

# Contents

<b>1</b>	<b>Séance 1</b>	<b>3</b>
1.1	Construction d'une application multi-plateforme . . . . .	3
1.2	Une bonne version de <b>node</b> . . . . .	4
1.3	Syntaxe ES6 . . . . .	4
1.3.1	Module Javascript . . . . .	4
1.3.2	Affectation de multiples variables . . . . .	5
1.3.3	Fonction fléchées (Arrow function) . . . . .	5
1.3.4	Modèles de libellés . . . . .	6
1.4	Initialisation d'une application . . . . .	6
1.4.1	Sur les machines du département et en particulier sur la VDI . . . . .	6
1.4.2	Sur votre machine personnelle . . . . .	7
1.4.3	Démarrage du projet . . . . .	7
1.4.4	Exécution sur tablette/mobile . . . . .	7
1.4.5	Gestion de la console de développement . . . . .	7
1.5	Analyse du composant principal . . . . .	7
1.6	Un premier composant . . . . .	9
1.7	Initiation à la gestion de l'état ( <i>state</i> ) . . . . .	10
1.8	Passage de propriétés ( <i>props</i> ) . . . . .	11
1.9	TP 1 . . . . .	12
1.9.1	Initialisation d'un projet <b>React Native</b> . . . . .	12
1.9.2	Démarrage du projet . . . . .	12
1.9.3	Application de gestion de <b>todos</b> . . . . .	12
<b>2</b>	<b>Séance 2</b>	<b>15</b>
2.1	Résumé de la Séance 1 . . . . .	15
2.2	Communications fils -> père . . . . .	16
2.3	Composant sous forme de classe . . . . .	16
2.4	TP 1 . . . . .	17
<b>3</b>	<b>Séance 3</b>	<b>21</b>
3.1	Modification d'état suite à des modifications de <i>props</i> . . . . .	21
3.1.1	Mise à jour du compteur de tâche . . . . .	22
3.2	Mise en page . . . . .	22

3.2.1	Utilisation de <b>flex</b> . . . . .	23
3.2.2	Répartition des tailles ( <i>main axis</i> ) . . . . .	23
3.2.3	Utilisation de la taille de l'écran . . . . .	24
3.3	TP 3 . . . . .	24
3.4	TP Sudoku . . . . .	24
3.5	Navigation . . . . .	26
3.5.1	Mise en oeuvre . . . . .	26
3.5.2	Persistence de variables pendant la navigation . . . . .	27
3.5.3	Navigation depuis un composant . . . . .	29
3.5.4	Imbrication de navigation . . . . .	29
3.5.5	Gestion des liens . . . . .	30
3.5.6	Structuration finale du code . . . . .	30
3.5.7	Liens . . . . .	31
3.6	Token d'authentification . . . . .	31
3.6.1	Les jetons <b>JWT</b> . . . . .	31
3.6.2	Utilisation du jeton par React . . . . .	32
3.6.3	API CRUD pour fournir le jeton . . . . .	33
3.7	TP 4 . . . . .	35
<b>4</b>	<b>Séance 5</b> . . . . .	<b>37</b>
4.1	Résumé de la séance précédente . . . . .	37
4.2	Conception générale d'une application React . . . . .	37
4.3	Hook d'effet . . . . .	38
4.4	TP 5 . . . . .	39
<b>5</b>	<b>Séance 6</b> . . . . .	<b>41</b>
5.1	Construire l'application . . . . .	41
5.2	Utilisation d' <b>expo</b> . . . . .	41
5.3	Utilisation d' <b>androidstudio</b> . . . . .	41
5.4	Procédure de développement classique, sans <b>expo</b> . . . . .	42
5.5	Génération de l' <b>apk debug</b> . . . . .	42
5.6	Génération d'un <b>apk release</b> . . . . .	43
5.7	Conclusion . . . . .	44
5.8	Liens / tips . . . . .	44
<b>6</b>	<b>Séance 7</b> . . . . .	<b>47</b>
6.1	Retour sur les fonctions asynchrones . . . . .	47
6.1.1	<b>async</b> et <b>await</b> . . . . .	48
6.1.2	Écriture de promesses personnalisées . . . . .	49
6.1.3	Lien . . . . .	50
6.2	Retour sur les <i>hooks</i> . . . . .	50
6.2.1	Hook d'état . . . . .	50
6.2.2	Hook de contexte . . . . .	50
6.3	Fonctionnalités additionnelles . . . . .	50
6.3.1	Activity Indicator . . . . .	50
6.3.2	Utilisation des fonctionnalités du téléphone . . . . .	52

# Chapter 1

## Séance 1

**React** est une librairie développée par Facebook, conçue pour programmer des interfaces web dynamiques, à l'aide de **Javascript**.

**React Native** est une version de React qui permet d'effectuer du développement mobile *multi-plateforme* : un même code pour Android et iOS. C'est une différence majeure avec le développement *natif*, qui utilise des langages différents selon les plateformes : `java` ou `kotlin` pour Android et Objective C ou Swift pour iOS. Le développement natif produit des applications plus performantes, mais demande de maintenir deux versions de l'application.

React Native utilise directement les composants graphiques fournis par le système : texte, bouton, image, etc. Ces composants peuvent avoir un rendu différent selon les plateformes, par exemple les boutons.

### 1.1 Construction d'une application multi-plateforme

React Native se programme en JSX, un mélange de Javascript et de balises en XML. Il n'est donc pas possible de l'exécuter directement, par exemple dans un navigateur. Il faut passer par un serveur qui effectue des traductions ou compiler le code pour construire une application, pouvant être rendue disponible sur les *stores*. Sur Android, c'est **AndroidStudio**, disponible sur toutes les plateformes, qui doit être utilisé pour construire une application. Pour iOS, il faut utiliser XCode, disponible uniquement sur les ordinateurs Apple.

Pour palier les multiplicités de ces points de vue, nous utiliserons **expo** (Expo Go est le nom complet de l'application), disponible sur Android et iOS, qui permet d'exécuter sur mobile du code en développement sur l'ordinateur. **expo** permet également de tester dans un navigateur web. Enfin, le site <https://expo.dev>

permet de partager des applications, fournit l'infrastructure pour les exécuter, les construire et les déposer sur les *stores*.

Attention cependant, **expo** ne semble pas adapté à du développement professionnel, voir cette discussion intéressante sur son intérêt d'Expo.

## 1.2 Une bonne version de node

**node** ou **nodejs** est un interpréteur de Javascript, central dans React. Il faut une version récente de **node**, ce qui n'est pas toujours disponible en standard sur toutes les distributions Linux.

Deux solutions :

1. utiliser **node\_source** qui met à jour la liste de paquets de la machine et installe la bonne version de **node**
2. installer **nvm** : **node version manager** : <https://github.com/nvm-sh/nvm>

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash
```

**nvm** s'installe dans le répertoire `~/.nvm` et modifie le fichier `.bashrc` de façon à lancer le script `~/.nvm/nvm.sh` qui permet d'installer toute version de **node** et paramétrer la variable `PATH`. Par exemple :

```
$ nvm install 14.15.1
Downloading and installing node v14.15.1...
Now using node v14.15.1 (npm v6.14.8)
$ node -v
v14.15.1
$ ls ~/.nvm/versions/node/v14.15.1/bin
node  npm  npx
```

## 1.3 Syntaxe ES6

React (et donc React Native) utilise la version ES6 de Javascript, avec laquelle il convient de se familiariser, sous peine de ne pas comprendre le code qu'on lit ou qu'on doit écrire.

### 1.3.1 Module Javascript

On peut exporter la définition d'une fonction ou d'une variable de deux façons :

- export nommé :

```
// définition dans le fichier du module MonModule.js
export maVar = ...
```

```
export function maFonction(...){ ...

// utilisation dans un autre module
import { maVar, maFonction } from './MonModule' // chemin relatif

  • export par défaut :

// définition dans le fichier du module MonModule.js
const maVar = ...
export default maVar;

// utilisation dans un autre module
import maVar from './MonModule'
```

### 1.3.2 Affectation de multiples variables

#### 1.3.2.1 à partir d'un objet structuré

```
const { film, displayDetailForFilm } = this.props
```

équivalent à

```
const film = this.props.film
const displayDetailForFilm = this.props.displayDetailForFilm
```

#### 1.3.2.2 à partir d'un tableau

C'est l'affectation par décomposition :

```
const [a, b] = tab
```

équivalent à

```
const a = tab[0]
const b = tab[1]
```

### 1.3.3 Fonction fléchées (Arrow function)

ES6 permet de définir des fonctions avec la syntaxe suivante :

```
// declaration de fonction
function coucou(aqui) {
  return `coucou, ${aqui}!`;
}

// expression de fonction
const coucou = function(aqui) {
  return `coucou, ${aqui}!`;
}

// arrow function
```

```
const coucou = (aqui) => {
  return `coucou, ${aqui}!`;
}
```

La première différence est qu'une fonction fléchée retourne implicitement une valeur si on n'utilise pas les accolades :

```
const increment = (num) => num + 1;
// équivaut à
const increment = (num) => {return num + 1};
```

La deuxième différence importante que nous utiliserons dans React est la valeur de `this` considérée par une méthode de classe, selon qu'elle est définie par une fonction classique ou une fonction fléchée.

Il faut retenir que :

- dans le cas d'une méthode définie par une fonction classique, `this` est déterminé par la fermeture du contexte d'appel (closure) : si `this` existe dans le contexte d'appel, c'est ce `this` qui sera utilisé à l'intérieur de la méthode. Ce n'est généralement pas ce que l'on souhaite lorsqu'on fournit une méthode comme *callback*, on souhaite que la *callback* utilise le `this` défini lors de l'écriture de la méthode.
- si la méthode est définie par une fonction fléchée, la nature de `this` est décidée *syntactiquement*, c'est-à-dire que c'est le `this` de la définition de la méthode.

Plus de détails

### 1.3.4 Modèles de libellés

Ce sont des chaînes de caractères délimités par des *backquote* permettant d'insérer des fragments de code Javascript :

```
var a = 5;
var b = 10;
console.log(`Quinze vaut ${a + b}`);
```

## 1.4 Initialisation d'une application

### 1.4.1 Sur les machines du département et en particulier sur la VDI

Voir les détails ici

- télécharger l'archive du projet avec le dossier `node_modules` dont les liens symboliques sont déréférencés
- la désarchiver dans `~/Documents`
- renommer le dossier selon votre choix

### 1.4.2 Sur votre machine personnelle

Voir <https://reactnative.dev/docs/environment-setup>

```
npx create-expo-app AwesomeProject
```

### 1.4.3 Démarrage du projet

Une fois installé, il suffit de faire `npm run start` ou `npx expo`. Un serveur `expo` est lancé, vous permettant d'ouvrir l'émulateur web :

- appuyer sur `w`
- le projet est alors compilé avec `webpack`
- le navigateur est ouvert pour pointer sur `http://localhost:19006/`
- en cas de modification d'un fichier source, le projet est recompilé

En cas d'erreur de compilation, un message est affiché dans la console du compilateur, c'est-à-dire dans le terminal qui a lancé la compilation.

### 1.4.4 Exécution sur tablette/mobile

Votre tablette/mobile doit être sur le même réseau que la machine qui fait tourner le serveur `expo`. Si votre serveur est sur la VDI ou une machine de TP, **ce n'est pas possible** car le wifi `eduroam` est un réseau indépendant de l'infrastructure de l'université.

Vous devez donc faire tourner le serveur sur votre machine personnelle.

Sur votre tablette/mobile, installez l'application `Expo go` et fournissez lui l'adresse du serveur.

Vous pouvez également utiliser les serveurs d'`expo` pour construire un `APK Android`, installable sur votre device.

### 1.4.5 Gestion de la console de développement

Les erreurs de compilation/paquetage sont affichées dans le terminal qui a lancé la commande `npx expo` ou `npm run start`.

Si vous utilisez le navigateur pour tester votre projet, la console qui reçoit vos affichages `console.log(...)` est celle du navigateur.

Si vous testez sur tablette/mobile, la console d'affichage est le terminal qui a lancé la commande `npx expo` ou `npm run start`.

## 1.5 Analyse du composant principal

`expo` initialise le projet avec un fichier `App.js`, qui contient le code suivant :

```

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

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Open up App.js to start working on your app!</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});

```

Le code est écrit en JSX et mélange Javascript et XML. Lorsque l'on souhaite écrire du Javascript dans les éléments ou les attributs XML, il faut enrober le code avec des accolades.

Le code est divisé en trois parties :

1. les imports. On trouvera toujours l'import de la librairie React, puis ensuite les imports des différents composants React Native, utilisés dans l'application : ici `StyleSheet`, `Text`, `View`.
2. le code du composant, défini comme une fonction, pour l'instant sans argument. Cette fonction doit retourner du code JSX, entre parenthèses pour en déclencher l'évaluation.
3. la définition des styles. Attention, ce n'est pas du CSS mais du Javascript. Notez l'utilisation de `const styles = StyleSheet.create({...})`, ce qui permet d'utiliser ce style comme attribut dans un composant à l'aide de `style={styles.container}`.

Les styles sont amenés à être réutilisés, et l'emploi de `StyleSheet.create({...})` est une bonne pratique. On aurait cependant pu écrire directement le style dans l'attribut du composant :

```
<View style={{flex:1, ...}}>
```

(noter l'utilisation de doubles accolades : une accolade pour introduire Javascript, une autre pour indiquer que l'on fournit une liste de paires attribut/valeur).

On pourrait également mélanger style nommé et style énuméré :

```
<View style={[styles.container, {flex:1, ...}]}>
```



## 1.6 Un premier composant

Le développement en React Native consiste à créer ses propres composants et à construire des interfaces en appelant ces composants, grâce à du code XML intégré dans du Javascript.

Pour définir un composant personnalisé, on utilise le gabarit suivant :

```
import React from 'react';
import { <mettre ici les composants natifs à utiliser> } from 'react-native';

export default function MonComposant() {
  return (
    ...
  );
}

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

Le fichier est traditionnellement placé dans le dossier `components`, possède le nom du composant et l'extension `.js`, donc `components/MonComposant.js` pour notre exemple.

Dans le composant qui l'utilise (ici `App`), on l'importe comme suit :

```
import MonComposant from './components/MonComposant';
```

Puis on l'inclut dans la vue d'`App` :

```
export default function App() {
  return (
    <View style={styles.container}>
      ...
      <MonComposant/>
    </View>
  );
}
```

Concrètement, dans notre composant, on ajoute un bouton `Press me` qui déclenche un affichage dans la console :

```
export default function MonComposant(){
  return (
    <Button title='Press me' onPress={() => {console.log("pressé")}} />
  )
}
```

Si on regarde l'application sur le web, la console est celle du navigateur, sinon c'est dans le terminal qui fait tourner le serveur.

Noter qu'on a utilisé une fonction fléchée pour gérer l'événement d'appui. On aurait également pu écrire :

```
<Button title='Press me' onPress={function(){console.log("pressé")}} />
```

mais il faut s'habituer à écrire des fonctions fléchées.

On aurait également pu faire appel à une fonction nommée :

```
export default function MonComposant(){
  const onPressed = () => {
    console.log("pressé");
  }

  return (
    <Button title='Press me' onPress={onPressed} />
  )
}
```

ou encore :

```
export default function MonComposant(){
  const onPressed = (message) => {
    console.log(message);
  }

  return (
    <Button title='Press me' onPress={() => onPressed("pressé")} />
  )
}
```

## 1.7 Initiation à la gestion de l'état (*state*)

La gestion de l'état d'un composant est primordial en React : l'état est la liste de toutes les variables qui, si elles sont modifiées, vont induire une mise-à-jour graphique du composant.

Ici, nous allons ajouter un texte qui indique combien de fois on a appuyé sur le bouton, et l'on souhaite que cet affichage évolue lorsqu'on appuie sur le bouton.

Commençons par utiliser le *hook* (hameçon) `useState`. Lorsque les composants sont définis comme des fonctions, on aura accès aux *hooks*, qui permettent de se brancher sur les fonctionnalités React. Les hooks sont une nouveauté très récente de React.

Les composants fonction étant définis comme *stateless* (sans état), le hook `useState` permet de pallier ce manque. Au début de la définition du composant, on va définir une paire (*variable, fonctionPourModifierLaVariable*), en appelant `useState` avec la valeur initiale de la variable :

```
function monComposant(){
  const [count, setCount] = useState(0);
  ...
}
```

Pour modifier la valeur du compteur, il suffira d'appeler `setCount(<nouvelle valeur>)`.

Le composant final est :

```
import React, {useState} from "react";
import { Button, StyleSheet, Text, View } from "react-native";

export default function MonComposant() {

  const [count, setCount] = useState(0);

  return (
    <View>
      <Text>Le bouton a été pressé {count} fois</Text>
      <Button title='Press me' onPress={() => setCount(count + 1)} />
    </View>
  )
}
```

Noter que, comme le composant contient deux composants natifs (`Text` et `Button`), il a fallu les enrober dans un composant `View`, qui est l'équivalent de `div` en HTML. On aura également pu utiliser un *fragment*, qui groupe une liste d'enfants sans ajouter de nœud supplémentaire au DOM.

```
<React.Fragment>
  <Text>Le bouton a été pressé {count} fois</Text>
  <Button title='Press me' onPress={() => setCount(count + 1)} />
</React.Fragment>
```

ou, avec une syntaxe concise :

```
<>
  <Text>Le bouton a été pressé {count} fois</Text>
  <Button title='Press me' onPress={() => setCount(count + 1)} />
</>
```

## 1.8 Passage de propriétés (*props*)

Lorsqu'un composant utilise un autre composant, il peut lui passer des arguments, appelés propriétés ou *props*. Il suffit simplement de déclarer un argument à la fonction qui définit le composant. Par exemple, si on veut initialiser le compteur de notre composant :

```
export default function MonComposant(props) {

    const [count, setCount] = useState(props.count);
    ...
}
```

Dans `App.js`, on fait appel au composant comme suit :

```
<MonComposant count={10}/>
```

Attention, `<MonComposant count='10'/>` ne serait pas correct, car la valeur du compteur serait une chaîne de caractères, et l'opérateur `+` de Javascript effectuerait une concaténation.

Bien noter que le passage de *props* est unidirectionnel : du composant père vers le composant fils. Le contraire n'est pas possible : la modification des *props* n'impacte pas le composant père. Dit autrement : les props ne sont accessibles qu'en lecture uniquement par le composants fils, seul le composant parent peut les déterminer au moment de l'appel.

## 1.9 TP 1

### 1.9.1 Initialisation d'un projet React Native

- télécharger l'archive du projet avec le dossier `node_modules` dont les liens symboliques sont déréférencés
- la désarchiver dans `~/Documents`
- renommer le dossier selon votre choix
- ouvrir le dossier avec l'éditeur `codium`

### 1.9.2 Démarrage du projet

Une fois installé, il suffit de faire `npm run start` ou `npx expo`. Un serveur `expo` est lancé, vous permettant d'ouvrir l'émulateur web :

- appuyer sur `w`
- le projet est alors compilé avec `webpack`
- le navigateur est ouvert pour pointer sur `http://localhost:19006/`
- en cas de modification d'un fichier source, le projet est recompilé

### 1.9.3 Application de gestion de todos

On souhaite aller vers la réalisation d'une application qui gère une liste de tâches (*todo*).

1. créer un composant `TodoItem` dans le dossier `components` (le fichier doit être `TodoItem.js`) en utilisant le gabarit ci-dessous :

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

```
export default function TodoItem(props){
  return (
    <View>
    </View>
  )
}
```

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

2. importer le composant dans `App.js` :

```
import TodoItem from './components/TodoItem';
```

3. dans le composant fonction de `App.js`, faire appel au composant :

```
<TodoItem/>
```

4. dans le composant, ajouter un composant `Switch` qui est le composant natif analogue à une boîte à cocher et un texte. Utilisez la documentation sur les composants

5. pour que ces deux composants soient sur la même ligne, ajoutez l'attribut `flexDirection: 'row'` au composant `View` :

```
<View style={{flexDirection: 'row'}}>
```

6. ou créez un style :

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

```
...
```

```
const styles = StyleSheet.create({
  content: {
    flexDirection: 'row'
  }
})
```

7. améliorez la présentation en insérant un peu d'espace entre le switch et le texte, en utilisant du style à base de `margin` et `padding`. Utilisez la page des styles natifs pour vous aider.

8. faites en sorte que le composant père transmette un *item* dans les *props*, qui sera structuré :

```
<TodoItem item={{id:1, content:"laver le linge", done:false}}/>
```

9. utilisez dans le composant fils les valeurs fournies en *props*. Faites la différence entre

- ce qui concerne l'état du composant (le champ `done` influe sur l'état du switch)

- ce qui doit être récupéré dans les *props* pour définir une fois pour toutes le composant (le champ **content** ne variera pas)
10. utilisez **useState** pour moduler l'état du **Switch** :
    - la propriété **value** indique son état
    - la propriété **onValueChange** indique la fonction à appeler lorsque l'utilisateur modifie l'état, son paramètre est le nouvel état du **Switch**.
  11. Faites en sorte que le texte soit barré (style **textDecorationLine: 'line-through'**) selon l'état du **Switch**. Pour cela, utilisez *l'opérateur conditionnel* en Javascript :

```
<Text style={done ? {textDecorationLine: 'line-through'} : {}}>{props.item.content}</Text>
```

ou

```
<Text style={{textDecorationLine: done ? 'line-through' : 'none'}}>{props.item.content}</Text>
```

12. Si vous avez défini un style pour le texte, vous pouvez l'ajouter au style conditionnel comme suit :

```
<Text style={[styles.text_item, {textDecorationLine: done ? 'line-through' : 'none'}}>{props.item.content}</Text>
```

13. Utilisez cette icône de poubelle et ajoutez-la après le texte à l'aide du composant **Image**. Les fichiers d'image doivent être placés dans le dossier **assets** et le composant doit être utilisé comme suit (il faut mettre une taille sinon l'image n'apparaîtra pas dans l'interface web) :

```
<Image source={require('...')} style={{height:..., width:...}}/>
```

14. Rendez l'icône cliquable à l'aide d'un composant **TouchableOpacity** dont l'attribut **onPress** permet de paramétrer une action.

# Chapter 2

## Séance 2

### 2.1 Résumé de la Séance 1

- React Native utilise la librairie React et utilise les composants natifs du périphérique.
- `expo-cli` permet d'initialiser et de construire une application React Native, qui pourra être exécutée dans un navigateur ou sur un périphérique Android ou iOS. Le site <https://expo.dev> permet de partager des applications, fournit l'infrastructure pour les exécuter, les construire et les déposer sur les *stores*.
- pour exécuter une application React Native
  - sur le web : il faut utiliser un serveur Node.js
  - sur Android : il faut la construire à l'aide d'AndroidStudio
  - sur iOS : il faut la construire à l'aide d'XCode (disponible uniquement sur un ordinateur Apple)
- React Native permet de définir ses propres composants à partir des composants natifs.
- un composant personnalisé est une fonction qui renvoie du JSX, un mélange de Javascript et de XML.
- les composants fonctions peuvent utiliser le hook `useState` pour définir un état et le modifier.
- les composants pères peuvent utiliser des composants fils en leur fournissant des *props*, accessibles uniquement en lecture par le fils.

## 2.2 Communications fils -> père

Les communications père -> fils sont gérées par les props, ce sont des paramètres passés au fils au moment de sa création. Lorsqu'ils sont modifiés par le père, ils sont automatiquement mis à jour dans le fils.

Cependant, il est fréquent de vouloir modifier l'état du père depuis le fils. Par exemple, un bouton chez les fils modifie un compteur chez le père. Pour cela, on va utiliser les *props* pour que le père indique au fils quelle fonction appeler pour modifier son état.

Dans l'exemple ci-dessous, l'information est dynamiquement mise à jour, dans les deux sens : - père vers fils : la *props* `count` met à jour dynamiquement le composant `Text` du fils - fils vers père : l'appui sur le bouton appelle la *props* `onPressed`, qui met à jour l'état du père.

Dans le composant père, on définit une fonction qui sera appelée par le fils, à qui on la passe par les *props* :

```
export default function App() {
  const [count, setCount] = useState(0);

  const onPressed = () => {setCount(count + 1)}

  return (
    <View style={styles.container}>
      <Text>Père : le bouton a été pressé {count} fois</Text>
      <MonComposant onPressed={onPressed} count={count}/>
    </View>
  );
}
```

Dans le composant fils, on appelle cette fonction issue des props :

```
export default function MonComposant(props) {

  return (
    <>
      <Text>Fils : le bouton a été pressé {props.count} fois</Text>
      <Button title='Press me' onPress={props.onPressed} />
    </>
  )
}
```

## 2.3 Composant sous forme de classe

Les composants sous forme de classe sont obsolètes. Il faut désormais rédiger les composants sous forme de fonction, comme décrit lors de la première séance.



Cependant, il subsiste beaucoup de documentation sur le web qui utilise des composants sous la forme de classe. Ce document peut permettre de comprendre cette syntaxe.

## 2.4 TP 1

1. récupérer l'archive contenant le code de démarrage et l'intégrer au code du TP précédent, en modifiant les fichiers qui le nécessitent
2. analyser le code résultant
  - le composant `App` fait appel au composant `TodoList`
  - le composant `TodoList` import des données de `Todo`, contenues dans le fichier `Helpers/todoData.js`.

```
const todoData = [
  {
    id: 1,
    content: "faire ses devoirs",
    done: true
  },
  {
    id: 2,
    content: "ranger sa chambre",
    done: false
  },
  {
    id: 3,
    content: "essuyer la vaisselle",
    done: false
  }
];
```

```
export default todoData
```

3. le composant `TodoList` utilise ces données pour construire une liste plate constituée de composants `TodoItem` pour chaque item de la liste :

```
<FlatList
  style={{ paddingLeft: 10 }}
  data={todoData}
  renderItem={({item}) => <TodoItem item={item} /> } />
```

4. Une `Flatlist` a besoin de données et d'une propriété `renderItem`, qui indique quelle fonction sera affichée pour rendre chaque item. Notez bien la syntaxe utilisée dans les paramètres : `({item})`, ce qui permet dans `TodoItem` d'obtenir l'objet désiré dans `props.item`, et non `(item)`, ce qui obligerait à utiliser `props.item.item`.

5. Normalement, une `Flatlist` a besoin d'un attribut indiquant comment calculer les clés des items, par exemple `keyExtractor={ (item, index) => index }` ou `keyExtractor={ (item) => item.id }`. Dans le cas présent, c'est inutile car nos items ont un champ `id`.
6. le composant `TodoItem` est celui qui a été développé au TP précédent.
7. installez les modules : `npm install` ou `npm i`
8. démarrez l'application : `npm start`
9. ajoutez un texte dans le composant `TodoList` permettant d'afficher le nombre de `TodoItem` réalisés. Pour cela, il faut un compteur comme vu en cours et passer au composant fils les fonctions nécessaires à sa modification. Pour initialiser le compteur, on peut utiliser `todoData.filter((item) => item.done).length`.
10. Faites en sorte qu'un appui sur l'icône de poubelle détruise l'item correspondant.

Attention : la fonction de modification d'état est *asynchrone*. Vous ne pouvez donc pas enchaîner deux modifications d'état avec la deuxième utilisant l'état modifié par la première. Le code ci-dessous ne fonctionnera donc pas :

```
const deleteTodo = (id) => {
  setTodos(todos.filter(item => item.id !== id))
  setCount(todos.filter(item => item.done).length)
}
```

Sinon, ici nous allons simplement calculer la nouvelle liste des todos dans une variable temporaire, qui servira à appeler les deux changements d'état :

```
const deleteTodo = (id) => {
  const newTodos = todos.filter(item => item.id !== id)
  setTodos(newTodos)
  setCount(newTodos.filter(item => item.done).length)
}
```

11. ajoutez un `TextInput` et un bouton `new` pour ajouter un item à la liste. Le `TextInput` se paramètre comme suit :

```
<TextInput
  onChangeText={setNewTodoText}
  placeholder='saisir ici un nouvel item'
  onSubmitEditing={addNewTodo}
  value={newTodoText}
/>
```

L'attribut `onSubmitEditing` indique la fonction à appeler quand on valide l'input à l'aide de la touche *Entrée*. Lorsqu'on valide la saisie, il faut réinitialiser la valeur du `TextInput`, c'est pour cette raison que sa valeur est gérée par une variable d'état et que l'attribut `onChangeText` modifie l'état.

Pour que la `Flatlist` reste cohérente, les identifiants des items doivent rester différents. Pour cela, le nouvel item créé pourra avoir comme identifiant le

maximum des identifiants existants plus 1. La fonction `Math.max(...tableau)` retourne le maximum d'un tableau et pour construire le tableau des identifiants on utilise la fonction `map`, qui applique sa fonction paramètre à chaque élément d'un tableau :

```
Math.max(...todos.map(item => item.id)) + 1
```

Il faudra passer à `setTodos` un tableau constitué des anciens `todos` et d'un nouveau *todo*, ce qui peut se faire à l'aide de la syntaxe suivante (`...todos` signifie *le contenu du tableau todos*) :

```
setTodos([...todos, { id: monId, content: monContenu, done: false }])
```

- ajoutez des boutons pour filtrer l'affichage : n'afficher que les *todos* en cours, non résolus, ou *tout afficher*.



# Chapter 3

## Séance 3

### 3.1 Modification d'état suite à des modifications de *props*.

Sur notre application de gestion des *todo*, on souhaite ajouter un bouton permettant de changer l'état de **tous** les items, deux boutons en fait : **checkAll** et **checkNone**. Les fonctions associées vont modifier la liste des *todo*, c'est-à-dire les propriétés transmises aux composants **TodoItem** par les *props*.

Cependant, dans le composant **TodoItem**, le **Switch** est paramétré par un état et non dirigé par les *props*, sauf bien-sûr à la construction du composant. Il faudrait ici que la modification des *props* influe sur l'état.

Pour qu'un composant, qui reçoit ses props en paramètre, mette à jour son état lors d'une modification des props en provenance du père, il faut utiliser le *hook* **useEffect**.

Ce *hook* est constitué :

- d'une fonction à appeler
- d'une liste de variable dont la modification déclenchera l'appel à la fonction.

Il est ainsi possible de gérer le cycle de vie du composant.

- à chaque nouveau rendu

```
function MonComposant() {  
  useEffect(() => {  
    // fonction exécutée à chaque rendu du composant  
  });  
}
```

- à l'initialisation du composant

```
function MonComposant() {
  useEffect(() => {
    // fonction exécutée à l'initialisation du composant
  }, []);
}
```

- en cas de modification d'une variable

```
function MonComposant(props) {
  const [done, setDone] = useState(props.item.done)

  useEffect(() => {
    changeDone(props.item.done)
  }, [props.item.done])
}
```

C'est cette dernière possibilité que nous utilisons ici pour mettre en place une liaison entre les *props* et l'état.

### 3.1.1 Mise à jour du compteur de tâche

On aimerait conjointement mettre à jour le nombre d'items réalisés. On pourrait dans `componentDidUpdate` appeler la méthode de mise à jour du compteur chez le père. Ainsi, plusieurs composants `TodoItem` appelleraient cette méthode, quasiment tous en même temps, pour modifier le compteur du père. Cependant, cette méthode utilise l'accesseur d'état `setState`, qui est asynchrone. On court le risque de déclencher quasi simultanément plusieurs modification du *même* état, ce qui ne fonctionnera pas.

La solution ici consiste à éviter le problème : ne pas appeler la méthode de modification du père depuis le fils, mais introduire cette action directement chez le père.

```
const checkAll = () => {
  setTodos(todos.map(item => {id: item.id, content: item.content, done: true }))
  setCount(todos.length)
}
```

D'une façon générale, il ne faut pas trop compter sur React pour gérer correctement des communications père -> plusieurs fils -> père. C'est au programmeur de décider comment bien séparer les acteurs et savoir rester interne aux modifications du père si nécessaire.

## 3.2 Mise en page

Cette page liste tous les attributs des différents composants.

Attention : tous les composants ne disposent pas des mêmes aptitudes à être stylés. En particulier, le `textInput` devra être encapsulé dans une `view` pour pouvoir le décorer.

### 3.2.1 Utilisation de `flex`

C'est le mode de rendu par défaut des composants. C'est également le mode à privilégier, car on ne connaît jamais à l'avance la taille de l'écran. On pourra cependant choisir des tailles fixes lorsqu'elles sont petites.

Deux directions (et les sens associés) permettent d'empiler les composants :

- `flexDirection: 'column'` est la valeur par défaut, les composants s'empilent du haut en bas. Il existe aussi `column-reverse`.
- `flexDirection: 'row'` empile les composants de gauche à droite. Il existe `row-reverse`.

### 3.2.2 Répartition des tailles (*main axis*)

Le paramètre `flex`, un entier, indique la proportion de l'espace occupé **selon l'axe principal** (défini par `flexDirection`), relativement à la somme de tous les paramètres `flex` concernés.

Par défaut, `flex` vaut 0, les composants sont rendus en fonction de la taille qu'ils occupent. S'ils sont munis d'une taille fixe, c'est celle-ci qui sera utilisée. Si `flex` vaut -1, c'est la même chose sauf que les composants passent à leur taille minimale en cas de manque de place.

#### 3.2.2.1 Alignement selon la direction principale

C'est le rôle du paramètre `justifyContent`, dont la valeur peut être `flex-start`, `flex-end`, `center`, `space-between`, `space-around` ou `space-evenly`.

#### 3.2.2.2 Alignement selon la direction secondaire (*cross axis*)

C'est le rôle du paramètre `alignItems`. `alignSelf` permet de surcharger localement le `alignItems` du père.

**Attention :** si l'on souhaite que la répartition des composants prenne toute la place selon l'axe secondaire, il ne faut pas aligner selon cet axe, sinon les composants auront leur taille calculée sur cet axe. En particulier, lors de l'initialisation d'un composant par `expo`, le composant principal `App` est aligné au centre sur l'axe secondaire, ce qui empêche les composants de prendre toute la largeur de l'écran.

### 3.2.2.3 Démo

<https://reactnative.dev/docs/flexbox>      <https://medium.com/edonec/layout-with-flexbox-in-react-native-a24dbe678e75>      <https://blog.logrocket.com/a-guide-to-flexbox-properties-in-react-native/>

## 3.2.3 Utilisation de la taille de l'écran

Si la valeur de `flex` permet de procéder à des alignements automatiques selon la direction principale choisie, on peut parfois avoir besoin de mesurer la taille de l'écran pour dimensionner un élément :

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

const screen = Dimensions.get("screen");
const styles = StyleSheet.create({
  image: {
    height: screen.width * 0.8,
  }
  ...
});
```

### 3.2.3.1 Pour aller plus loin

Je vous conseille cet article qui explore les possibilités suivantes :

- regrouper ses styles dans un même fichier pour les exporter puis les importer où bon vous semble
- produire des styles dynamiques en les faisant retourner par des fonctions

## 3.3 TP 3

La liste de *todos* doit comporter des boutons pour

- cocher toutes les tâches
- décocher toutes les tâches
- n'afficher que les tâches cochées
- n'afficher que les tâches non cochées

## 3.4 TP Sudoku

*Ce TP est à effectuer sous forme de travail personnel.*

On voudrait afficher une grille de *sudoku*, constituée de trois *lignes* de 3 *carrés* contenant chacun 3 lignes de 3 cellules, contenant chacune un chiffre aléatoire entre 1 et 9.

1. initialiser l'application à l'aide `expo init sudoku`



2. commenter la ligne `alignItems: 'center'` dans le style du composant principal `App`.
3. créer des composants `Grid`, `Line`, `Square`, `LineSquare`, `Cell`.
4. utiliser les styles suivants pour visualiser la grille :

```
const styles = StyleSheet.create({
  grid:{
    flex: 1,
    flexDirection: '...',
  },
  line:{
    flex: 1,
    flexDirection: '...',
    borderWidth: 8,
    borderColor: 'lightgray',
  },
  square:{
    flex: 1,
    flexDirection: '...',
    borderWidth: 4,
    borderColor: 'gray',
  },
  lineSquare:{
    flex:1,
    flexDirection: '...',
    borderWidth: 2,
    borderColor: 'lightgreen',
  },
  cell:{
    flex:1,
    flexDirection: '...',
    borderWidth: 1,
    borderColor: 'green',
  }
})
```

5. compléter les styles avec les `flexDirections` adéquats
6. compléter le style d'une cellule pour que le chiffre qu'elle contient soit centré horizontalement et verticalement.
7. reprendre le code du TP précédent pour effectuer une mise en forme correcte de l'interface. # Séance 4

## 3.5 Navigation

Le but de cette séance est d'apprendre à concevoir un menu pour enchaîner des écrans. Traditionnellement sur mobile, une barre de navigation est disponible en bas de l'écran. L'appui sur les items de menus déclenche l'affichage de l'écran correspondant. On peut également trouver des icônes plutôt que des items textuels. Le concept est celui de `TabNavigator`.

### 3.5.1 Mise en oeuvre

Pour regrouper tous les éléments de navigation au sein d'un même composant, on crée un composant dans un dossier dédié : `Navigation/Navigation.js`.

```
import React from 'react'
import { View, Text } from 'react-native'
import { NavigationContainer } from '@react-navigation/native'
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs'

import TodoListsScreen from '../Screen/TodoListsScreen'
import HomeScreen from '../Screen/HomeScreen'
import SignInScreen from '../Screen/SignInScreen'
import SignOutScreen from '../Screen/SignOutScreen'

const Tab = createBottomTabNavigator()

export default function Navigation () {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        <Tab.Screen name='Home' component={HomeScreen} />
        <Tab.Screen name='TodoLists' component={TodoListsScreen} />
        <Tab.Screen name='SignOut' component={SignOutScreen} />
      </Tab.Navigator>
    </NavigationContainer>
  )
}
```

Le composant principal défini dans `App.js` fera simplement appel à la navigation :

```
export default function App () {
  return <Navigation />
}
```

La navigation est constituée:

- d'un container `NavigationContainer`

- d'un type de navigateur, ici `Tab.Navigator`, `Tab` ayant été initialisé comme un `createBottomTabNavigator()`
- d'écrans avec un label qui sera affiché et un composant qui sera appelé comme écran.

Lorsqu'on arrive sur un écran par le biais de la navigation, on dispose dans les *props* d'un champ `navigation` et d'un champ `route`. `navigation` permet de naviguer sur un autre écran comme suit :

```
export default function SignOutScreen ({ navigation, route }) {
  return <Button title='Sign me out' onPress={() => navigation.navigate('Home')} />
}
```

La `route` contient éventuellement des paramètres au moment de l'appel de l'écran :

```
// à l'appel de l'écran
<Button onPress={() => props.navigation.navigate('Details', {id: props.id})}>

// à l'arrivée sur l'écran
<Text>{ route.params.id }</Text>
```

### 3.5.2 Persistence de variables pendant la navigation

Il est utile de partager des variables entre les écrans de navigation. C'est la notion de *contexte*, qui permet de définir des variables *globales*, globales dans le sens où elles seront accessibles à chaque partie de l'arborescence des composants de l'application. Ce peut être le nom de l'utilisateur authentifié (ou un jeton), le thème ou la préférence de langue.

Les contextes sont à différencier des *props*. Les *props* sont transmises d'un composant *parent* à l'*enfant*. Le contexte peut quant à lui être partagé par des composants qui ne sont pas issus d'un parent commun.

Les contextes seront définis dans un fichier spécifique: `Contexte/Context.js`. On pourra définir un contexte particulier pour chaque variable à gérer :

```
import React from 'react';

export const TokenContext = React.createContext();

export const UsernameContext = React.createContext();
```

Ci-dessus nous avons défini deux contextes: l'un pour le jeton d'authentification auprès de l'API, l'autre pour le nom de l'utilisateur.

Les contextes sont accompagnés de deux fonctionnalités :

1. le `Provider`, qui donne la valeur au contexte
2. le `Consumer`, qui permet de lire la valeur du contexte

En général, le contexte sera géré de la même façon que l'on gère un *état* : on y stocke une paire (*valeur*, *setValeur*). C'est une base pratique que de récupérer cette paire à l'aide de `useState`.

Voici le composant principal dans `App.js`, pourvu d'un contexte et son `Provider` :

```
import { TokenContext, UsernameContext } from '../Context/Context'

export default function App () {
  const [token, setToken] = useState(null)
  const [username, setUsername] = useState(null)

  console.log('token', token)
  return (
    <UsernameContext.Provider value={[username, setUsername]}>
      <TokenContext.Provider value={[token, setToken]}>
        <Navigation />
      </TokenContext.Provider>
    </UsernameContext.Provider>
  )
}
```

Dans un écran de la navigation, on consommera le contexte comme suit :

```
export default function HomeScreen () {
  return (
    <UsernameContext.Consumer>
      {[username, setUsername]} => {
        return (
          <>
            <Text>Welcome !</Text>
            <Text>You are logged as {username}</Text>
          </>
        )
      }
    </UsernameContext.Consumer>
  )
}
```

Si besoin, il faudra emboîter plusieurs consommateurs :

```
export default function SignInScreen ({ navigation }) {
  return (
    <TokenContext.Consumer>
      {[token, setToken]} => (
        <UsernameContext.Consumer>
          {[username, setUsername]} => {
            ...
          }
        </UsernameContext.Consumer>
      )
    </TokenContext.Consumer>
  )
}
```

```

    }}
    </UsernameContext.Consumer>
  )}
  </TokenContext.Consumer>
)
}

```

Dans les composants fonction, on pourra préférer utiliser le *hook* `useContext`. Cela permet d'alléger le code :

```

export default function HomeScreen () {
  const [username, setUsername] = useContext(UsernameContext)
  return (
    <>
      <Text>Welcome !</Text>
      <Text>You are logged as {username}</Text>
    </>
  )
}

```

### 3.5.3 Navigation depuis un composant

Si le composant est défini comme un écran de la navigation, il est appelé avec l'objet de navigation dans les *props* :

```

export default function SignInScreen ({ navigation }) {
  ...
  // si besoin :
  navigation.navigate(...)
}

```

Si c'est un composant qui n'est pas un écran de navigation, il ne dispose pas de la navigation dans ses *props*. Il faudra donc éventuellement que l'écran ancêtre transmette la navigation dans les *props*.

Notez cependant que forcer la navigation est souvent superflu :

- si on modifie l'état d'une variable qui modifie la navigation, par exemple on récupère un jeton sur l'écran `SignIn`, ce qui bascule la navigation vers l'ensemble de pages accessibles aux utilisateurs enregistrés
- si on utilise les liens comme indiqué ci-dessous.

### 3.5.4 Imbrication de navigation

Il est tout à fait possible d'imprimer plusieurs navigation, par exemple :

```

function NavigationTodo () {
  return (
    <Stack.Navigator initialRouteName='List'>

```

```

        <Stack.Screen name='List' component={TaskListsScreen} />
        <Stack.Screen name='Details' component={TaskListDetailsScreen} />
      </Stack.Navigator>
    )
  }

export default function App(){
  return (
    <Tab.Navigator>
      <Tab.Screen name='Home' component={HomeScreen} />
      <Tab.Screen name='TodoLists' component={NavigationTodo} />
      <Tab.Screen name='SignOut' component={SignOutScreen} />
    </Tab.Navigator>
  )
}

```

### 3.5.5 Gestion des liens

En React Native, on peut définir des liens vers des écrans et leur passer des paramètres. Des détails ici

```

import { Link } from '@react-navigation/native';

// ...

function Home() {
  return (
    <Link to={{ screen: 'Profile', params: { id: 'jane' } }}>
      Go to Jane's profile
    </Link>
  );
}

```

### 3.5.6 Structuration finale du code

On évitera de mélanger les composants écran avec les autres, on mettra à part la navigation et la définition du contexte :

```

+-- App.js
+-- components
|   |-- SignIn.js
+-- Context
|   |-- Context.js
+-- Navigation
|   |-- Navigation.js
+-- Screen
|   +-- HomeScreen.js

```

```
|   --- SignInScreen.js
|   --- SignOutScreen.js
|   --- TodoLists.js
|   \-- TodoListsScreen.js
```

### 3.5.7 Liens

- <https://reactnavigation.org/>
- <https://www.bigbinary.com/learn-react-native/react-native-using-nested-navigators>
- <https://medium.com/swlh/react-native-nested-stack-navigation-bad56ea33820>
- <https://heartbeat.comet.ml/nesting-tab-and-stack-navigators-in-react-native-and-expo-apps-cc118a141e70>
- <https://javascript.plainenglish.io/react-native-navigation-8ccea9e14523>

## 3.6 Token d'authentification

Les applications modernes utilisent des API dites CRUD, qui permettent d'exposer un système d'information pour y envoyer, par exemple par HTTP, des requêtes pour créer (Create), lire (Read), modifier (Update) ou détruire (Delete) des contenus. Ces API peuvent être développées dans de nombreux langages, par exemple PHP ou Node.js pour répondre à une requête HTTP, et servent d'interface avec un *point d'accès* (*endpoint*) à la base de données.

La technologie GraphQL, parmi d'autres, fournit une spécification pour définir le schéma de données et effectuer des requêtes.

Lors de la réalisation d'une API CRUD, il n'est pas toujours possible de compter sur la disponibilité d'une *cookie* d'authentification. Il faut alors se tourner vers une autre technologie : les *jetons* d'authentification.

Ces jetons sont fournis par le serveur, qui va hacher un ensemble d'informations à l'aide d'une clé secrète. L'utilisateur récupère ce jeton lors de son inscription ou de sa connexion, et doit le transmettre lors de chaque opération pour s'authentifier. Lorsque le serveur reçoit un jeton, il lui est facile de le dé-hacher pour vérifier les informations sur l'utilisateur.

### 3.6.1 Les jetons JWT

Nous utilisons ici une technologie particulière de jeton : JWT (JSON web token). Lors de la connexion (signIn), on récupère le jeton :

```
mutation {
  signIn(username: "admin", password: "rootroot")
}
```

```
{
  "data": {
    "signIn": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxZmNlODk2Yy0zMmJiLTQyZj"
  }
}
```

Pour utiliser le jeton, il faut ajouter un entête HTTP à la requête GraphQL :

```
{
  "authorization": "Bearer eyJ0eXAi...j8"
}
```

Noter que le jeton est en trois parties, séparées par des points. Les deux premières consistent en un encodage **base64** des données et ne sont pas cryptées (il ne faut donc pas y mettre de données sensibles), la dernière est une signature par la clé, qui certifie que le jeton n'a pu être fourni que par le serveur. On peut effectuer le décodage le token à l'aide de <https://jwt.io/> (secret: dFt8QaYykR6PauvxcyKVXKauxvQuWQTc)

- header (type de token + algorithme de hachage)

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

- payload (la donnée véhiculée)
  - sub (l'id du souscripteur)
  - roles
  - jti (un id tiré au hasard)
  - iat (la date à partir de laquelle le jeton est valide)
  - exp (la date jusqu'à laquelle le jeton est valide)

```
{
  "sub": "1fce896c-36bb-42f4-aab2-14fb231f9031",
  "roles": [
    "admin"
  ],
  "jti": "285a66ba-f8a9-40a5-9e7d-b289fa8aecbd",
  "iat": 1634830627
}
```

### 3.6.2 Utilisation du jeton par React

Dans une application React, le jeton sera géré par un **Context**. Si le jeton n'est pas défini, il faudra forcer l'utilisateur à se rendre sur l'écran de connexion.

Pour cela, la navigation est construite à l'aide d'un test en ligne :

- la navigation consomme le contexte du jeton



- si le jeton n'est pas défini, on affiche les écrans `SignIn` et `SignUp`
- sinon on affiche la navigation standard

```
export default function Navigation () {
  return (
    <TokenContext.Consumer>
      {([token, setToken]) => (
        <NavigationContainer>
          {token == null ? (
            <Tab.Navigator>
              <Tab.Screen name='SignIn' component={SignInScreen} />
              <Tab.Screen name='SignUp' component={SignUpScreen} />
            </Tab.Navigator>
          ) : (
            <Tab.Navigator>
              <Tab.Screen name='Home' component={HomeScreen} />
              <Tab.Screen name='TodoLists' component={TodoListsScreen} />
              <Tab.Screen name='SignOut' component={SignOutScreen} />
            </Tab.Navigator>
          )}
        </NavigationContainer>
      )}
    </TokenContext.Consumer>
  )
}
```

### 3.6.3 API CRUD pour fournir le jeton

La machine virtuelle Neo4j + GraphQL fournit l'API CRUD.

En particulier, cette API répond aux mutations suivantes, qui permettent de se connecter ou de s'enregistrer :

```
mutation{signIn(username:"...", password:"...")}
mutation{signUp(username:"...", password:"...")}
```

Une autre façon d'écrire les mutations est de leur attribuer des variables paramètres :

```
'mutation($username:String!, $password:String!){signIn(username:$username, password:$password)}
```

La mutation doit être transmise en POST et accompagnée de la valeur des variables. Voici comment réaliser cela en Javascript avec l'API `fetch`, disponible en standard :

```
const API_URL = 'http://graphql.unicaen.fr:4000'

const SIGN_IN = `
mutation SignIn($username: String!, $password: String!) {
```

```

    signIn(username: $username, password: $password)
  }
}

export function signIn(username, password) {
  return fetch(API_URL, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      query: SIGN_IN,
      variables: {
        username: username,
        password: password
      }
    })
  })
  .then(response => {
    return response.json()
  })
  .then(jsonResponse => {
    if (jsonResponse.errors !== null) {
      throw jsonResponse.errors[0]
    }
    return jsonResponse.data.signIn
  })
  .catch(error => {
    throw error
  })
}

```

Il est important de noter que :

- on appelle `fetch` sur une URL
- avec un objet
  - un champ `method`
  - des précisions sur l'entête
  - un corps contenant
    - \* une requête
    - \* la valeur des variables
- `fetch` est une promesse **qu'il faut retourner**. Dans le code qui utilise la fonction `signIn`, il faudra également écrire une promesse :

```

signIn(login, password)
  .then(token => {
    setToken(token)
  })

```

```

      setUsername(login)
      props.navigate('Home')
    })
    .catch(err => {
      setError(err.message)
    })
  })

```

- lorsqu'une promesse réussit, elle enchaîne avec le **then** qui appelle la fonction fléchée qu'il contient avec comme paramètre ce qui a été retourné par le **then** de la promesse précédente.
- lorsqu'une promesse échoue, elle enchaîne avec le **catch** qui appelle la fonction fléchée qu'il contient avec comme paramètre ce qui a été propagé (avec **throw**) par la promesse précédente.
- le premier **then** du **fetch** reçoit la réponse, qui est retournée en JSON
- le deuxième **then** reçoit donc la réponse au format JSON
- en Javascript, une erreur est un objet, que l'on construit avec **new Error(message)**. On affiche une erreur sous forme de chaîne de caractères à l'aide de **error.message**.

### 3.7 TP 4

1. concevez une application qui contient les écrans suivants, si le jeton existe :
  - Home : un message 'Welcome ! You are logged as...'
  - TodoLists : un message 'Liste des TodoList'
  - SignOut : l'appui sur un bouton remet à zéro le jeton et le nom de l'utilisateur. Notez qu'il n'est pas nécessaire de forcer la navigation à retourner à l'écran de SignIn : React détecte le changement de jeton et met automatiquement à jour l'application.
2. ou si le jeton n'existe pas :
  - SignIn : un **TextInput** pour le nom et un autre pour le mot de passe avec l'attribut **secureTextEntry={true}**. Ajouter un bouton pour la soumission. Dans un premier temps, définir un jeton factice et rediriger vers la page d'accueil.
  - SignUp : même chose
3. mettre en place l'API pour récupérer le jeton. Testez les différents cas d'erreurs :
  - le serveur de l'API ne fonctionne pas -> erreur de réseau
  - le mot de passe n'est pas correct
  - le login n'est pas correct



## Chapter 4

## Séance 5

### 4.1 Résumé de la séance précédente

Nous avons découvert :

- comment concevoir une navigation avec un menu
- comment proposer une navigation optionnelle, par exemple selon que l'utilisateur est connecté ou non
- les jetons JWT d'authentification pour une API. Ces jeton contiennent l'identifiant de l'utilisateur et son signé par le serveur : ils sont infalsifiables
- nous avons réalisé une première brique de l'API : la connexion qui renvoie un jeton. Cette fonction est une *promesse*, réalisée en asynchrone et permet de traiter les erreurs.

### 4.2 Conception générale d'une application React

*Au fil de cette présentation, nous considérerons la réalisation d'une application de gestion de TodoList, en liaison avec une API CRUD appelée par des requêtes GraphQL.*

Lorsqu'on conçoit une application complète, il convient d'organiser les différents flux d'informations entre les différents composants :

1. la première chose à décider concerne la réalisation du contexte, qui contient les variables globales, nécessaires pour la majorité des écrans, indépendamment de leur position dans la hiérarchie des composants. Ici, notre contexte contiendra le jeton et le nom de l'utilisateur.
2. ensuite, il faut déterminer les variables d'état et les composants qui accueilleront leur définition. Ces variables d'état (et leurs accesseurs)

pourront potentiellement être transférées à des composants enfants par l'intermédiaire des *props*. Elles doivent donc être définies dans les composants le haut dans l'arbre des composants, en général dans les écrans.

3. déterminer quels composants doivent être des fonctions ou des classes. L'intérêt des classes est d'avoir un état interne, un cycle de vie du composant et des méthodes auxiliaires qui permettent de gérer le composant. En général, l'utilisation des *hook* d'état, de contexte et d'effet rendent inutile l'utilisation d'une classe.

Pour cette conception, on pourra s'organiser les idées en représentant la hiérarchie des composants.

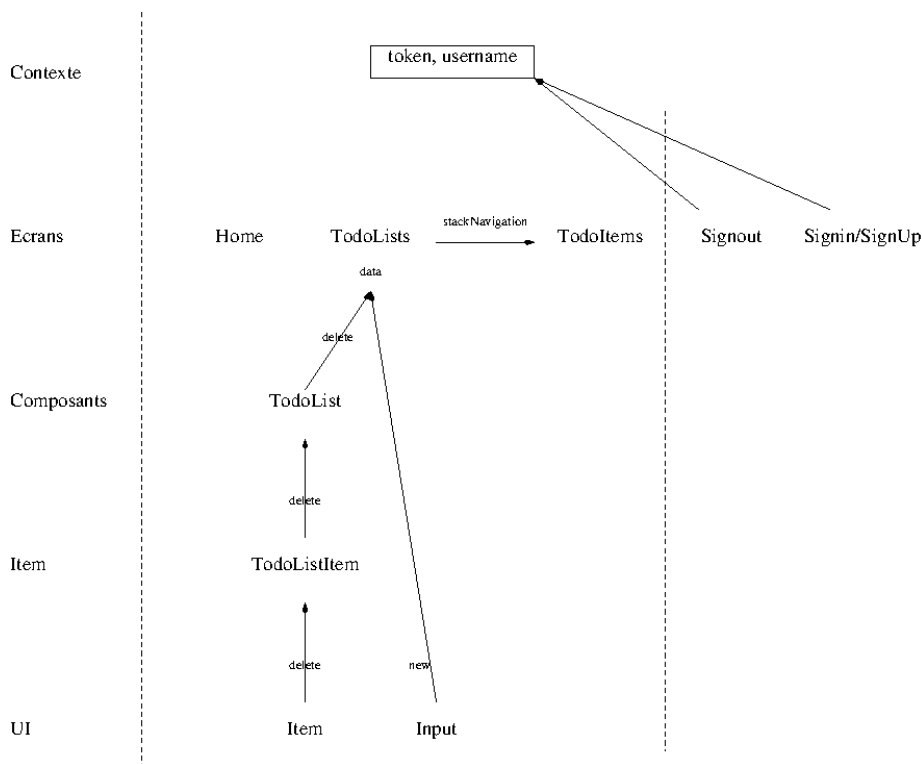


Figure 4.1: Hiérarchie des composants

### 4.3 Hook d'effet

Le *hook* d'effet `useEffect` est appelé par React à chaque mise à jour du DOM. Il permet d'éviter d'écrire les composants comme des classes, qui possèdent des méthodes de gestion du cycle de vie comme `componentDidMount`.

`useEffect` permet de mettre à jour l'état du composant en cas d'*effet de bord*, par exemple un appel initial à l'API pour charger la liste des `ToDoList`. `useEffect` peut être pourvu d'un tableau de variables d'état. Il ne sera exécuté qu'en cas de modification de ces variables.

```
useEffect(() => {
  if (data.length == 0) {
    console.log('chargement initial')
    ...
  }
}, [data])
```

Il est possible d'avoir plusieurs hooks `useEffect` pour gérer différents effets.

## 4.4 TP 5

Les tâches de ce TP sont à traiter dans l'ordre qui vous convient. En fin de séance, vous devez avoir un écran de listant les `ToDoList`, permettant de les détruire individuellement et d'en ajouter une nouvelle.

- reprenez votre application du TP précédent
- sur l'écran de la liste des `ToDoList`, faites une requête sur l'API pour obtenir la liste (utilisez `useEffect()` pour déclencher ce chargement au moment où le composant est créé). La requête GraphQL est :

```
query TodoLists($where: ToDoListWhere) {
  todoLists(where: $where) {
    id
    title
  }
}
```

- ajoutez un composant générique `Input`, que vous stockerez dans `components/UI/`, qui contient un `TextInput` et un bouton de soumission. Inspirez-vous du code utilisé dans les TP précédents. Déterminer quels arguments ce composant doit recevoir par ses *props*. La soumission de cette entrée déclenchera l'enregistrement en base de données d'une nouvelle `ToDoList`, à l'aide de la requête GraphQL suivante :

```
mutation createTodoLists($input: [ToDoListCreateInput!]) {
  createTodoLists(input: $input) {
    todoLists {
      id
      owner {
        username
      }
      title
    }
  }
}
```

```
}  
}~
```

- faites en sorte qu'un icône de poubelle soit placée à droite du titre d'une liste et que le click sur la poubelle détruise l'item. Vous pouvez réaliser un composant générique en vous inspirant du code des TP précédents, qui contienne une liste de texte suivie de l'icône de poubelle.



# Chapter 5

## Séance 6

### 5.1 Construire l'application

Lorsque l'application est stable, il est nécessaire de la *construire* (ou emballer) pour la déployer sur un périphérique mobile. Il s'agit de générer un fichier au format `apk`.

### 5.2 Utilisation d'expo

On peut utiliser `expo` pour construire l'application :

```
$ expo build:android -t apk
```

Après avoir répondu à quelques questions (nom du package, clé de chiffage), la construction est lancée sur les serveurs d'`expo`. Le temps d'attente est variable selon leur occupation.

Le gros intérêt de cette pratique est qu'il n'est pas nécessaire de modifier le code de l'application s'il a été initié à l'aide d'`expo`.

Des détails ici

Cependant, utiliser `expo` procure de nombreux désagréments, voir par exemple ce post en français.

### 5.3 Utilisation d'androidstudio

Utiliser `androidstudio` pose quelques difficultés :

- ce produit est très lourd et requiert de nombreux modules tout aussi lourds
- l'émulateur tourne sur une machine virtuelle incompatible avec Virtualbox

Cependant, `androidstudio` est utile pour installer l'émulateur. Une fois cette étape réalisée, on peut quitter `androidstudio`, il n'est plus nécessaire.

En particulier, **il vaut mieux éviter de mettre à jour `gradle` comme le propose `androidstudio`.**

Des détails ici

Détails de l'installation d'`androidstudio` :

## 5.4 Procédure de développement classique, sans expo

1. initialiser l'application à l'aide de

```
react-native init <nomProjet>
```

2. *éjecter expo* : il s'agit de se libérer du framework. À ce stade, on recopiera simplement dans le nouveau dossier du projet créé par `react-native` les fichiers et dossiers de l'application gérée par `expo`. Il faut éventuellement mettre à jour le fichier `package.json` et bien sûr `App.js`.

3. ajouter les modules pour la navigation

```
npm i --save react-native-safe-area-context
npm i --save react-native-screens
```

4. dans le dossier de l'application, lancer l'empaqueteur `metro` :

```
react-native start
```

5. lancer l'émulateur android (il faut parfois insister en relançant la commande, car l'émulateur prend du temps pour s'initialiser) :

```
react-native run-android
```

Cela construit un `.apk` de développement dans le dossier `android/app/build/outputs/apk/debug`.

## 5.5 Génération de l'apk *debug*

1. empaqueter le Javascript de l'application (*bundle*)

```
mkdir android/app/src/main/assets
react-native bundle --platform android --dev false --entry-file index.js --bundle-output
```

2. On peut également explicitement construire l'apk :

```
cd android
./gradlew assembleDebug
```

Pour cette dernière étape, il n'est pas nécessaire d'avoir lancé `androidstudio` (mais il faut avoir fermé `Virtualbox`).

On peut cependant :

- lancer `androidstudio`
- ouvrir comme projet le dossier `android`
- sélectionner le bon périphérique et exécuter le projet : il s'exécute dans l'émulateur à l'intérieur d'`androidstudio`.

## 5.6 Génération d'un .apk *release*

<https://medium.com/geekculture/react-native-generate-apk-debug-and-release-apk-4e9981a2ea51>

1. générer une clé de signature :

```
keytool -genkey -v -keystore your_key_name.keystore -alias your_key_alias -keyalg RSA -keysize 2048
```

Un fichier `your_key_name.keystore` est généré. Le positionner dans le dossier `android/app` :

```
mv your_key_name.keystore android/app/
```

2. éditer le fichier de configuration de `gradle` comme suit :

```
signingConfigs {
    debug {
        ...
    }
    release {
        storeFile file('your_key_name.keystore')
        storePassword 'password'
        keyAlias 'your_key_alias'
        keyPassword 'passwd (le même)'
    }
}
buildTypes {
    debug {
        signingConfig signingConfigs.debug
        if (nativeArchitectures) {
            ndk {
                abiFilters nativeArchitectures.split(',')
            }
        }
    }
    release {
        // Caution! In production, you need to generate your own keystore file.
        // see https://reactnative.dev/docs/signed-apk-android.
        signingConfig signingConfigs.release    // <- changer ici debug en release
        ...
    }
}
```

```
}
```

3. autoriser l'application à faire des échanges avec l'API. Dans `android/app/src/main/AndroidManifest.xml` ajouter un attribut à la balise `application` :

```
<application
  ...
  android:usesCleartextTraffic="true"
```

4. paqueter l'application

```
mkdir android/app/src/main/assets
react-native bundle --platform android --dev false --entry-file index.js --bundle-output
```

Voir également ce post.

5. construire l'apk

```
cd android
./gradlew assembleRelease
```

6. s'il y a des problèmes avec des ressources dupliquées : les détruire !

```
rm -f app/src/main/res/drawable-*/*
```

L'apk construit est dans `app/build/outputs/apk/release/app-release.apk`.

## 5.7 Conclusion

Pour la même application :

- APK expo : 64Mo, 132 Mo de stockage interne
- APK debug : 42 Mo (dont 1 Mo de bundle Javascript), 65,41 Mo de stockage interne
- APK release : 32 Mo, 63,28 Mo de stockage interne

`expo` est un bon choix s'il l'on souhaite travailler rapidement, mais l'apk produit est très lourd.

La démarche classique (pas d'`expo`) est à privilégier en contexte professionnel. Il faudra alors **dès le début** travailler en condition réelles et tester la génération d'apk à chaque étape du développement.

## 5.8 Liens / tips

<https://medium.com/reactbrasil/being-free-from-expo-in-react-native-apps-310034a3729>  
<https://apiko.com/blog/expo-vs-vanilla-react-native/>  
<https://fulcrum.rocks/blog/react-native-init-vs-expo/>

```
npm i react-native-screens --save
npm install @react-navigation/native --save
```

```
npm install react-native-reanimated react-native-gesture-handler react-native-screens react-native-safe-area-context
```



# Chapter 6

## Séance 7

### 6.1 Retour sur les fonctions asynchrones

Nous avons expérimenté l'utilisation de fonctions asynchrone, dont on connaît le début d'exécution mais pas la fin. C'est par exemple le cas du *hook* de changement d'état `useState`, qui modifie la variable d'état, mais on ne sait pas quand cette modification est effective.

Souvent, les fonctions asynchrones prennent en paramètre une fonction de rappel (*callback*) qui est appelée lorsque la fonction asynchrone a terminé sa tâche. Pour des raisons de lisibilité du code (et éviter les callback qui prennent en argument des callback, etc.), on préfère utiliser des promesses : la fonction principale *retourne* une valeur qui est utilisée dans la section `then` ou lance une exception qui est captée dans la section `catch`.

C'était le cas par exemple lors de l'accès à l'API `fetch` :

```
function signIn(username, password){
  return fetch(...)
    .then(response => {
      return response.json()
    })
    .then(jsonResponse => {
      if (jsonResponse.errors != null) {
        throw jsonResponse.errors[0]
      }
      return jsonResponse.data.signIn
    })
    .catch(error => {
      throw error
    })
}
```

La fonction `signIn` qui utilise une promesse doit être traitée comme une promesse :

```
signIn(login, password)
  .then(token => {
    setUsername(login)
    setToken(token)
  })
  .catch(err => {
    setError(err.message)
  })
```

Détaillons maintenant deux mécanismes :

- utilisation des mots clés `async` et `await` lors de l'écriture de fonction asynchrone
- l'écriture de promesses personnalisées

### 6.1.1 `async` et `await`

`async` permet d'indiquer que la fonction écrite est asynchrone.

Par exemple, définissons la fonction simple suivante, qui renvoie `Bonjour` :

```
function hello() { return "Bonjour" };
undefined
hello()
"Bonjour"
```

Si l'on utilise `async`, la fonction renvoie désormais une promesse :

```
async function hello() { return "Bonjour" };
undefined
hello()
Promise { <state>: "fulfilled", <value>: "Bonjour" }
```

On peut enchaîner `hello()` avec `then`, qui renvoie une autre promesse :

```
hello().then(d => console.log(d))
Bonjour debugger eval code:1:27
Promise { <state>: "fulfilled", <value>: undefined }
```

D'autres écritures sont possibles :

- une *expression de fonction asynchrone* :  

```
let hello = async function() { return "Bonjour" };
```
- une fonction fléchée :  

```
let hello = async () => { return "Bonjour" };
```



Le mot clé `await`, placé devant du code asynchrone, n'est utilisable que dans une fonction asynchrone. Cela demande à l'interpréteur Javascript d'attendre la résolution du code asynchrone avant de passer à la ligne suivante. Cela peut être pertinent, par exemple pour des raisons de sécurité ou d'autorisation.

Par exemple, pour la fonction `signIn` précédente, on peut écrire :

```
async function signIn(username, password){
  let response = await fetch(...)
  if (response.json().errors != null)
    throw response.json().errors[0]
  return response.json().data.signIn
}
```

Le code obtenu est plus clair mais également moins efficace : l'exécution est suspendue pendant que JavaScript attend l'issue du code asynchrone. Or on cherche à éviter ce genre de blocage, car c'est la raison d'être du code asynchrone : permettre de faire autre chose pendant l'attente. Il faut donc éviter l'utilisation de `await` (sauf si c'est vraiment ce que l'on souhaite).

**Attention !** il ne sert à rien de mettre un `await` avant un `setState`. Si on veut être sûr d'appeler une fonction *après* le changement d'état, il faut utiliser un `useEffect` guétant la modification de l'état. Voir ce post pour en comprendre la complexe raison, liée aux *fermetures*.

### 6.1.2 Écriture de promesses personnalisées

Une promesse est un objet qui représente l'état intermédiaire d'une opération. Elle permet de reporter d'autres actions jusqu'à ce qu'une action précédente soit terminée ou échoue. Elles sont au coeur de la réalisation d'opérations asynchrones.

Une promesse se construit à l'aide du constructeur `Promise`, qui prend en paramètre une fonction à deux fonctions paramètres, `resolve` et `reject`, qui seront appelées pour réaliser ou rejeter la promesse. Lors de l'utilisation de la promesse, `resolve` sera la fonction incluse dans la section `then`, `reject` sera la fonction incluse dans la section `catch`.

Par exemple, écrivons une promesse qui patiente 2 secondes et renvoie 'OK' à `resolve` :

```
let timeoutPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('OK');
  }, 2000);
});
undefined
timeoutPromise
Promise { <state>: "fulfilled", <value>: "Succès !" }
```

*Attention : cela ne fonctionne pas correctement dans la console du navigateur, il faudra intégrer le code dans une page HTML.*

### 6.1.3 Lien

- <https://devtrium.com/posts/async-functions-useeffect>
- [https://developer.mozilla.org/fr/docs/Learn/JavaScript/Asynchronous/Promises#construire\\_vos\\_propres\\_promesses\\_personnalis%C3%A9es](https://developer.mozilla.org/fr/docs/Learn/JavaScript/Asynchronous/Promises#construire_vos_propres_promesses_personnalis%C3%A9es)

## 6.2 Retour sur les *hooks*

### 6.2.1 Hook d'état

Le mutateur d'état prend en général en paramètre la nouvelle valeur pour l'état. On peut cependant lui passer une fonction dont le paramètre est l'ancienne valeur de l'état.

En reprenant l'exemple de la documentation :

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Total : {count}
      <button onClick={() => setCount(initialCount)}>Réinitialiser</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

### 6.2.2 Hook de contexte

Il est plus agréable d'utiliser le *hook* `useContext`, dont le fonctionnement est similaire au *hook* d'état, plutôt que les syntaxes alourdies du type `<MyContext.Consumer>...`

Attention cependant, `useContext` est équivalent à `<MyContext.Consumer>...`, permet de lire le contexte et de s'abonner à ses modification. Mais **on ne peut pas se dispenser d'un `<MyContext.Provider>` plus haut dans l'arbre.**

## 6.3 Fonctionnalités additionnelles

### 6.3.1 Activity Indicator

Le composant `ActivityIndicator` fournit une animation graphique destinée à montrer à l'utilisateur que l'application est en activité. Cela permet de patienter

pendant un chargement potentiellement long.

Il est donc d'usage de gérer un booléen `isLoading` indiquant que les constituants d'un composant sont en cours de chargement (accès à l'API). L'`ActivityIndicator` sera affiché tant que le chargement est en cours.

```
export default function TodoLists () {
  const [isLoading, setIsLoading] = useState(true)
  ...

  useEffect(() => {
    getTaskLists(username, token)
      .then(data => {
        setData(data)
        setIsLoading(false)
      })
      .catch(error =>
        showMessage({
          message: 'API Error: ' + error.message,
          type: 'error'
        })
      )
  })
  }, [])

  return (
    {isLoading ? (
      <ActivityIndicator />
    ) : (
      <TaskLists
        data={data}
        username={username}
        token={token}
        delete={removeTaskList}
        navigation={navigation}
      />
    )}
  )
}
```

Noter que, pour tester cette fonctionnalité, il peut être utile de ralentir les communications avec l'API. Les outils de développement du navigateur permettent pour cela de limiter la bande passante.

### 6.3.2 Utilisation des fonctionnalités du téléphone

Les téléphones et tablettes disposent d'un appareil photo, d'un système GPS, etc. On examine dans la suite les possibilités offertes par **React Native** d'utiliser dispositifs.

Deux types de modules sont disponibles :

- des modules développés par la communauté : <https://reactnative.directory>
- des modules proposés par Expo.

Pour avoir passé du temps à mettre en place les modules de la communauté, je conclus que pour des développements individuels, il vaut mieux utiliser les propositions d'Expo : elles sont instantanément fonctionnelles, ce qui n'est pas le cas des modules de la communauté, qu'on n'utilisera que dans un contexte professionnel où le temps de mise-au-point est disponible.

#### 6.3.2.1 Lecteur de QR code

Examinons le code disponible à <https://docs.expo.dev/versions/latest/sdk/bar-code-scanner/>

```
export default function App() {
  const [hasPermission, setHasPermission] = useState(null);
  const [scanned, setScanned] = useState(false);

  useEffect(() => {
    (async () => {
      const { status } = await BarcodeScanner.requestPermissionsAsync();
      setHasPermission(status === 'granted');
    })();
  }, []);

  const handleBarCodeScanned = ({ type, data }) => {
    setScanned(true);
    alert(`Bar code with type ${type} and data ${data} has been scanned!`);
  };

  if (hasPermission === null) {
    return <Text>Requesting for camera permission</Text>;
  }
  if (hasPermission === false) {
    return <Text>No access to camera</Text>;
  }

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

```

    <BarcodeScanner
      onBarcodeScanned={scanned ? undefined : handleBarcodeScanned}
      style={StyleSheet.absoluteFillObject}
    />
    {scanned && <Button title={'Tap to Scan Again'} onPress={() => setScanned(false)} />} />
  </View>
);
}

```

Noter que la fonctionnalité ne se lance qu'après le `useEffect`, lui-même bloqué par le `await` qui attend que les permissions soient accordées.

### 6.3.2.2 GPS

Expo propose un module GPS : <https://docs.expo.dev/versions/latest/sdk/location/#locationobjectcoords>

```

export default function App() {
  const [location, setLocation] = useState(null);
  const [errorMsg, setErrorMsg] = useState(null);

  useEffect(() => {
    (async () => {
      let { status } = await Location.requestForegroundPermissionsAsync();
      if (status !== 'granted') {
        setErrorMsg('Permission to access location was denied');
        return;
      }

      let location = await Location.getCurrentPositionAsync({});
      setLocation(location);
    })();
  }, []);

  let text = 'Waiting..';
  if (errorMsg) {
    text = errorMsg;
  } else if (location) {
    text = JSON.stringify(location);
  }

  return (
    <View style={styles.container}>
      <Text style={styles.paragraph}>{text}</Text>
    </View>
  );
}

```

Remarquer là encore le `useEffect` bloqué lors de l'attente de permissions.