

Les composants React Native

Table des matières

I. Contexte	3
II. Qu'est-ce qu'un composant ?	3
III. Exercice : Appliquez la notion	6
IV. Le JSX et la notion de parent/enfant	6
V. Exercice : Appliquez la notion	8
VI. Le cycle de vie des composants	8
VII. Exercice : Appliquez la notion	14
VIII. Essentiel	15
IX. Auto-évaluation	15
A. Exercice final	15
B. Exercice : Défi.....	16
Solutions des exercices	17

I. Contexte

Durée : 1 h

Environnement de travail : Une application React Native initialisée, ou utiliser <https://snack.expo.io/>

Pré-requis : Introduction à l'outil Expo

Contexte

Pour concevoir des applications React Native, il faut surtout écrire du code JavaScript. React Native étant très lié et proche de son homologue React Web, il reprend le concept de composants réutilisables, et leur composition.

Les composants permettent de factoriser des bouts de code pour les réutiliser dans l'application, ils possèdent un cycle de vie qui leur est propre et permettent de gérer les effets de bord.

Ils permettent notamment de définir des éléments autonomes (comme un bouton ou une carte permettant d'afficher la présentation d'un plat d'un restaurant) et de pouvoir les réutiliser très simplement dans une application en leur passant simplement des paramètres.

Ils peuvent s'écrire grâce à plusieurs syntaxes différentes, comme les classes ou des fonctions, et c'est ce que nous allons voir dans ce cours.

II. Qu'est-ce qu'un composant ?

Objectifs

- Comprendre ce qu'est un composant React
- Voir les différents types de composants
- Étudier la syntaxe pour l'utilisation des composants

Mise en situation

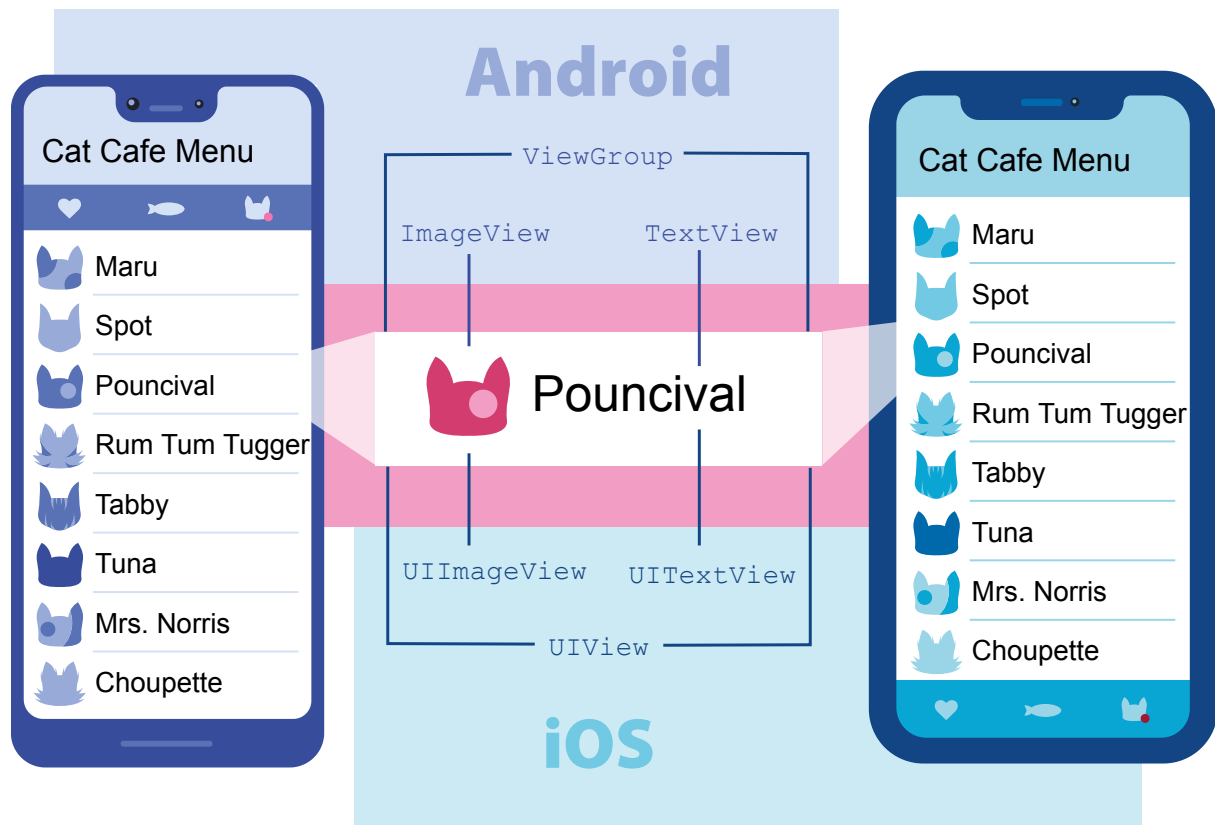
Un composant dans React est une brique logicielle réutilisable qui permet de factoriser du code métier en fournissant une abstraction. C'est l'élément de base d'une application React et, en les assemblant, on peut former une application entière.

Concrètement, un composant est une classe JavaScript ou une fonction qui reçoit des arguments (les props) et qui affiche ou déclenche des effets de bord par rapport aux valeurs de ces arguments.

Pour React Web, il n'y a pas réellement (ou très peu) de composants de base fournis. La plupart du code est produit à base de balises HTML, comme une `div` ou un `p`, ces éléments étant communs à tous les navigateurs web.

Cependant, avec React Native, le framework est obligé de fournir une abstraction qui s'adaptera en fonction de la plateforme Android ou iOS au moment où le moteur d'interprétation (bridge) va la rencontrer.

Par exemple, comme le montre ce schéma issu de la documentation React Native, le composant `Text` de React Native permettra de créer un élément `UiTextView` sur iOS, et un élément `TextView` sur Android. Le composant `Image`, lui, créera respectivement un élément `UIImageView` sur iOS et `ImageView` sur Android.



Remarque

React Native fournit donc un certain nombre de composants par défaut afin de faire abstraction de la plateforme de rendu finale. Si jamais celui que l'on souhaite n'existe pas, nous pourrions ajouter les nôtres avec un peu de code natif (et le *workflow bare* d'Expo) ou regarder dans le monde *open source* si quelqu'un ne l'a pas déjà fait.

Créer un composant, mais pour quoi faire ?

Maintenant que nous avons connaissance de certains composants de base, on peut en retrouver la liste exhaustive à cette adresse : <https://reactnative.dev/docs/components-and-apis>. En utilisant ces composants, on peut les combiner pour pouvoir générer un nouveau composant qui sera réutilisable.

Par exemple, imaginons que nous avons toujours le même bouton, avec le même design et le même fonctionnement. Plutôt que de répéter du code, il serait intelligent de créer une brique réutilisable pour traiter ce cas d'utilisation. Un composant s'y prête parfaitement.

Écrire un composant : plusieurs syntaxes

Pour rédiger les composants, on peut être amené à observer plusieurs syntaxes. Concrètement, les composants basés sur les classes sont la manière la plus ancienne de procéder et peuvent dans 99 % des cas être remplacés par des composants basés sur des fonctions. Cependant, il faudra les coupler avec des React hooks.

Les composants basés sur les classes

Pour définir un composant de classe, on utilise la syntaxe suivante :

Exemple

```
1 import * as React from 'react';
2 import { Text, View } from 'react-native';
3
4 class App extends React.Component {
5   constructor(props) {
6     super(props);
7
8     this.state = { name: 'Jean' };
9     this.sayHowAreYou = this.sayHowAreYou.bind(this); // On s'assure que this soit bien
10  }
11
12  sayHowAreYou(name) {
13    return <Text>Comment ça va {name}</Text>;
14  }
15
16  render() {
17    return (
18      <View>
19        <Text>Coucou {this.state.name}</Text>
20        {this.sayHowAreYou(this.state.name)}
21      </View>
22    );
23  }
24 }
```

Essayons de jouer avec ce morceau de code dans notre application. Rapidement, on peut observer une méthode `render` qui contient la partie la plus importante : la définition `JSX` (que nous allons voir plus loin dans ce cours).

On peut accéder à l'ensemble des éléments via le mot-clé `this`, tels que le `state` et les `props`. Pas d'inquiétude si toutes ces notions ne semblent pas claires, elles sont abordées dans un autre chapitre.

Les composants basés sur les fonctions

Si l'on souhaite écrire l'équivalent du composant plus haut en utilisant un composant basé sur une fonction, on obtiendra ceci :

Exemple

```
1 import * as React from 'react';
2 import { Text, View } from 'react-native';
3
4 function App() {
5   const [name, setName] = React.useState('Jean');
6
7   function sayHowAreYou(name) {
8     return <Text>Comment ça va {name}</Text>;
9   }
10
11   return (
12     <View>
13       <Text>Coucou {name}</Text>
14       {sayHowAreYou(name)}
15     </View>
16   );
17 }
```

```
15     </View>
16   );
17 }
```

Ici, le corps de la fonction est l'équivalent de la méthode `render`, sauf qu'on va pouvoir faire d'autres choses grâce aux hooks (comme le hook `useState` qui apparaît dans notre exemple).

Les deux composants s'utilisent en invoquant le JSX `<App />`.

Complément

Notez qu'on peut parfaitement mixer les deux types de composants dans un projet.

Rappel

Pour les composants basés sur une fonction, il faut utiliser des hooks, qui sont une nouvelle façon de gérer des états et effets de bords avec React, il en existe de nombreux comme `useState`.

Syntaxe À retenir

- React Native reprend les principes de base de React Web et y apporte un certain nombre de composants qui permettent une abstraction envers leurs équivalents natifs.
- Un composant est une brique logicielle qui permet de réutiliser du code pour le factoriser. Ils peuvent s'écrire de différentes manières : avec des classes JavaScript ou avec des fonctions.
- Pour les composants basés sur une fonction, il faut utiliser des hooks, qui sont une nouvelle façon de gérer des états et effets de bords avec React.

Complément

Composants et API¹

III. Exercice : Appliquez la notion

Question

[solution n°1 p.19]

En vous basant sur ce que nous avons vu précédemment, écrivez un même composant deux fois en utilisant les deux représentations (classe et fonction).

Ce composant doit afficher un texte en utilisant le composant `Text` avec le message « *J'ai réussi* » et utiliser une fonction qui prend en paramètre un texte, et qui affiche ce deuxième texte à la suite du premier.

Il faudra très certainement utiliser un conteneur `View`, car on ne peut renvoyer qu'un seul composant via la méthode `render` ou un fonction `component`.

IV. Le JSX et la notion de parent/enfant

Objectifs

- Comprendre la syntaxe JSX (JavaScript XML) et son utilité
- Comprendre la relation entre les composants dans le virtual DOM

¹ <https://reactnative.dev/docs/components-and-apis>

Mise en situation

JSX (JavaScript XML) est une syntaxe proche du XML permettant de représenter – un peu à la manière du HTML – l'arbre d'imbrication des composants d'une application React. Sa conception est née de la complexité de représenter des interfaces et relations entre composants d'une manière lisible pour les développeurs. Le JSX compile vers du code JavaScript et n'est utilisé qu'à des fins d'amélioration de la lisibilité du code et d'une meilleure expérience de développement.

Exemple

Par exemple, le bouton suivant en JSX :

```
1 const element = (  
2   <Button onPress={() => console.log("I've been clicked")}>  
3     Click me  
4   </Button>  
5 );
```

Revient à écrire et compile vers :

```
1 const element = React.createElement(  
2   'Button',  
3   {onPress: () => console.log("I've been clicked")},  
4   Click me  
5 );
```

`React.createElement` effectue quelques vérifications, mais sa principale fonction est de créer un objet qui ressemble à cela :

```
1 const element = {  
2   type: 'Button',  
3   props: {  
4     onPress: () => console.log("I've been clicked")  
5     children: 'Click me'  
6   }  
7 };
```

Tiens donc, props. Nous avons déjà vu (et allons le voir plus tard), mais children... Qu'est-ce que c'est que cette histoire d'enfants ?

L'imbrication des composants

Les composants s'imbriquent entre eux pour former un grand arbre de dépendances, à la manière de poupées russes.

Remarque

Au final, une application React ou React Native n'est qu'une grosse imbrication de composants pour former, en web, le virtual DOM, puis le DOM (*Data Object Model*) ; et, en mobile, son équivalent virtual DOM et son arbre de composants selon la plateforme.

De cela naît alors une relation d'antérieur (le parent) et postérieur (l'enfant), indiquant qui affiche qui lorsque l'application fonctionne.

Fondamental

Les balises JSX (tout comme le XML ou le HTML) peuvent être ouvrantes/fermantes ou auto-fermantes

- `<Text>Coucou</Text>` est une balise qui s'ouvre au niveau de `<Text>` et se ferme avec `</Text>`. Ici, son enfant est une chaîne de caractères « *Coucou* », mais nous aurions pu utiliser un autre composant JSX.
- `<View />` est une balise auto-fermante. Une balise auto-fermante peut être utilisée si la balise ne possède pas d'enfant direct. Sinon, il faut utiliser la syntaxe ouvrante/fermante.

Prenons par exemple le morceau de code suivant, où s'imbriquent un composant `View` (qui est un conteneur) et un composant `Text`, pour afficher un message dans l'application :

```
1 function App() {
2   return (
3     <View>
4       <Text>
5         Je suis l'enfant
6       </Text>
7     </View>
8   );
9 }
```

Ici, le composant `Text` est l'enfant du composant `<View>`. Il est lui-même le parent de son enfant : le texte « *Je suis l'enfant* ». Nous verrons plus tard sur le chapitre des props et du state comment utiliser les enfants d'un composant via les props. En attendant, il faut savoir qu'un composant qui se *re-render* via un changement de props ou de state déclenche par défaut le *re-render* de tous ses enfants.

Syntaxe

À retenir

- Le JSX (JavaScript XML) est une syntaxe destinée à simplifier l'imbrication des composants dans une application React. Il est compilé vers une syntaxe JavaScript au moment où on génère le fichier JS final via le packager.
- Les composants ont une relation de parent/enfant et forment un grand arbre de composants pour définir la structure d'une application.

Complément

- Introduction au JSX¹

V. Exercice : Appliquez la notion

Question

[solution n°2 p.19]

En vous basant sur ce chapitre, créez un composant personnalisé que vous nommerez comme vous le souhaitez et qui devrait imbriquer un composant `View` parent et deux enfants `Text` qui afficheront respectivement « *Coucou* » et « *Ça marche* ».

VI. Le cycle de vie des composants

¹ <https://fr.reactjs.org/docs/introducing-jsx.html>

Objectifs

- Comprendre la notion de cycle de vie des composants
- Interagir avec le cycle de vie des composants

Mise en situation

Lorsque l'on imbrique des composants React, ces derniers passent par un certain nombre d'états appelés le **cycle de vie d'un composant**. Ces états se déclenchent quand le composant est présent dans un retour d'une méthode `render`, ou lorsqu'il reçoit de nouvelles props ou un nouvel état, par exemple. Jouer avec ces états permet de déclencher des effets de bord ou de calculer des propriétés sans lesquelles cela n'aurait pas été possible.

Le cycle de vie peut se résumer à quatre états, auxquels on peut s'amarrer (*to hook*) :

- **Le composant a été monté** (`componentDidMount`) : l'instance courante du composant s'affiche pour la première fois dans l'arbre des composants. Sa méthode `render` a donc été appelée une seule fois. C'est utile par exemple pour aller effectuer une requête à une base de données ou une API web externe.
- **Le composant va être mis à jour** (`componentWillUpdate`), c'est-à-dire que son state ou ses props (que nous aborderons dans un autre chapitre) ont changé et que l'affichage va se mettre à jour. C'est utile par exemple pour sauvegarder son état précédent avant la mise à jour.
- **Le composant a été mis à jour** (`componentDidUpdate`), une fois que la mise à jour graphique a été traitée. On peut par exemple déclencher un effet de bord pour mettre à jour une propriété calculée à partir du nouvel état ou des nouvelles props.
- **Le composant va se démonter** (`componentWillUnmount`), ce qui permet d'effectuer des actions avant que le composant ne soit retiré de l'arbre et ne termine son cycle de vie. Par exemple, si nous avons ouvert des gestionnaires d'événements ou une WebSocket, il est temps de les fermer ici. C'est un hook de nettoyage.

Le cycle de vie avec les classes

Pour utiliser ces points d'ancrage dans une classe, il suffit d'ajouter une fonction dans la classe du composant qui sera automatiquement appelée par React quand nécessaire.

Attention Les hooks des cycles de vie n'ont pas de relation directe avec les hooks de React

Dans ce chapitre, on pourra parfois lire qu'on peut « hooker le cycle de vie » du composant. Bien que cela soit correct, mais également possible en utilisant les hooks de React (syntaxe pour les composants basée sur une fonction), on ne fait pas forcément allusion à la même chose.

Les hooks comme `useState`, `useEffect` ou encore `useCallback` permettent de hooker au moment du rendu.

Voici un exemple de ce qu'il est possible de faire avec des hooks sur une classe :

Exemple

```
1 /* eslint-disable */
2 import * as React from 'react';
3 import { Text, View, StyleSheet } from 'react-native';
4 import Constants from 'expo-constants';
5
6 export default class App extends React.Component {
7   constructor(props) {
8     super(props)
9   }
```

```

10   this.state = {message: "coucou"}
11 }
12
13 componentDidMount(){
14   console.log("Le composant à été monté une première fois")
15   this.setState({message: "Coucou 2"})
16 }
17
18 componentWillUpdate(){
19   console.log("Avant la mise à jour : ", this.state)
20 }
21
22 componentDidUpdate(){
23   console.log("Après la mise à jour : ", this.state)
24 }
25
26 componentWillUnmount(){
27   console.log("Quand le composant n'est plus affiché dans le JSX, cette méthode est
appelée")
28 }
29
30 render(){
31   // On affiche les valeurs dans la console
32   console.log("Valeurs au rendu : ", this.state)
33
34   return <Text>{this.state.message}</Text>
35 }
36 }

```

Et voici les résultats dans la console :

```

ERRORS      LOGS

ONEPLUS A6003: Valeurs au rendu : ► { message: "coucou" }
ONEPLUS A6003: Avant la mise à jour : ► { message: "coucou" }
ONEPLUS A6003: Le composant à été monté une première fois
ONEPLUS A6003: Après la mise à jour : ► { message: "Coucou 2" }
ONEPLUS A6003: Valeurs au rendu : ► { message: "Coucou 2" }

```

Comme on peut le voir, on peut accéder aux valeurs à différents moments. Par contre, de manière asynchrone, car React n'attend pas que l'on retourne une valeur dans ces fonctions et continue son cycle de vie de son côté.

Le cycle de vie avec les fonctions

Le fonctionnement avec des fonctions est semblable, mais tout se résume à l'utilisation d'un hook appelé `useEffect`. En effet, on ne définit aucune valeur sur le prototype de la fonction. L'exemple ci-dessous est l'équivalent de ce que nous avons fait plus haut en utilisant les composants basés sur les fonctions et les hooks.

Exemple

```
1 /* eslint-disable */
2 import * as React from 'react';
3 import { Text, View, StyleSheet } from 'react-native';
4 import Constants from 'expo-constants';
5
6 export default function App() {
7   const [message, setMessage] = React.useState('coucou');
8   const isFirstRender = React.useRef(true);
9
10  // Cet effet est l'équivalent de notre componentDidMount et componentWillUnmount
11  React.useEffect(() => {
12    console.log('Le composant à été monté une première fois');
13    console.log('Avant la mise à jour : ', message);
14    setMessage('Coucou 2');
15
16    return () => {
17      console.log(
18        "Quand le composant n'est plus affiché dans le JSX, cette méthode est appelée"
19      );
20    };
21  }, []);
22
23  // Équivalent de componentDidUpdate
24  React.useEffect(() => {
25    if (isFirstRender.current) {
26      isFirstRender.current = false;
27      return;
28    }
29
30    console.log('Après la mise à jour : ', message);
31  });
32
33  console.log('Valeurs au rendu : ', message);
34
35  return <Text>{message}</Text>;
36 }
```

Bien que différent, ce code est ce qui se rapproche le plus de l'exemple vu plus haut.

Remarque

`console.log` étant asynchrone, il ne faut pas trop se fier à l'ordre d'affichage dans la console : les fonctions sont appelées dans le bon ordre.

En bref, le hook `useEffect` prend en arguments deux paramètres :

- Une fonction à exécuter lors du déclenchement du hook, soit quand le tableau des dépendances change, qui doit retourner une fonction de nettoyage à exécuter juste avant que cet effet ne se déclenche à nouveau.
- Un tableau de dépendances (deps), qui sont des variables observées qui vont être évaluées à chaque rendu et, si l'une d'entre elles a été modifiée, qui vont déclencher la fonction de l'effet, en utilisant la fonction de nettoyage de l'effet précédent si jamais elle existe. Si jamais ce paramètre n'est pas donné, le hook s'exécute à chaque `render`.

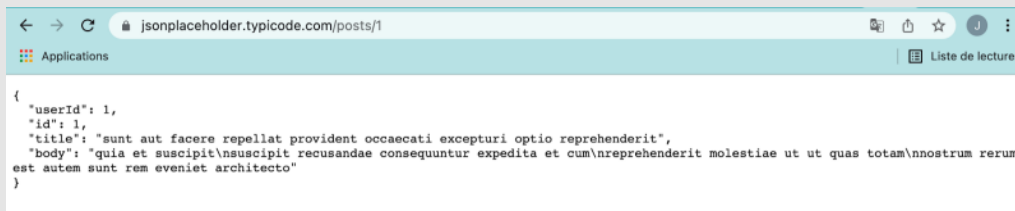
Pour plus de détails, consultez la documentation officielle qui est très exhaustive à ce sujet : <https://fr.reactjs.org/docs/hooks-effect.html>.

Méthode Un cas d'utilisation de useEffect avec fetch

Comme vu au-dessus, `useEffect` peut être utilisé pour lancer une fonction au montage du composant.

Cette fonction sera donc lancée après le premier rendu, mais pourra charger une information dynamique à l'initialisation du composant.

Dans cette méthode, nous allons donc charger un post (publication) à partir d'une API en appelant une source extérieure. Pour cet exemple, nous appellerons une API nommée JSON Placeholder, cette API nous générera une réponse avec de fausses données (comme l'image ci-dessous).



Pour cela, nous utiliserons `useEffect(() => <Function Call posts>)` qui exécutera la fonction pendant le montage du composant, nous utiliserons `fetch` et cette dernière mettra l'information dans une state.

```
1 import * as React from 'react'
2 import {
3   View,
4   Text
5 } from 'react-native'
6
7 const getPost = () => {
8   const [isLoading, setLoading] = React.useState(true)
9   // Nous utiliserons cette state pour afficher un message de chargement
10
11   const [post, setPost] = React.useState(null)
12   // Nous utiliserons cette state pour réceptionner le corps de la publication
13
14   React.useEffect(() => {
15     fetch('https://jsonplaceholder.typicode.com/posts/1')
16       .then((response) => response.json())
17       .then((json) => setPost(json.body))
18       .catch((error) => console.error(error))
19       .finally(() => setLoading(false));
20   }, [])
21
22   return (
23     <View>
24       {isLoading
25       ? <Text>En chargement</Text>
26       : <Text>{post}</Text>
27     }
28     </View>
29   )
30 }
31
32 export default getPost
```

1. Nous lançons une fonction `useEffect` de React pour qu'elle se lance à l'initiation de l'application.
2. Nous réalisons une requête `fetch` (de base cela fait une requête de type GET) à l'URL passé en paramètre `fetch(<url>)` nous retournera une promesse. Une promesse est un objet qui représente l'accomplissement d'une tâche, celle-ci nous renvoie une réussite avec sa méthode `.then()` et `.finally()`, ou un échec avec `.catch()`.

Si nous imaginons ce concept, nous pourrions prendre le téléchargement d'un livre, la promesse constitue son résultat, soit celui-ci serait à 100%, fini et nous renverrai la méthode `.then()` ou `finally()` soit il aurait échoué par exemple à 20% et nous renverrai sa méthode `.catch()`. L'avantage des promesses est que notre code attendra d'avoir une réponse ou une erreur pour lancer le code dans le `.then()`, `.finally()` ou `.catch()`.

3. Avec le premier `.then()` nous récupérons le flux réponse, seulement, React Native ne peut le lire tel qu'il est.

Nous le mettons donc au format json grâce à la fonction du même nom. La fonction `json` renvoi aussi une promesse.

4. Maintenant que nous avons notre réponse au bon format, nous devons la mettre dans l'état (le state). Dans notre cas la réponse contient plusieurs champs mais nous ne souhaitons que le champs `body`.

```

1  {
2    "userId": 1,
3    "id": 1,
4    "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
5    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"
6  }

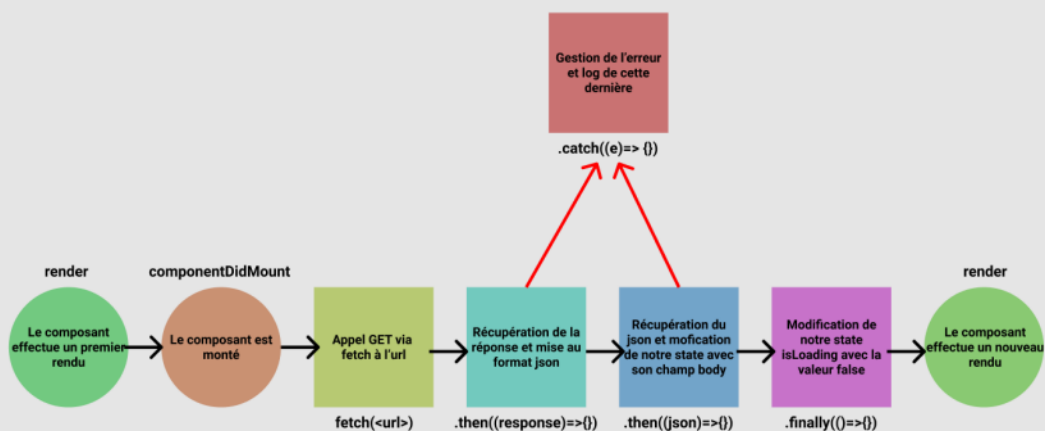
```

Nous passons donc seulement le `body` dans l'état : `setPost (json.body)`

5. Avec le `.catch()`, nous gérons le cas où il y aurait une erreur et le mettons en console.

6. Le `finally`, permet de lancer une fonction, une fois que les 2 `.then()` ont été réalisés sans erreur, dans notre cas, nous passons le statut en chargement (`isLoading`) sur `faux` pour que notre interface nous affiche le résultat.

Si nous schématisons cette utilisation de `useEffect` avec `fetch`, voici ce qui se déroulerait :



Bien que non visible (vous ne verrez probablement pas le message En chargement) car trop rapide dans un cas simple comme celui-ci, il est important de savoir comment fonctionne le cycle de vie de React native et comment l'hook `useEffect` permet d'interagir avec ce dernier.

Son utilisation avec `fetch` en est un bon exemple, une application ayant très souvent besoin d'une API pour fonctionner.

Sur les 3 premières étapes de notre schéma, notre composant affichera En chargement, puis affichera les informations récupérées via l'URL (à l'exception du cas où une erreur serait advenue).

Syntaxe **À retenir**

- Les composants possèdent un cycle de vie qui leur permet de venir déclencher des effets de bord lorsqu'ils sont créés, détruits ou qu'ils reçoivent de nouveaux arguments.
- Bien qu'on puisse faire la même chose avec des composants basés sur des classes ou d'autres basés sur des fonctions, la syntaxe n'est pas tout à fait équivalente.
- La plupart du temps, on utilisera les hooks pour aller chercher des données distantes ou pour initialiser des gestionnaires d'événements.
- L'hook `useEffect` permet d'interagir avec React native en fonction de son avancement dans le cycle de vie du composant, il permet par exemple de récupérer des données distantes avec `fetch`, au lancement du composant.

Complément

- <https://fr.reactjs.org/docs/hooks-effect.html>
- <https://fr.reactjs.org/docs/hooks-reference.html#useeffect>
- <https://fr.reactjs.org/docs/state-and-lifecycle.html>

VII. Exercice : Appliquez la notion

Question

[solution n°3 p.20]

Avec ce que vous venez de voir et en vous aidant de la documentation, écrivez le code du composant `Coucou` pour qu'il affiche le message « *Je vais disparaître* » dans la console, lorsque celui-ci est retiré de l'arbre des composants (au clic sur le bouton de son parent).

Pour cela, vous aurez besoin de l'instruction `console.log`.

Ajoutez également le message « *Je suis apparu* » quand celui-ci réapparaît.

Vous utiliserez la syntaxe des composants fonctionnels pour cet exercice.

Le code du parent avec le bouton est fourni en indice : il ne faut cependant pas le modifier, mais simplement implémenter le composant enfant `Coucou`.

Indice :

```
1 /* eslint-disable */
2 import * as React from 'react';
3 import { Text, Button, View } from 'react-native';
4
5 export default function App() {
6   const [showCoucou, setShowCoucou] = React.useState(true);
7
8   return (
9     <View>
10       <Button title={showCoucou ? "Masquer coucou" : "Afficher coucou"} onPress={() =>
11         setShowCoucou((prev) => !prev)} />
12       {showCoucou ? <Coucou /> : <Text>Pas coucou</Text>}
13     </View>
14   );
15 }
16
17 function Coucou() {
18   // À nous de jouer, ajoutons le code nécessaire pour déclencher l'effet de bord
19   return <Text>Coucou</Text>;
20 }
```

VIII. Essentiel

IX. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°4 p.20]

Exercice

Quel est l'équivalent Android du composant `Image` ?

- ☐ `AndroidImage`
- ☐ `ImageView`
- ☐ `ImageComponent`
- ☐ `UIImage`

Exercice

Quel est l'équivalent iOS du composant `Text` ?

- ☐ `Text`
- ☐ `AppleText`
- ☐ `UITextView`
- ☐ `UIText`

Exercice

On peut utiliser des hooks directement dans la méthode `render` d'un composant basé sur une classe.

- ☐ Vrai
- ☐ Faux
- ☐ Vrai, mais sous conditions

Exercice

On peut mixer les composants basés sur une fonction et ceux basés sur des classes dans une application React.

- ☐ Vrai
- ☐ Faux
- ☐ Faux, sauf dans un cas

Exercice

Que signifie l'acronyme JSX ?

- ☐ JavaScript XML
- ☐ JavaScript X (JavaScript 10)
- ☐ JavaScript Super XML

Exercice

Si un composant n'a pas d'enfant, quel type de syntaxe est-il préférable d'utiliser en JSX ?

- ☐ Les balises ouvrantes/fermantes
- ☐ La balise ouvrante uniquement
- ☐ La balise auto-fermante

Exercice

Quel point d'ancrage de classes est utilisé pour déclencher un effet de bord sur le démontage d'un composant ?

- ☐ `componentWillUnmount`
- ☐ `componentWillDismount`
- ☐ `componentDismounted`
- ☐ `iWillDismount`

Exercice

Quel hook (au sens *function component*) doit-on utiliser pour s'approcher du fonctionnement des hooks de style de vie d'un composant basé sur une classe ?

- ☐ `useLifecycle`
- ☐ `useLifecycleEffect`
- ☐ `useEffect`

Exercice

Quelle phrase représente le mieux le ou les arguments de la fonction `useEffect` du hook pour les composants basés sur les fonctions ?

- ☐ Il n'y a qu'un seul et unique argument obligatoire
- ☐ Le premier argument est obligatoire et le second est optionnel
- ☐ Les deux arguments sont optionnels
- ☐ Les deux arguments sont obligatoires

Exercice

Parmi ces composants, lequel ne fait pas partie des composants par défaut fournis avec React Native ?

- ☐ `View`
- ☐ `Text`
- ☐ `Video`
- ☐ `Image`

B. Exercice : Défi

Grâce à tout ce que vous venez de voir, réécrivez les composants basés sur les classes ci-dessous en utilisant la syntaxe des composants basés sur des fonctions (fonctionnels).

Question

[solution n°5 p.23]

Cela n'a pas encore été abordé, mais, pour représenter l'état, on peut utiliser le hook `useState` qui est documenté ici¹ ou dans le chapitre « Les props et le state ».

Le comportement à observer n'affiche rien grâce à la valeur `null`, puis, 3 secondes après, le texte apparaît avec une modale. Enfin, 3 secondes après, une autre modale apparaît et le texte disparaît.

```
1 import * as React from 'react';
2 import { Alert, Text, View, StyleSheet } from 'react-native';
3
4 export default class App extends React.Component {
5   constructor(props) {
6     super(props);
7     this.state = { showText: false };
8   }
9
10  componentDidMount() {
11    // Afficher après 3 secondes
12    setTimeout(() => {
13      this.setState({ showText: true });
14    }, 3000);
15
16    // Masquer après 6 secondes
17    setTimeout(() => {
18      this.setState({ showText: false });
19    }, 6000);
20  }
21
22  render() {
23    return <View>{this.state.showText ? <MySpecialText /> : null}</View>;
24  }
25 }
26
27 class MySpecialText extends React.Component {
28   componentDidMount() {
29     Alert.alert(
30       'Le composant à été monté',
31       'Ce message n'apparaît qu'une seule fois'
32     );
33   }
34
35   componentWillUnmount() {
36     Alert.alert(
37       'Le composant à été démonté',
38       'Ce message n'apparaît qu'une seule fois'
39     );
40   }
41
42   render() {
43     return <Text>Mon texte spécial</Text>;
44   }
45 }
```

Solutions des exercices

1 <https://fr.reactjs.org/docs/hooks-reference.html#usestate>

p.6 Solution n°1

```
1 import * as React from 'react';
2 import { Text, View } from 'react-native';
3
4 // Première façon de procéder en mode "Function Component"
5 function App(props) {
6   // Déclare une fonction scopée
7   function displayText(value) {
8     return <Text>{value}</Text>;
9   }
10
11   return (
12     <View>
13       <Text>J'ai réussi</Text>
14       {displayText('à afficher mon texte')}
15     </View>
16   );
17 }
18
19 // Deuxième façon de procéder en mode "Classe"
20 class App2 extends React.Component {
21   // Attache une fonction au prototype de la classe
22   displayText(value) {
23     return <Text>{value}</Text>;
24   }
25
26   render() {
27     return (
28       <View>
29         <Text>J'ai réussi</Text>
30         {this.displayText('à afficher mon texte')}
31       </View>
32     );
33   }
34 }
35
36 export default App;
```

p.8 Solution n°2

```
1 import * as React from 'react';
2 import { Text, View } from 'react-native';
3
4 function MyCustomComponent() {
5   return (
6     <View>
7       <Text>Coucou</Text>
8       <Text>Ça marche</Text>
9     </View>
10   );
11 }
```


p. 14 Solution n°3

```
1 /* eslint-disable */
2 import * as React from 'react';
3 import { Text, Button, View } from 'react-native';
4
5 export default function App() {
6   const [showCoucou, setShowCoucou] = React.useState(true);
7
8   return (
9     <View>
10       <Button
11         title={showCoucou ? 'Masquer coucou' : 'Afficher coucou'}
12         onPress={() => setShowCoucou((prev) => !prev)}
13       />
14       {showCoucou ? <Coucou /> : <Text>Pas coucou</Text>}
15     </View>
16   );
17 }
18
19 function Coucou() {
20   React.useEffect(() => {
21     console.log("Je suis apparu")
22
23     return () => {
24       console.log('Je vais disparaître');
25     };
26   }, []);
27
28   return <Text>Coucou</Text>;
29 }
```

Exercice p. 15 Solution n°4

Exercice


Quel est l'équivalent Android du composant Image ?

- ☐ AndroidImage
- ☒ ImageView
- ☐ ImageComponent
- ☐ UIImage
-  La bonne réponse est bien évidemment ImageView.

Exercice

Quel est l'équivalent iOS du composant Text ?


- ☐ Text
- ☐ AppleText
- ☒ UITextView
- ☐ UIText

 La bonne réponse est UITextView.

Exercice

On peut utiliser des hooks directement dans la méthode `render` d'un composant basé sur une classe.


- ☐ Vrai
- ☒ Faux
- ☐ Vrai, mais sous conditions

 Non, ce n'est pas possible, les hooks sont réservés aux composants basés sur des fonctions.

Exercice

On peut mixer les composants basés sur une fonction et ceux basés sur des classes dans une application React.


- ☒ Vrai
- ☐ Faux
- ☐ Faux, sauf dans un cas

 Il est tout à fait possible de mixer les deux types de composants : React ne fera pas la différence.

Exercice

Que signifie l'acronyme JSX ?


- ☒ JavaScript XML
- ☐ JavaScript X (JavaScript 10)
- ☐ JavaScript Super XML

 JSX est une syntaxe basée sur XML, la bonne réponse est donc JavaScript XML. Il permet de simplifier la représentation de l'arbre des composants.

Exercice

Si un composant n'a pas d'enfant, quel type de syntaxe est-il préférable d'utiliser en JSX ?


- ☐ Les balises ouvrantes/fermantes
- ☐ La balise ouvrante uniquement
- ☒ La balise auto-fermante

 Si un composant n'a pas d'enfant, afin de l'identifier simplement, il est préférable d'utiliser la balise auto-fermante. Par exemple, pour ce bouton qui ne fait rien : `<Button title="Click me" onPress={() => null} />`.

Exercice

Quel point d'ancrage de classes est utilisé pour déclencher un effet de bord sur le démontage d'un composant ?


- ☒ `componentWillUnmount`
- ☐ `componentWillDismount`
- ☐ `componentDismounted`
- ☐ `iWillDismount`

 La bonne syntaxe est `componentWillUnmount`, les autres syntaxes n'existent pas.

Exercice

Quel hook (au sens *function component*) doit-on utiliser pour s'approcher du fonctionnement des hooks de style de vie d'un composant basé sur une classe ?


- ☐ `useLifecycle`
- ☐ `useLifecycleEffect`
- ☒ `useEffect`

 La seule syntaxe qui existe et qui se rapproche le plus de ce qui se fait pour les classes est le hook `useEffect`.

Exercice

Quelle phrase représente le mieux le ou les arguments de la fonction `useEffect` du hook pour les composants basés sur les fonctions ?


- ☐ Il n'y a qu'un seul et unique argument obligatoire
- ☒ Le premier argument est obligatoire et le second est optionnel
- ☐ Les deux arguments sont optionnels
- ☐ Les deux arguments sont obligatoires

 Le premier argument est obligatoire et c'est une fonction qui définit ce que doit faire l'effet et comment le nettoyer si besoin. Le second est optionnel et prend la forme d'un tableau, qui permet de configurer sous quelles conditions se déclenche l'effet.

Exercice

Parmi ces composants, lequel ne fait pas partie des composants par défaut fournis avec React Native ?

- ☐ `View`
- ☐ `Text`
- ☒ `Video`
- ☐ `Image`

 Il existe bien un composant `Video`, mais il ne fait pas partie de React Native par défaut et requiert l'installation de dépendances annexes. On peut par exemple utiliser <https://docs.expo.io/versions/latest/sdk/video/> avec Expo.

p. 17 Solution n°5

```
1 import * as React from 'react';
2 import { Alert, Text, View, StyleSheet } from 'react-native';
3
4 export default function App() {
5   const [showText, setShowText] = React.useState(false);
6
7   React.useEffect(() => {
8     // Afficher après 3 secondes
9     setTimeout(() => {
10       setShowText(true);
11     }, 3000);
12
13     // Masquer après 6 secondes
14     setTimeout(() => {
15       setShowText(false);
16     }, 6000);
17   }, []);
18
19   return <View>{showText ? <MySpecialText /> : null}</View>;
20 }
21
22 function MySpecialText() {
23   React.useEffect(() => {
24     Alert.alert(
25       'Le composant à été monté',
26       "Ce message n'apparaît qu'une seule fois"
27     );
28
29     return () => {
30       Alert.alert(
31         'Le composant à été démonté',
32         "Ce message n'apparaît qu'une seule fois"
33       );
34     };
35   }, []);
36
37   return <Text>Mon texte spécial</Text>;
38 }
```