06

Exercice

Maintenant que l'on a vu comment set un item de notre AsyncStorage et que l'on a implémenté l'ajout, essayez de :


- Trouver un moyen fonctionnel de proposer la suppression d'une humeur via une pression longue sur un mood
- Supprimer ce mood de l'AsyncStorage



07

Redux

Aller plus loin avec le state et les
composants



C'est quoi Redux ?



“C’est une librairie Javascript qui permet de gérer un state global à l’application.”

Pour faire simple, au lieu d’avoir un state par composant, on pourra aussi avoir un state global à l’application, afin de partager des données dans tous les composants si on le souhaite.

Exemple: gérer si l'utilisateur est connecté ou non, et faire des conditions en fonction de ce booléen, dans chaque screen.

Redux – la problématique

Dans notre contexte, prenons le deuxième screen “Infos”. Admettons que l’on veuille ici afficher les statistiques de nos humeurs. Par exemple : combien d’humeurs sont présentes, quel est ma note moyenne, une phrase donnant mon humeur en fonction de cette note moyenne, etc.

On pourrait simplement appeler l’AsyncStorage à nouveau, faire nos calculs et les afficher. Et cela va fonctionner la première fois que l’on rentre dans le screen. Mais si on retourne sur le screen “Moods” et que l’on en supprime ou en ajoute un, puis que l’on revient sur Infos, les données ne seront pas à jour car ce component a déjà été “monté”. On ne peut pas lui faire passer les infos car il n’est pas non plus un enfant de `<List />`.

C’est ici, entre autres, que Redux intervient. Le fait de définir un state global va permettre d’appeler la liste des humeurs de partout.

Redux – le fonctionnement flux

Redux est basé sur Flux, qui est une architecture de gestion de données. Pour résumer ça dans notre contexte : on **pull des données** (par exemple via un appel API) avec une fonction dite “**action**”, puis **on l’envoi au store via le dispatcher**, ce **store gère le state global et va aller fournir les composants**. C’est un peu flou dit comme ça, mais en réalité ça ne l’est pas tellement et on va décortiquer tout ça ensemble.



Redux – Setup

01

```
$ npm install redux  
$ npm install react-redux  
$ npm install redux-devtools-extension  
$ npm install redux-persist  
$ npm install redux-thunk
```

02

Architecture

On va aborder chacune des notions. Mais en attendant on va préparer notre arborescence, pour que ça soit un peu plus clair durant les explications :

- Supprimez votre fichier store.js, et par conséquent, ses répercussions dans les composants. Le store va désormais être géré par redux.
- Créez un dossier src/redux et un fichier index.js dedans (c'est là qu'on va créer la définition de votre nouveau store plus tard.
- Créez respectivement src/redux/moods/actions.js, src/redux/moods/constants.js, src/redux/moods/reducer.js.

Redux – actions.js

On l'a vu via le schéma, **tout part de l'action**. Ne vous compliquez pas trop les choses, avec tous ces nouveaux mots “actions”, “reducers”, etc, vous allez les retenir à force car n'est pas si complexe ça.

Par exemple, **une action n'est au final qu'une fonction qui a pour vocation de transmettre des données au reducer**, mais, **concrètement, cela reste une fonction**. Mettons en place notre première action dans actions.js.

```
actions.js                                store.js
1  import moodsActions from "../constants";
2
3  export function getMoods() {
4    // Pour le moment, on repart sur un tableau en dur.
5    // Mais typiquement, on pourrait ici retrouver un appel API.
6    const moods = [
7      {
8        id: 1,
9        mood: 5,
10       title: "Parfait"
11     }
12   ];
13
14   // On passe les infos au reducer, via le dispatcher.
15   return async function (dispatch) {
16     dispatch({
17       type: moodsActions.GET_MOODS,
18       moods: moods,
19     })
20   };
21 }
22
```

Redux – constants.js

Les constantes servent uniquement à lister les types d'actions que l'on créé, et donc, à les appeler de partout au besoin. C'est une bonne pratique de les extraire dans un fichier constant.js car dans certains cas une action être dispatch de beaucoup d'endroits différents.

Voici à quoi ressemble en général ce type de fichier.

```
constants.js
1  const moodsActions = {
2    GET_MOODS: "GET_MOODS",
3  };
4
5  export default authActions;
6
```


Redux – reducer.js : initial state

Le reducer, c'est là que tout se joue, c'est la partie qui va générer le fameux state global.

On va le décortiquer en deux étapes. Premièrement, on a “l'initial state”, qui va définir notre state et son état à l'arrivé sur l'application.

En général on définit un object js avec des clés / valeurs. Les valeurs sont normalement vides, puisqu'elles seront peuplés par les actions.

Voici notre initialState dans reducer.js.

```
reducer.js  x  constants.js
1  import moodsActions from "../constants";
2
3  const initialState = {
4    moods: [],
5  };
6
```

Redux – reducer.js : le state

Une fois l'initialState créé. Il ne nous reste plus qu'à **écouter nos différents types d'actions**.

Si une action dispatch des données vers le reducer, ce dernier va récupérer ces données et **mettre à jour son state en se basant sur l'archi de l'initialState**.

Voici un exemple du code pour notre action de type "GET_MOODS" afin de clarifier.

```
reducer.js      constants.js      ac
1  import moodsActions from "../constants";
2
3  const initialState = {
4    moods: [],
5  };
6
7  export default function moodsReducer(state = initialState, action) {
8    switch (action.type) {
9      case moodsActions.GET_MOODS:
10       return {
11         ...state,
12         moods: {
13           ...moods,
14           action.moods,
15         }
16       };
17     }
18   }
19 }
```

Redux – reducer.js : le state

moodsReducer prend en paramètre son **state**, s'il n'est pas encore défini, il prend l'**initialState** : C'est le cas à l'ouverture de l'app, lorsqu'aucune action n'a encore été déclenchée. Ensuite, **on reçoit en second paramètre l'action**, et on crée un switch sur son type.

Selon le type, on vient modifier la partie du state qui est sensé changer, en utilisant la déstructuration de tableau vu plus tôt dans cette formation.

Au final, **un reducer un est gros switch case** qui vient mettre à jour le **state** avec les données envoyées par l'action correspondante.

```
reducer.js      constants.js      ac
1  import moodsActions from "../constants";
2
3  const initialState = {
4    moods: [],
5  };
6
7  export default function moodsReducer(state = initialState, action) {
8    switch (action.type) {
9      case moodsActions.GET_MOODS:
10       return {
11         ...state,
12         moods: {
13           ...moods,
14           action.moods,
15         }
16       };
17     }
18   }
19 }
```

Redux – le store

On a créé notre premier global state via Redux ! Très bien. Maintenant, dans la logique des choses, **il faut afficher tout ça dans l'application pour que l'on voit nos moods apparaître à nouveau.**

Rappelez-vous, à la base, on avait un fichier config/store.js qui renvoyait la liste des moods stockés dans **l'AsyncStorage**, appelé directement via le component List. Je vous ai dit de supprimer ce fichier car on allait passer par Redux. C'est vrai, mais il faut que l'on **définisse à nouveau notre store** pour que nos states globaux soient accessibles partout.

C'est là qu'intervient notre fichier **redux/index.js**. **Il va s'occuper de rassembler tous nos reducers et de créer à partir d'eux le store** (on a défini le moodReducer uniquement, mais on pourrait - et c'est généralement le cas - en avoir une dizaine).

Vous devez aussi vous demander, si vous suivez, qu'est-ce qu'on fait de la persistance des données ? Où est passé notre AsyncStorage ? Et bien, c'est également **dans le store que l'on va définir notre système de stockage.**

Redux – redux/index.js

```
▼ src
  > components
  > config
  > navigation
  ▼ redux
    ▼ moods
      actions.js
      constants.js
      reducer.js
    index.js
    package.json
  screens
    ▼ Infos
      index.js
    ▼ List
      index.js
      package.json
  .gitignore
  App.js
```

```
index.js — redux      reducer.js      constants.js
1  import { createStore, applyMiddleware } from "redux";
2  import { composeWithDevTools } from "redux-devtools-extension";
3  import { persistCombineReducers } from "redux-persist";
4  import { AsyncStorage } from "react-native";
5  import thunk from "redux-thunk";
6  import moods from "@redux/moods/reducer";
7
8  const reducers = {
9    moods,
10 };
11
12 export default createStore(
13   persistCombineReducers(
14     {
15       key: "root",
16       storage: AsyncStorage,
17     },
18     reducers
19   ),
20   composeWithDevTools(applyMiddleware(thunk))
21 );
22
```

Redux – composants

On touche au but. Redux est maintenant en place. Je vous rassure, à force de créer des reducers et d'ajouter des actions, vous prendrez vite la main. **La dernière chose à faire va être d'appeler le store à partir du composant** pour afficher nos moods.

Pour ça, on va supprimer ce qu'on avait fait au niveau du state local du composant. **Notre constructeur et notre fonction componentDidMount n'ont plus d'intérêts pour nous**, puisqu'on utilise le state global. **Plus globalement, notre composant List n'as plus besoin d'être une class** (puisque'elle n'a plus besoin de méthodes lifecycle).

Copiez / collez le code ici, et nous allons le décortiquer ensemble.

```
index.js
1  import React from "react";
2  import { View, Text, Button, StyleSheet } from "react-native";
3  import Mood from "@components/Mood"
4
5  const List = () => {
6    const { moods } = props;
7    const addIsOpen = false;
8
9    return (
10     <View style={styles.container}>
11       <View style={styles.moods}>
12         {moods && moods.length > 0 && (
13           <View>
14             {moods.map((mood) => (
15               <Mood
16                 mood={mood.mood}
17                 title={mood.title}
18                 key={mood.id.toString()}
19               />
20             )}}
21           </View>
22         )}
23       </View>
24
25       {addIsOpen && (
26         <Text>toto</Text>
27       )}
28
29       <Button
30         onPress={}
31         title={addIsOpen ? "-" : "+"}
32         style={styles.button}
33       />
34     </View>
35   );
36 }
37
38 const styles = StyleSheet.create({
```

Redux – composants

Pour infos, je vous ai montré un composant sous forme de class, avec les méthodes lyfecyles, etc. Ce n'est presque plus utilisé. Je l'ai fait pour vous faire comprendre comment fonctionne React de base, ce qui est important pour que vous ayez l'historique et la logique des choses, mais **la bonne pratique est celle que nous sommes actuellement entrain de mettre en place**. Plus pratique, plus lisible, et surtout, plus performante.

Reprenons notre code maintenant, on peut voir **const { moods } = props**. Wtf, on a généré nos moods dans le state global, et on les retrouve dans props ? Oui, car c'est comme ça que fonctionne Redux. Actuellement, cela ne doit pas marcher chez vous, car il faut créer un mapping via la méthode qui parle d'elle même “**mapStateToProps**”.

```
43     flex: 1,  
44     flexDirection: "row",  
45   },  
46 });  
47  
48   const mapStateToProps = (state) => ({  
49     moods: state.moods.moods,  
50   });  
51  
52   export default connect(mapStateToProps, null)(List);
```

Redux – composants

mapStateToProps, comme vous l'aurez compris va donc **mapper le state global sur les props du composant et trigger le re-rendering du composant**. DONE. Cela fait beaucoup de notions, et ce n'est pas complètement terminé, mais l'essentiel est là.

Pour finir, vous avez dû remarquer que l'on a supprimé l'ajout d'humeur, etc. il va falloir les remettre en place, via redux. Si vous avez bien compris comment fonctionne **mapStateToProps**, vous devriez comprendre facilement **mapDispatchToProps**. Voici à quoi ressemble la fin de notre fichier désormais :

```
42     moods: {
43       flex: 1,
44       flexDirection: "row",
45     },
46   });
47
48   const mapStateToProps = (state) => ({
49     moods: state.moods.moods,
50   });
51
52   const mapDispatchToProps = (dispatch) =>
53     bindActionCreators(
54       {
55         addMood,
56       },
57       dispatch
58     );
59
60
61   export default connect(mapStateToProps, mapDispatchToProps)(List);
```


Redux – Exercices

03

addMood

C'est votre tour, prenez le temps de :

- Créer votre action addMood
- Créer le type d'action ADD_MOOD dans constant.js
- Catcher l'action dans le reducer.js afin de mettre à jour le state

04

deleteMood

Remettez en place votre suppression.

05

<Infos />


Dans votre composant Info, affichez quelques statistiques en faisant appel à vos moods comme par exemple le nombre de moods et le mood global.



08

Les hooks

Optimisez vos composants



React – les hooks

On a vu comment créer un state global. Cependant, on peut toujours avoir besoin de créer un state local à un composant. Prenons par exemple notre système avec le bouton “Add Mood”. On a aucun intérêt à le store côté Redux, puisqu’il n’est présent qu’ici. Mais comment faire ? On a supprimé le constructor, et le côté lifecycle car on y gagne en performance et que l’on a vu que c’est la bonne pratique.

Et bien pour ça, React depuis v16.8 a introduit des “hooks d’effets”. On va voir ensemble les deux plus répandues “useEffect”, et “useState”. Mais pour généraliser, ils vont servir à remplacer les méthodes lifecycle.

React - useState

Dans notre composant List, on va utiliser addIsOpen en state local.

Il suffit :

- Importer useState en haut du fichier
- Déclarer un state local avec const [nomState, setNomState] = useState(defaultValue)

Et le tour est joué. Attention, il est important de garder le côté camelcase ici.

Enfin, au onPress, on a plus qu'à appeler le setter, et le composant va se re-rendre de lui-même.

```
1 import React, { useState } from "react";
2 import { View, Text, Button, StyleSheet } from "react-native";
3 import Mood from "@components/Mood"
4
5 const List = () => {
6   const { moods } = props;
7   const [addIsOpen, setAddIsOpen] = useState(false);
8
9   return (
10     <View style={styles.container}>
11       <View style={styles.moods}>
12         {moods && moods.length > 0 && (
13           <View>
14             {moods.map((mood) => (
15               <Mood
16                 mood={mood.mood}
17                 title={mood.title}
18                 key={mood.id.toString()}
19               />
20             ))}
21           </View>
22         )}
23       </View>
24
25       {addIsOpen && (
26         <Text>toto</Text>
27       )}
28
29       <Button
30         onPress={setAddIsOpen(!addIsOpen)}
31         title={addIsOpen ? "-" : "+"}
32         style={styles.button}
33       />
34     </View>
35   );
36 }
```

React – useEffect

Le **useEffect** quant à lui **va servir à remplacer les fonctions tels que `componentDidMount`, `componentDidUpdate`**, et autres. Je vous laisserai le tester durant l'exercice final, il est un peu plus complexe à appréhender.


Exemple de mise en place précise ici <https://fr.reactjs.org/docs/hooks-effect.html>



09

Aller plus loin

React Native debugger, TypeScript,
Sentry, ...



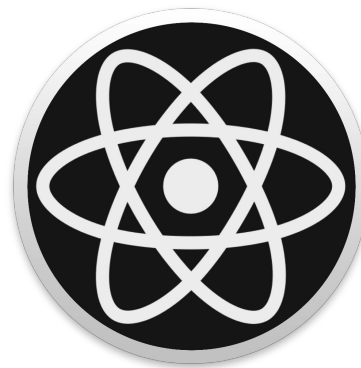
React Native Debugger

On l'a vu, on peut trigger le "remote js debug" avec le simulateur / device lorsque l'on code sous React Native.

Mais, lorsque l'on commence à avoir plusieurs reducer, un workflow plus complexe, il devient difficile de suivre son state global et de debugger son application.

C'est là qu'intervient cet outil. Il va vous permettre, en plus de la console, de suivre en temps réel l'état de votre state redux, les actions qui sont déclenchées, les diffs, les variables qui son passé via le dispatcher, etc.

Il est indispensable.



TypeScript

TypeScript est un langage de programmation créé par Microsoft en 2012. Rappelez vous, au tout début, lorsqu'on a init notre application expo nous a demandé si on voulait démarrer avec un projet "blank" ou "typescript". Et bien c'est de ce langage qu'il parlait.

Son intérêt principal c'est qu'il va amener des concepts de base de la programmation objet au javascript. Tout le monde le sait, js est un langage un peu bâtard et manque de certains principes de base.

Par exemple TypeScript va amener le typage des variables.

Si vous commencez à avoir beaucoup de code métier, c'est aussi un indispensable.



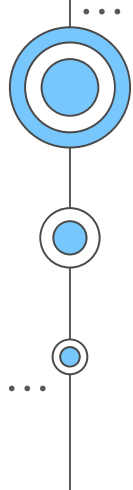
Sentry

Sentry c'est un tracker d'erreur. Il est extrêmement utile à partir du moment où vous passez votre application en production.

Concrètement, vous allez pousser vers Sentry toutes vos errors remontées par vos try catch. Il va s'occuper de les lister, en donnant la version de l'utilisateur qui a eu l'erreur, son OS, sa version d'OS, localisation, etc. Jusqu'à la trace dans le code.

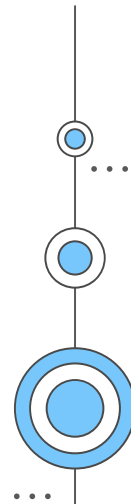
Quand on développe une application, il y a une multitude de support et de versions différentes, il est quasi impossible de pouvoir tout tester sans passer par un centre de QA (et pour ça, il faut le budget). Sentry permettra donc d'avoir une vision d'ensemble sur les problèmes rencontrés par vos utilisateurs, et de les isoler afin de les résoudre.





10

Evaluation



Votre première app React Native



Je vous propose d'être seul, ou en groupe de 3 maximum, et de créer votre propre application. Il n'y a pas de thème imposé, faites vous plaisir, soyez créatif, ou à défaut, calquez un concept existant.

Critères (si tout est présent et fonctionnel, vous assurez un 16) :

- Au moins 3 screens
- Au moins 5 composants (avec des props)
- Au moins deux reducers
- Utilisez useState et useEffect
- Mettez en place une navigation
- Code réfléchi et optimisé

Pour avoir 20 :

- Mettez en place Realm ou Firebase
- Mettez en place TypeScript



Merci!

Si besoin contact : LéoIios sur Kecharm

Sources

- <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/>
- <https://github.com/jhen0409/react-native-debugger/releases>
- <https://reactnative.dev/>
- <https://www.typescriptlang.org/>
- <https://redux.js.org/>
- <https://realm.io/>
- <https://firebase.google.com/>