Marquer comme terminé

`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.

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.

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
```

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.

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'
```

Affectation de multiples variables

à partir d'un objet structuré

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

équivaut à

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

à partir d'un tableau

C'est l'affectation par décomposition\ :

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

équivaut à

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

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 *syntaxiquement*, c'est-à-dire que c'est le `this` de la définition de la méthode.

Plus de détails

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}`);
```

?

Initialisation d'une application

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 déférencés
- la désarchiver dans `~/Documents`
- renommer le dossier selon votre choix

Sur votre machine personnelle

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

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

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.

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 reseau 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.

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 recoît 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`.

Analyse du composant principal

`expo` initialise le projet avec une 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, ...}]}>
```

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")} />
    )
}
```

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) `setState`. 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 falu 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)} />
        </>
```

?

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.

Modifié le: lundi 9 septembre 2024, 11:22

◄ Projet React Native vide

Choisir un élément

Aller à...

trash-can-outline.png ►