

# Les props et le state

# Table des matières

<b>I. Contexte</b>	<b>3</b>
<b>II. Changer l'état d'un composant</b>	<b>3</b>
<b>III. Exercice : Appliquez la notion</b>	<b>8</b>
<b>IV. Faire communiquer les composants entre eux</b>	<b>8</b>
<b>V. Exercice : Appliquez la notion</b>	<b>11</b>
<b>VI. La communication bidirectionnelle entre les composants</b>	<b>11</b>
<b>VII. Exercice : Appliquez la notion</b>	<b>13</b>
<b>VIII. Valider l'interface de données de ses props et valeurs par défaut</b>	<b>14</b>
<b>IX. Exercice : Appliquez la notion</b>	<b>17</b>
<b>X. Essentiel</b>	<b>18</b>
<b>XI. Auto-évaluation</b>	<b>18</b>
A. Exercice final .....	18
B. Exercice : Défi.....	20
<b>Solutions des exercices</b>	<b>20</b>

## I. Contexte

**Durée :** 1 h

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

**Pré-requis :** Aucun

### Contexte

Jusqu'ici, nos composants étaient statiques, c'est-à-dire que leur état correspondait uniquement au code que nous avons écrit et l'utilisateur ne pouvait pas altérer les valeurs durant le fonctionnement de l'application.

Cependant, il est possible de faire varier l'affichage d'un composant dynamiquement, en changeant son état (*state* en anglais). Par exemple, pour changer une couleur lors d'un clic sur un bouton.

On remarque également que l'on réécrit beaucoup de code similaire : il peut donc être parfois intéressant d'extraire du code redondant dans un composant spécifique afin de réduire cet effet. De cette manière, on peut simplement utiliser ce nouveau composant et lui communiquer les valeurs à utiliser via des propriétés (props).

Une application est comme un gros Lego : un assemblage de plusieurs états, qui vont définir dynamiquement plusieurs propriétés dans des composants personnalisés.

D'une manière générale, l'état sert à faire varier un composant en réponse à des événements, et les propriétés (props) permettent de définir une interface pour fournir à un composant les données nécessaires à son rendu graphique.

## II. Changer l'état d'un composant

### Objectifs

- Découvrir la gestion et l'utilisation de l'état dans une application React
- Faire varier l'affichage d'un écran sans recharger l'application
- Utiliser des valeurs dynamiques calculées avec un petit algorithme qui réagit aux événements

### Mise en situation

Dans une vraie application, l'interface graphique doit réagir à des modifications, telles que les actions de l'utilisateur, ou bien à la réception de nouvelles données à afficher (par exemple, de nouveaux messages envoyés dans un *chat*).

Dans une application React, ces événements et données sont conservés dans des variables spéciales, appelées état (*state* en anglais), que l'on peut faire varier durant le fonctionnement de l'application. De cette manière, on peut modifier l'application sans avoir à la recharger, et rendre l'expérience utilisateur plus agréable et dynamique. Cette partie nous apprend à utiliser un état dans une application React. Pour cela, nous allons utiliser un hook conçu spécialement pour : le hook `useState`.

Le hook `useState` est conçu pour gérer un état applicatif unique, comme son nom l'indique. Nous pouvons utiliser plusieurs gestionnaires d'état dans un même composant. Par exemple, si l'on souhaite gérer un état d'un chiffre et d'un texte, nous ferons deux fois appel à `useState` pour chacune des valeurs.

## Introduction aux composants

Un composant est un élément d'interface indépendant et réutilisable (par exemple : une page d'accueil, un formulaire de connexion).

Il en existe 2 types :

- Les composants de classe qui seront déclarés dans une class JavaScript :

```
1 import * as React from 'react';
2 import { Text, View } from 'react-native';
3
4 class App extends React.Component {
5   render() {
6     return (
7       <View>
8         <Text>Hello World</Text>
9       </View>
10    );
11  }
12 }
```

- Les composants fonctionnels qui seront déclarés dans une function JavaScript :

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

Les composants sont constitués d'un return qui renverra du texte JSX (un langage proche du HTML/XML) ou du HTML, cependant comme le HTML sur un site standard, le code donne lieu à du contenu static, pour pallier cela nous utilisons les states et les props que nous allons voir dans ce chapitre.

Les composants seront approfondi, au chapitre suivant.

### Attention

La seule contrainte des hooks est que votre code doit rester pur et non conditionnel. Un hook doit être invoqué à chaque rendu et ne peut pas se trouver après une structure conditionnelle comme un if. Sans quoi, le moteur de React perd les pédales et votre application ne fonctionnera pas correctement.

### Remarque

On peut observer une syntaxe un peu étrange lors de l'utilisation de `useState` :

```
1 const [state, setState] = useState(0)
```

Cette instruction utilise l'opérateur de décomposition.

`useState(0)` renvoie en réalité un tableau avec deux éléments :

- Le premier à l'index 0 est le state
- Le deuxième à l'index 1 est une fonction pour mettre à jour le state

La syntaxe indiquée plus haut permet de dire que la valeur du tableau à l'index 0 est stockée dans une variable `state` et la valeur à l'index 1 dans une variable `setState`.

On pourrait également écrire le hook de la manière suivante, ce qui serait la même chose, mais plus verbeux :

```
1 const hookData = useState(0)
2 // hookData[0] est l'équivalent de state
3 // hookData[1] est l'équivalent de setState
```

Pour plus d'information sur cette syntaxe, vous pouvez consulter cet article.<sup>1</sup>

Pour illustrer la gestion de l'état, dans l'exemple qui suit, nous allons faire varier la couleur d'une boîte du rouge au vert en fonction du bouton cliqué.

Le premier élément est la valeur courante de notre état au moment du rendu `boxColor`, la deuxième `setBoxColor` est le mutateur qui nous permet d'altérer cette valeur et de déclencher un re-rendu graphique.

Ici, nous les avons nommées comme ceci, mais nous pouvons les nommer comme bon nous semble.

#### Exemple

```
1 import React from "react";
2 import { View, Button } from "react-native";
3
4 const MyApplicationRoot = () => {
5   // On déclare un état avec pour valeur initiale red (rouge)
6   const [boxColor, setBoxColor] = React.useState("red");
7
8   return (
9     <View>
10      /* On définit la couleur d'arrière plan à la valeur de notre état application "boxColor"
11     */
12     <View style={{ backgroundColor: boxColor, width: 50, height: 50 }} />
13     <Button
14       // Lors du clic sur le bouton, on met à jour l'état avec la valeur verte.
15       onPress={() => setBoxColor("green")}
16       title="Afficher en vert"
17     />
18     <Button
19       // Lors du clic sur le bouton, on met à jour l'état avec la valeur rouge.
20       onPress={() => setBoxColor("red")}
21       title="Afficher en rouge"
22     />
23   </View>
24 );
25
26 export default MyApplicationRoot
```

Ceci couvre le cas d'usage simple du hook `useState` : ce hook permet de conserver une valeur entre les différents rendus de notre composant. Maintenant, si notre nouvel état dépend de la valeur précédente de notre état, on pourrait être tenté de faire la chose suivante, dans le cas d'un compteur numérique, par exemple.

<sup>1</sup> [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/Affecter\\_par\\_d%C3%A9finition](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/Affecter_par_d%C3%A9finition)

### Exemple

```

1 import React from "react";
2 import { View, Button } from "react-native";
3
4 const MyApplicationRoot = () => {
5   // On déclare un état avec pour valeur initiale 0
6   const [counter, setCounter] = React.useState(0);
7
8   return (
9     <View>
10      <Button
11        // Au clic on augmente la valeur de 1
12        onPress={() => setCounter(counter + 1)}
13        title={"Valeur du compteur " + counter}
14      />
15    </View>
16  );
17 };
18
19 export default MyApplicationRoot

```

### Attention L'exemple en amont est faux

En effet, on ne peut pas utiliser la valeur précédente de cette manière et cela pourrait créer des bugs. La bonne manière de procéder est de passer une fonction à notre méthode `setCounter`.

### Conseil Le bon exemple

```

1 import React from "react";
2 import { View, Button } from "react-native";
3
4 const MyApplicationRoot = () => {
5   // On déclare un état avec pour valeur initiale 0
6   const [counter, setCounter] = React.useState(0);
7
8   return (
9     <View>
10      <Button
11        // Au clic on augmente la valeur de 1
12        onPress={() => setCounter(previousCounterValue => previousCounterValue + 1)}
13        title={"Valeur du compteur " + counter}
14      />
15    </View>
16  );
17 };
18
19 export default MyApplicationRoot

```

De cette manière, la mise à jour est sécurisée dans le temps. Pour plus d'informations sur la gestion de l'état et sur le hook `useState`, on peut se référer à la documentation de React, qui est très exhaustive sur ce sujet.

### Complément Les hooks pour les composants basés sur les fonctions

Le vrai avantage des composants basés sur les fonctions est l'utilisation des hooks, qui est une fonctionnalité assez récente dans React. Elle permet de définir des briques de logique réutilisables, qui ne sont pas des composants.

Par exemple, le hook `useState` permet de conserver la valeur d'un état à travers les rendus successifs du composant.

React est fourni avec un certain nombre de hooks par défaut. La liste est d'ailleurs accessible ici<sup>1</sup>. Et on peut, tout comme les composants, composer les hooks qui existent pour fabriquer nos propres hooks.

```
1  import * as React from 'react';
2  import { Text, View } from 'react-native';
3
4  // MAUVAIS EXEMPLE, structure conditionnelle
5  function App(props) {
6    if (props.randomInteger < 5) {
7      return null;
8    }
9
10   const [state, setState] = React.useState()
11
12   return <Text>Hello</Text>;
13 }
14
15 // BON EXEMPLE, car le hook est avant la structure conditionnelle
16 function App2(props) {
17   const [state, setState] = React.useState()
18
19   if (props.randomInteger < 5) {
20     return null;
21   }
22
23   return <Text>Hello</Text>;
24 }
```

### Fondamental Un composant déclenche son re-rendu sur divers événements

- Si son state (état) change
- Si une de ses props change
- Si son parent a une de ces deux valeurs qui change

On peut donc avoir un re-rendu en chaîne sous certaines conditions.

### Syntaxe À retenir

- On peut faire varier le rendu graphique d'une application dynamiquement selon divers événements en utilisant le state.
- Le principal outil pour cela est le hook `useState` qui donne accès à deux variables :
  - L'état courant de l'état,
  - Une fonction permettant de mettre à jour l'état.
- Lorsqu'on se réfère à la valeur précédente de l'état dans la mise à jour, il faut utiliser une fonction de rappel plutôt que la variable de l'état, sinon on risque de créer des bugs très difficiles à déboguer.
- 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.

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

### Complément

- <https://fr.reactjs.org/docs/state-and-lifecycle.html>
- <https://fr.reactjs.org/docs/hooks-state.html>
- [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Opérateurs/Affecter\\_par\\_décomposition](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Opérateurs/Affecter_par_décomposition)

## III. Exercice : Appliquez la notion

### Question

[solution n°1 p.21]

Améliorez le compteur ci-dessous avec un texte et séparez le bouton +1 du bouton -1 pour pouvoir également soustraire 1 au compteur.

- Au clic sur -1, si le résultat du clic devrait normalement afficher -1, gardez 0 et affichez un message d'erreur pour expliquer que le nombre ne peut pas descendre en dessous de 0.
- Le message d'erreur doit disparaître à la prochaine action autorisée de l'utilisateur (c'est-à-dire s'il clique sur +1 après avoir essayé de cliquer sur -1 quand le compteur valait 0 et que le message s'était affiché).

```
1 import React from "react";
2 import { View, Button } from "react-native";
3
4 const MyApplicationRoot = () => {
5   // On déclare un état avec pour valeur initiale 0
6   const [counter, setCounter] = React.useState(0);
7
8   return (
9     <View>
10       <Button
11         // Au clic on augmente la valeur de 1
12         onPress={() => setCounter(previousCounterValue => previousCounterValue + 1)}
13         title={"Valeur du compteur " + counter}
14       />
15     </View>
16   );
17 };
18
19 export default MyApplicationRoot
```

## IV. Faire communiquer les composants entre eux

### Objectifs

- Découvrir l'utilisation des props dans React
- Permettre à des composants d'échanger des données entre eux
- Comprendre comment factoriser du code pour en écrire moins et le réutiliser

### Mise en situation

Maintenant que nous avons un premier rendu et que nous arrivons à faire varier l'état de nos composants, on remarque que, si l'on souhaite réutiliser notre compteur, on est obligé de réécrire le code du compteur, ce qui conduit à une répétition du code. Pas très efficace, n'est-ce pas ?

Ce compteur pourrait être rendu autonome et réutilisé dans une autre partie de l'application sans aucun problème. On pourrait donc choisir de factoriser ce code afin de réafficher très simplement ce composant ailleurs, en lui donnant simplement les informations spécifiques nécessaires à son rendu, telles que la valeur initiale du compteur.



Nous allons aborder la notion de **props** et de réutilisation du code à travers un exemple d'un texte avec une variation de couleur, et plus tard, dans un exercice, nous essaierons de créer notre compteur réutilisable.

Avant ce chapitre, si nous avions besoin d'afficher deux fois un texte avec du style, nous aurions fait quelque chose de la sorte, en dupliquant du code :

**Exemple**

```
1 import React from "react";
2 import { View, Text } from "react-native";
3
4 const MyApplicationRoot = () => {
5   return <View>
6     <View style={{ backgroundColor: "red", padding: 5 }}>
7       <Text>Mon super texte</Text>
8     </View>
9     <View style={{ backgroundColor: "red", padding: 5 }}>
10      <Text>Mon super texte 2</Text>
11    </View>
12  </View>;
13 };
14
15 export default MyApplicationRoot
```

Désormais, nous pouvons utiliser le mécanisme des props de React pour définir une interface à ce texte formaté, afin de le faire varier selon notre contexte. De cette manière, on pourrait créer un composant spécifique pour notre texte et lui passer les éléments nécessaires à son affichage à chaque instance.

**Exemple**

```
1 import React from "react";
2 import { View, Text } from "react-native";
3
4 // Par défaut si la props n'est pas spécifié la couleur sera "red"
5 const MyCustomText = (props) => {
6   <View style={{ backgroundColor: props.backgroundColor || "red", padding: 5 }}>
7     <Text>{props.title}</Text>
8   </View>;
9 };
10
11 const MyApplicationRoot = () => {
12   return (
13     <View>
14       <MyCustomText backgroundColor="red" title="Mon super texte" />
15       <MyCustomText backgroundColor="red" title="Mon super texte 2" />
16     </View>
17   );
18 };
19
20 export default MyApplicationRoot
```

## Qu'est-ce qu'une props ?

Les props sont des propriétés passées sous la forme d'un objet à un composant en utilisant du JSX. Nous pouvons observer sur le schéma ci-dessous la corrélation entre les attributs passés au composant et leur utilisation. On peut passer n'importe quel type primitif (number, string, boolean...) statique ou dynamique, même des objets, fonctions ou autres composants React en paramètres via les props.

```
// Par défaut si la props n'est pas spécifié la couleur sera "red"
const MyCustomText = (props) => {
  <View style={{ backgroundColor: props.backgroundColor || "red", padding: 5 }}>
    <Text>{props.title}</Text>
  </View>;
};

const MyApplicationRoot = () => {
  return (
    <View>
      <MyCustomText backgroundColor="red" title="Mon super texte" />
      <MyCustomText backgroundColor="red" title="Mon super texte 2" />
    </View>;
  );
};
```

Diagram illustrating prop usage:

- Arrows point from `props.backgroundColor` and `props.title` in the `MyCustomText` function to their respective values in the JSX elements.
- Labels "utilisation d'une props" (usage of a prop) are placed near the arrows pointing to the props in the function.
- Labels "props" are placed near the arrows pointing to the props in the JSX elements.

### Attention Les props sont en lecture seule

Il ne faut surtout pas altérer les props en essayant de modifier la variable directement.

Comme le composant parent passe les props à un composant enfant, on parle de communication descendante (ou, en anglais, *props injection* / *props drilling*). Ce genre de communication est très pratique, car on peut factoriser du code qui se répète pour une meilleure réutilisation.

Cependant, le composant parent passe également une fonction permettant d'effectuer un effet de bord. Plus tard, nous verrons que cet effet de bord peut concerner l'état du parent : on parle alors de communication montante (ou *state lifting*).

## La prop children

Nous pouvons également utiliser une props un peu spéciale, fournie automatiquement : `children`, qui sert à afficher les enfants de notre composant.

### Exemple

```
1 const MyComponent = (props) => <Text>{props.children}</Text>
2
3 const App = () => <MyComponent>Coucou</MyComponent>
```

Ici, on voit qu'un enfant `MyComponent` possède la chaîne de caractères « *Coucou* ». Cette dernière est utilisable dans le composant via la prop `children`, comme montré dans cet exemple. On aurait également très bien pu afficher un autre composant de cette manière.

### Syntaxe À retenir

- Les props sont un moyen de définir une interface d'échange de données entre des composants. En créant de nouveaux composants, on favorise la factorisation du code et sa réutilisation.
- Elles peuvent être optionnelles ou obligatoires et reçoivent n'importe quel type de données qui pourrait être exploité par un composant.

- Elles sont en lecture seule, c'est-à-dire qu'il ne faut pas altérer les valeurs des props depuis le composant qui les utilise.

Bref, il faut en abuser ! À consommer sans modération.

#### Complément

- <https://fr.reactjs.org/docs/components-and-props.html>
- <https://fr.reactjs.org/docs/render-props.html>
- <https://fr.reactjs.org/docs/jsx-in-depth.html>

## V. Exercice : Appliquez la notion

### Question

[solution n°2 p.21]

Créez une petite application qui aura trois compteurs affichés à l'écran, indépendants les uns des autres (c'est-à-dire qu'incrémenter un des compteurs s'applique uniquement à lui-même).

Chaque compteur est composé d'un bouton +1 et d'un bouton -1 et affiche la valeur actuelle du compteur. Le design n'a pas d'importance dans ce module.

- On peut donner la valeur par défaut du compteur via les props : si elle n'est pas donnée, le compteur commence à 10.
- Le pas du compteur est de 1 à chaque clic.
- Essayez de donner une valeur par défaut à 7 pour l'un des compteurs.

Les compteurs n'ont pas besoin d'avoir la sécurité du module précédent, qui les empêche de descendre en dessous de 0.

## VI. La communication bidirectionnelle entre les composants

### Objectifs

- Comprendre la communication inversée (montante) qui permet à un enfant d'altérer l'état d'un parent via les props
- Comprendre la notion de « *side effects* » (effets de bord) en permettant à un parent d'injecter une fonction de rappel à son enfant

### Mise en situation

Jusqu'à maintenant, un composant parent envoyait des données à un composant enfant par l'intermédiaire de props, mais le composant n'avait aucun moyen de notifier le composant parent d'un événement.

Par exemple, le composant enfant pourrait être un bouton avec un design spécifique, mais la logique de l'effet devant déclencher le bouton n'est pas assez générique pour se trouver dans le composant bouton. Il faut donc que le parent dise à l'enfant ce qu'il faut faire quand ce dernier est cliqué.

C'est possible en utilisant des fonctions de rappel : on parle donc de **communication montante**, car l'enfant communique avec le parent.

Imaginons un composant `Button` que l'on souhaiterait réutiliser dans notre application : c'est le genre de composants qui peut vraiment être utilisé à plusieurs endroits très différents, tels que le panneau de connexion utilisateur ou lorsque l'on souhaite confirmer un choix.

### Quelles sont les caractéristiques et le rôle d'un bouton ?

Le rôle d'un bouton est d'effectuer une action lorsqu'on clique dessus. Ses caractéristiques peuvent être nombreuses, mais, dans l'exemple qui suit, nous partirons du principe qu'un bouton possède trois caractéristiques :

- Une couleur de remplissage (`fillColor`)
- Un texte (`title`)
- L'action à effectuer lorsqu'on clique dessus (`onPress`)

En reprenant ce que nous avons vu plus haut, implémentons une communication montante grâce à la props `onPress`. De cette manière, on rend le composant générique. Dans notre cas, lors du clic, on souhaite afficher une fenêtre d'alerte avec un titre *BOUM* et un texte différent selon le bouton cliqué.

#### Exemple

```

1 import React from "react";
2 import { Alert, View, Text, TouchableOpacity } from "react-native";
3
4 const MyCustomButton = (props) => (
5   <TouchableOpacity
6     onPress={props.onPress}
7     style={{ backgroundColor: props.fillColor || "white" }}
8   >
9     <Text>{props.title}</Text>
10  </TouchableOpacity>
11 );
12
13 const MyApplicationRoot = () => {
14   return (
15     <View>
16       <MyCustomButton
17         onPress={() => {
18           Alert.alert("BOUM", "Vous avez cliqué sur le bouton 1");
19         }}
20         title="Mon Super Bouton 1"
21         fillColor="red"
22       />
23       <MyCustomButton
24         onPress={() => {
25           Alert.alert("BOUM", "Vous avez cliqué sur le bouton 2");
26         }}
27         title="Mon Super Bouton 2"
28         fillColor="blue"
29       />
30     </View>
31   );
32 };
33
34 export default MyApplicationRoot;

```

Dans cet exemple, on peut constater que notre composant parent `myApplicationRoot` injecte à ses enfants de type `MyCustomButton` des fonctions de rappel générant des effets de bord. Ces effets de bord déclenchent ici une modale système avec un message.

Passer une fonction de rappel donne d'innombrables possibilités. On pourrait par exemple effectuer un appel réseau au clic, changer un état qui conduirait au fait de masquer un composant dans l'interface ou bien changer d'écran dans la navigation.

Nous verrons souvent des références à cette technique quand on nous demandera de *lift the state* (comprendre « faire remonter l'état »). De cette manière, on rend nos composants plus génériques en donnant la possibilité au parent de les spécialiser.

#### Syntaxe À retenir

- Il est possible de déclencher des effets de bord depuis un enfant dans un composant parent : on parle alors de communication montante ou ascendante.
- Pour cela, on injecte une fonction de rappel depuis le parent dans l'enfant en utilisant les props : on parle de communication descendante.
- Ces effets de bord sont nécessaires pour réagir aux événements de l'application, tels que le clic sur un bouton.
- Un composant peut donc avoir une communication bidirectionnelle.

#### Complément

- <https://fr.reactjs.org/docs/lifting-state-up.html>

## VII. Exercice : Appliquez la notion

### Question

[solution n°3 p.22]

Améliorez la petite application avec les compteurs du corrigé de l'exercice 2 (*voir ci-dessous en indice*). Désormais, lorsque l'un des compteurs passe au-dessus d'un nombre défini par la props obligatoire `maxValue`, on déclenche une modale d'alerte qui affiche un message notifiant que le nombre est supérieur à `maxValue` et que ce n'est pas possible.

**Attention, cette modale est déclenchée par le parent et non par l'enfant**, il vous faudra donc sûrement notifier le parent de la valeur courante.

Pour déclencher une modale d'alerte en React Native, utilisez la fonction suivante : `Alert.alert("Votre titre", "Votre message")`.

N'oubliez pas de rajouter l'import `import { Alert } from "react-native"` ; au préalable. Plus d'information ici : <https://reactnative.dev/docs/alert>.

- Choisissez vous-même les valeurs de `maxValue`.
- Factorisez la fonction `onMaxValueReached` qui recevra l'événement de valeur atteinte maximum dans le parent et affichera la modale grâce à un paramètre.

### Indice :

```
1 import React from 'react';
2 import { View, Button, Text } from 'react-native';
3
4 const MyApplicationRoot = () => {
5   return (
6     <View>
7       <Counter />
```

```

8      <Counter defaultValue={7} />
9      <Counter defaultValue={0} />
10    </View>
11  );
12};
13
14const Counter = (props) => {
15  // On déclare un état basé sur la valeur des props
16  const [counter, setCounter] = React.useState(
17    typeof props.defaultValue !== 'undefined' ? props.defaultValue : 10
18  );
19
20  return (
21    <View>
22      <Text>{counter}</Text>
23      <Button
24        onPress={() => {
25          // Au clic on augmente la valeur de 1
26          setCounter((previousValue) => previousValue + 1);
27        }}
28        title="+1"
29      />
30      <Button
31        // Au clic on réduit la valeur de 1
32        onPress={() => {
33          setCounter((previousValue) => previousValue - 1);
34        }}
35        title="-1"
36      />
37    </View>
38  );
39};
40
41export default MyApplicationRoot;

```

## VIII. Valider l'interface de données de ses props et valeurs par défaut

### Objectifs

- Définir une interface précisant le type de données attendu par les props d'un composant
- Améliorer l'expérience de développement
- Sécuriser son code
- Donner des valeurs par défaut aux composants en utilisant les valeurs optionnelles et par défaut

### Mise en situation

Maintenant que nous spécifions des props à nos composants, nous devons définir et aider les développeurs à savoir quelles props possèdent un composant : on parle d'une **interface de communication**. On souhaite également s'assurer que les bonnes données sont passées aux props, afin que le composant fonctionne correctement.

Par exemple, si l'on attend pour la props `onPress` une fonction, on souhaiterait éviter qu'un nombre soit passé. On souhaite également indiquer quelles props sont obligatoires et lesquelles sont optionnelles, et définir dans ce cas une valeur par défaut.

Il existe plusieurs méthodes pour définir l'interface d'un composant, et si l'on n'utilise pas TypeScript (une surcouche à JavaScript qui apporte des fonctionnalités telles que du typage statique), la plus probable sera sûrement d'utiliser `prop-types`.

`prop-types` est un petit utilitaire qui nous permet de définir la forme des props (l'interface). Le cas échéant, la console du navigateur affichera une erreur spécifiant que le contrat d'interface n'est pas respecté.

Installons donc tout d'abord notre utilitaire : `npm install prop-types` pour la version desktop, ou directement `import PropTypes from "prop-types";` pour la version Cloud web.

Reprenons ensuite notre composant bouton plus haut et définissons son interface :

- `title` est une chaîne de caractères requise
- `onPress` est une fonction sans paramètre requise
- `fillColor` est une chaîne de caractères optionnelle

#### Exemple

```
1 import React from "react";
2 import { Alert, View, Text, TouchableOpacity } from "react-native";
3 import PropTypes from "prop-types";
4
5 const MyCustomButton = (props) => (
6   <TouchableOpacity
7     onPress={props.onPress}
8     style={{ backgroundColor: props.fillColor || "white" }}
9   >
10     <Text>{props.title}</Text>
11   </TouchableOpacity>
12 );
13
14 // Ici on ajoute une propriété "propTypes" au prototype de notre composant.
15 MyCustomButton.propTypes = {
16   // On peut déclarer qu'une prop est d'un certain type JS. Par défaut,
17   // elles sont toutes optionnelles.
18   onPress: PropTypes.func.isRequired,
19   title: PropTypes.string.isRequired,
20   fillColor: PropTypes.string
21 };
22
23 const MyApplicationRoot = () => {
24   return (
25     <View>
26       <MyCustomButton
27         onPress={() => {
28           Alert.alert("BOUM", "Vous avez cliqué sur le bouton 1");
29         }}
30         title="Mon Super Bouton 1"
31         fillColor="red"
32       />
33       <MyCustomButton
34         onPress={() => {
35           Alert.alert("BOUM", "Vous avez cliqué sur le bouton 2");
36         }}
37         title="Mon Super Bouton 2"
```

```

38     fillColor="blue"
39   />
40   <MyCustomButton />
41 </View>
42 );
43 };
44
45 export default MyApplicationRoot

```

Dans le cas de notre troisième bouton, nous oublions de passer des props. La console du navigateur affiche donc une série d'erreurs pour nous aider à résoudre le problème, ce qui favorise l'expérience de développement.

Nous avons vu un cas très simple d'utilisation de `propTypes`. Or, nous pouvons nous retrouver face à une situation où nous avons des objets plus complexes en props, ou encore où nous avons besoin d'écrire un petit algorithme conditionnel. Dans ce cas, pour aller plus loin avec `propTypes`, on peut consulter la documentation officielle qui est exhaustive et qui aiguillera très simplement sur son utilisation.

### Rappel

Gardez à l'esprit que `propTypes` ne bloque pas l'utilisation de votre application si la validation des props ne passe pas. C'est seulement un garde-fou de développement afin de nous permettre de détecter plus vite de potentiels bugs où nous aurions passé de mauvaises props.

## Utiliser des valeurs par défaut

Lorsque les props sont optionnelles, il est possible de passer des valeurs par défaut qui seront exploitables par le composant. Pour utiliser des valeurs par défaut, il existe entre autres deux solutions :

- **Tester l'existence de la props**

Par exemple, avec une fonction comme `typeof`, puis donner une valeur de `fallback`.

### Exemple

```

1 const MyComponent = (props) => {
2   const propExists = typeof props.maProps !== "undefined"
3
4   return <Text>{propExists ? props.maProps : "cette chaine est la valeur par défaut si la
5     props n'est pas passée"}</Text>
6 }

```

- **En utilisant `defaultProps` sur le prototype du composant**

La seconde manière consiste à passer des props par défaut au composant en utilisant la syntaxe suivante :

### Exemple

```

1 const MyComponent = (props) => {
2   // ...
3 }
4
5 MyComponent.defaultProps = {
6   maProps: "cette chaine est la valeur par défaut si la props n'est pas passée"
7 }

```



**Syntaxe**    **À retenir**

- On peut valider l'interface des props d'un composant en utilisant une librairie comme `propTypes`. De cette manière, on rend le code plus robuste et on améliore l'expérience de développement.
- On peut utiliser des valeurs par défaut pour les props. Pour cela, plusieurs techniques sont possibles, mais l'idée est de traiter les cas où la props n'est pas présente et de fournir une valeur par défaut.
- Plus tard, nous apprendrons qu'avec des outils comme TypeScript nous pouvons encore améliorer cette validation statique.

**Complément**

- <https://fr.reactjs.org/docs/typechecking-with-proptypes.html>

## IX. Exercice : Appliquez la notion

### Question

[solution n°4 p.23]

Reprenez le corrigé de la partie 3 (*en indice ci-dessous*) et ajoutez les `propTypes` qui correspondent au composant `Counter` afin de définir l'interface externe du composant.

Pour rappel, `defaultValue` est optionnel et `maxValue` et `onMaxValueReached` ne le sont pas.

**Indice :**

```

1 import React from 'react';
2 import { Alert, View, Button, Text } from 'react-native';
3
4 const MyApplicationRoot = () => {
5   // On définit une fonction qui affiche notre modale et reçoit en paramètre la valeur max
   // atteinte
6   const onMaxValueReached = (maxVal) => {
7     Alert.alert(
8       'Erreur',
9       'Le compteur ne peut pas dépasser la valeur maximale de ' + maxVal
10    );
11  };
12
13  return (
14    <View>
15      <Counter maxValue={12} onMaxValueReached={onMaxValueReached} />
16      <Counter
17        defaultValue={7}
18        maxValue={10}
19        onMaxValueReached={onMaxValueReached}
20      />
21      <Counter
22        defaultValue={0}
23        maxValue={7}
24        onMaxValueReached={onMaxValueReached}
25      />
26    </View>
27  );
28 };
29
30 const Counter = (props) => {
31   // On déclare un état basé sur la valeur des props
32   const [counter, setCounter] = React.useState(

```

```

33   typeof props.defaultValue !== 'undefined' ? props.defaultValue : 10
34 );
35
36 return (
37   <View>
38     <Text>{counter}</Text>
39     <Button
40       onPress={() => {
41         if (counter < props.maxValue) {
42           // Au clic on augmente la valeur de 1
43           setCounter((previousValue) => previousValue + 1);
44         } else {
45           props.onMaxValueReached(props.maxValue);
46         }
47       }}
48       title="+1"
49     />
50     <Button
51       // Au clic on réduit la valeur de 1
52       onPress={() => {
53         setCounter((previousValue) => previousValue - 1);
54       }}
55       title="-1"
56     />
57   </View>
58 );
59 };
60
61 export default MyApplicationRoot;

```

## X. Essentiel

## XI. Auto-évaluation

### A. Exercice final

#### Exercice 1

[solution n°5 p.24]

Exercice

Quel hook permet de gérer l'état d'un composant ?

- ☐ useState
- ☐ useReducer
- ☐ useContext
- ☐ useLayoutEffect
- ☐ useEffect

Exercice

Pour communiquer d'un parent à un enfant, on utilise...

- ☐ Les paramètres
- ☐ Les arguments
- ☐ Les props
- ☐ Le state

Exercice

Pour valider l'interface externe d'un composant, on utilise généralement une librairie appelée...

- ☐ prop-validator
- ☐ prop-interface-validator
- ☐ prop-types
- ☐ interface-types

Exercice

Les props sont-elles toujours obligatoires ou peuvent-elles être optionnelles ?

- ☐ Elles sont obligatoires
- ☐ Elles sont optionnelles
- ☐ Elles peuvent être optionnelles et/ou obligatoires

Exercice

Les composants ne peuvent communiquer que dans un sens à la fois.

- ☐ Vrai, seulement un sens est possible
- ☐ Faux, ils peuvent faire les deux à la fois

Exercice

Pour faire varier un compteur défini par le hook state `const [counter, setCounter] = useState(0)`, j'utilise la syntaxe...

- ☐ `setCounter(counter + 1)`
- ☐ `setCounter(previousValue => previousValue + 1)`
- ☐ `setCounter(+1)`
- ☐ `setCounter = counter + 1`

Exercice

Il est possible d'altérer la valeur d'une props directement en la modifiant.

- ☐ Vrai, seulement si c'est un chiffre
- ☐ Vrai, seulement si c'est un booléen
- ☐ Vrai, seulement si on a passé la props `allowEdition={true}` au préalable
- ☐ Faux, les props sont en lecture seule uniquement

### Exercice

Pour donner une valeur par défaut à une props, il faut utiliser la librairie `default-props`.

- ☐ Vrai, sinon on est bloqué
- ☐ Faux, on utilise la propriété du composant `defaultValues` ou on traite les cas par défaut avec du code

### Exercice

On peut passer un composant React via du JSX en props d'un autre composant, par exemple `title={<ComposantTitre>Mon Titre</ComposantTitre>}`.

- ☐ Vrai, n'importe quelle valeur est acceptée en props
- ☐ Faux, seulement des types primitifs (string, number, boolean...)

### Exercice

Si le contrat donné par `prop-types` n'est pas respecté, que se passera-t-il ?

- ☐ Mon application risque de planter
- ☐ Le composant apparaîtra en rouge dans l'interface
- ☐ J'aurai un message d'erreur dans la console en environnement de développement
- ☐ Rien de tout cela

## B. Exercice : Défi

### Question

[solution n°6 p.27]

En vous basant sur ce que nous avons vu précédemment, vous devez réaliser une petite application qui possède un champ de saisie qui reçoit un nombre permettant de définir le pas d'un compteur.

- Le compteur commence à 0 et possède deux boutons, + et -
- Sur les boutons s'affichent +X et -X, où X est le nombre saisi dans le champ du compteur ; si le champ est vide ou n'est pas un nombre, X prend la valeur par défaut de 1
- Quand on clique sur +X ou -X, le compteur affiche la bonne valeur correspondant au changement
- Le composant `Compteur` doit être réutilisable et personnalisable

## Solutions des exercices

## p. 8 Solution n°1

```

1 import React from 'react';
2 import { View, Button, Text } from 'react-native';
3
4 const MyApplicationRoot = () => {
5   // On déclare un état avec pour valeur initiale 0
6   const [counter, setCounter] = React.useState(0);
7   // Par défaut on n'affiche pas l'erreur
8   const [error, setError] = React.useState(false);
9
10  return (
11    <View>
12      {error ? <Text>Le compteur ne peut pas descendre en dessous de 0 !</Text> : null}
13      <Text>{counter}</Text>
14      <Button
15        // Au clic on augmente la valeur de 1
16        onPress={() => {
17          // On réinitialise une éventuelle erreur affichée
18          setError(false);
19
20          // On incrémente de 1
21          setCounter(previous => previous + 1);
22        }}
23        title="+1"
24      />
25      <Button
26        // Au clic on réduit la valeur de 1, si 1 > 0
27        onPress={() => {
28          if (counter > 0) {
29            setCounter(previous => previous - 1);
30          } else {
31            setError(true);
32          }
33        }}
34        title="-1"
35      />
36    </View>
37  );
38 };
39
40 export default MyApplicationRoot

```

## p. 11 Solution n°2

```

1 import React from 'react';
2 import { View, Button, Text } from 'react-native';
3
4 const MyApplicationRoot = () => {
5   return (
6     <View>
7       <Counter />
8       <Counter defaultValue={7} />
9       <Counter defaultValue={0} />
10    </View>

```

```

11 );
12 };
13
14 const Counter = (props) => {
15   // On déclare un état basé sur la valeur des props
16   const [counter, setCounter] = React.useState(
17     typeof props.defaultValue !== 'undefined' ? props.defaultValue : 10
18   );
19
20   return (
21     <View>
22       <Text>{counter}</Text>
23       <Button
24         onPress={() => {
25           // Au clic on augmente la valeur de 1
26           setCounter((previousValue) => previousValue + 1);
27         }}
28         title="+1"
29       />
30       <Button
31         // Au clic on réduit la valeur de 1
32         onPress={() => {
33           setCounter((previousValue) => previousValue - 1);
34         }}
35         title="-1"
36       />
37     </View>
38   );
39 };
40
41 export default MyApplicationRoot;

```

### p. 13 Solution n°3

```

1 import React from 'react';
2 import { Alert, View, Button, Text } from 'react-native';
3
4 const MyApplicationRoot = () => {
5   // On définit une fonction qui affiche notre modale et reçoit en paramètre la valeur max
   // atteinte
6   const onMaxValueReached = (maxVal) => {
7     Alert.alert(
8       'Erreur',
9       'Le compteur ne peut pas dépasser la valeur maximale de ' + maxVal
10    );
11  };
12
13  return (
14    <View>
15      <Counter maxValue={12} onMaxValueReached={onMaxValueReached} />
16      <Counter
17        defaultValue={7}
18        maxValue={10}
19        onMaxValueReached={onMaxValueReached}
20      />
21      <Counter
22        defaultValue={0}

```

```

23     maxValue={7}
24     onMaxValueReached={onMaxValueReached}
25   />
26 </View>
27 );
28 };
29
30 const Counter = (props) => {
31   // On déclare un état basé sur la valeur des props
32   const [counter, setCounter] = React.useState(
33     typeof props.defaultValue !== 'undefined' ? props.defaultValue : 10
34   );
35
36   return (
37     <View>
38       <Text>{counter}</Text>
39       <Button
40         onPress={() => {
41           if (counter < props.maxValue) {
42             // Au clic on augmente la valeur de 1
43             setCounter((previousValue) => previousValue + 1);
44           } else {
45             props.onMaxValueReached(props.maxValue);
46           }
47         }}
48         title="+1"
49       />
50       <Button
51         // Au clic on réduit la valeur de 1
52         onPress={() => {
53           setCounter((previousValue) => previousValue - 1);
54         }}
55         title="-1"
56       />
57     </View>
58   );
59 };
60
61 export default MyApplicationRoot;

```

## p. 17 Solution n°4

```

1 import React from 'react';
2 import { Alert, View, Button, Text } from 'react-native';
3 import PropTypes from "prop-types"
4
5 const MyApplicationRoot = () => {
6   // On définit une fonction qui affiche notre modale et reçoit en paramètre la valeur max
   atteinte
7   const onMaxValueReached = (maxVal) => {
8     Alert.alert(
9       'Erreur',
10      'Le compteur ne peut pas dépasser la valeur maximale de ' + maxVal
11    );
12  };
13
14  return (

```

```

15   <View>
16     <Counter maxValue={12} onMaxValueReached={onMaxValueReached} />
17   <Counter
18     defaultValue={7}
19     maxValue={10}
20     onMaxValueReached={onMaxValueReached}
21   />
22   <Counter
23     defaultValue={0}
24     maxValue={7}
25     onMaxValueReached={onMaxValueReached}
26   />
27 </View>
28 );
29 };
30
31 const Counter = (props) => {
32   // On déclare un état basé sur la valeur des props
33   const [counter, setCounter] = React.useState(
34     typeof props.defaultValue !== 'undefined' ? props.defaultValue : 10
35   );
36
37   return (
38     <View>
39       <Text>{counter}</Text>
40       <Button
41         onPress={() => {
42           if (counter < props.maxValue) {
43             // Au clic on augmente la valeur de 1
44             setCounter((previousValue) => previousValue + 1);
45           } else {
46             props.onMaxValueReached(props.maxValue);
47           }
48         }}
49         title="+1"
50       />
51       <Button
52         // Au clic on réduit la valeur de 1
53         onPress={() => {
54           setCounter((previousValue) => previousValue - 1);
55         }}
56         title="-1"
57       />
58     </View>
59   );
60 };
61
62 Counter.propTypes = {
63   defaultValue: propTypes.number,
64   maxValue: propTypes.number.isRequired,
65   onMaxValueReached: propTypes.func.isRequired
66 }
67
68 export default MyApplicationRoot;

```

### Exercice p. 18 Solution n°5




### Exercice

---

Quel hook permet de gérer l'état d'un composant ?

- ☐ `useComponentState`
- ☒ `useState`
- ☐ `hookState`
- ☐ `useEffect`


 Dans cette liste, seuls `useState` et `useEffect` existent réellement et la bonne réponse est bien évidemment `useState`, que nous avons vu dans le premier chapitre de ce module.

### Exercice

---

Pour communiquer d'un parent à un enfant, on utilise...

- ☐ Les paramètres
- ☐ Les arguments
- ☒ Les props
- ☐ Le state

 Les props permettent à un parent de communiquer des valeurs à un composant enfant : on pourrait qualifier cela de paramètres ou d'arguments, mais, dans une application React, ce type d'échanges est nommé props.


Le state, lui, permet de faire varier une valeur entre les différents rendus de l'application : il est généralement utilisé en synergie avec les props.

### Exercice

---

Pour valider l'interface externe d'un composant, on utilise généralement une librairie appelée...

- ☐ `prop-validator`
- ☐ `prop-interface-validator`
- ☒ `prop-types`
- ☐ `interface-types`


 `prop-types` est la librairie la plus commune pour valider les props et, au jour de la rédaction de ce cours, les autres librairies listées n'existent pas.

### Exercice

---

Les props sont-elles toujours obligatoires ou peuvent-elles être optionnelles ?


- ☐ Elles sont obligatoires
- ☐ Elles sont optionnelles
- ☒ Elles peuvent être optionnelles et/ou obligatoires

 C'est à nous de définir quelles propriétés sont obligatoires ou non, de les valider (par exemple, avec `prop-types`) et de gérer les valeurs par défaut si besoin.

### Exercice

---


Les composants ne peuvent communiquer que dans un sens à la fois.

- ☐ Vrai, seulement un sens est possible
- ☒ Faux, ils peuvent faire les deux à la fois
-  En effet, les composants peuvent parler avec leurs parents en utilisant des props de type fonction et reçoivent des données via des props également.

### Exercice

Pour faire varier un compteur défini par le hook state `const [counter, setCounter] = useState(0)`, j'utilise la syntaxe...

- ☐ `setCounter(counter + 1)`
- ☒ `setCounter(previousValue => previousValue + 1)`
- ☐ `setCounter(+1)`
- ☐ `setCounter = counter + 1`


 La seule syntaxe viable est la deuxième, car elle permet de s'assurer qu'on aura toujours la dernière bonne valeur à incrémenter.

En effet, lorsque le setter d'un hook de state reçoit une fonction en argument, il recevra la valeur précédente de l'état qui permettra de mettre à jour cet état de manière sécurisée, en évitant d'éventuels conflits de modification.

<https://fr.reactjs.org/docs/hooks-reference.html#functional-updates>


### Exercice

Il est possible d'altérer la valeur d'une props directement en la modifiant.

- ☐ Vrai, seulement si c'est un chiffre
- ☐ Vrai, seulement si c'est un booléen
- ☐ Vrai, seulement si on a passé la props `allowEdition={true}` au préalable
- ☒ Faux, les props sont en lecture seule uniquement
-  Il est strictement interdit de modifier une props : on peut cloner la valeur pour la modifier si on souhaite l'altérer.

### Exercice

Pour donner une valeur par défaut à une props, il faut utiliser la librairie `default-props`.

- ☐ Vrai, sinon on est bloqué
- ☒ Faux, on utilise la propriété du composant `defaultValues` ou on traite les cas par défaut avec du code
-  Les valeurs par défaut se gèrent avec du code normal en évaluant la présence de la props, par exemple avec la fonction `typeof`. On peut également attacher un attribut `defaultProps` au prototype du composant.

### Exercice

On peut passer un composant React via du JSX en props d'un autre composant, par exemple `title=<ComposantTitre>Mon Titre</ComposantTitre>`.

- ☒ Vrai, n'importe quelle valeur est acceptée en props
- ☐ Faux, seulement des types primitifs (string, number, boolean...)

🔍 On a entièrement la liberté d'utiliser les props avec le type de données que l'on souhaite.

### Exercice

Si le contrat donné par `prop-types` n'est pas respecté, que se passera-t-il ?

- ☒ Mon application risque de planter
- ☐ Le composant apparaîtra en rouge dans l'interface
- ☒ J'aurai un message d'erreur dans la console en environnement de développement
- ☐ Rien de tout cela

🔍 `prop-types` est un outil de développement type « garde-fou », il n'empêchera en rien le *runtime* de planter. C'est à nous de bien vérifier que notre application fonctionne correctement avant de la déployer.

### p. 20 Solution n°6

```

1 import React from 'react';
2 import { Alert, View, Button, Text, TextInput } from 'react-native';
3 import PropTypes from 'prop-types';
4
5 const MyApplicationRoot = () => {
6   const [step, setStep] = React.useState(1);
7
8   return (
9     <View>
10       <TextInput
11         placeholder="Saisir le pas du compteur"
12         value={step}
13         // On cast en nombre car on reçoit une chaîne de caractère
14         onChangeText={val => setStep(Number(val))}
15         // On notifie que l'on souhaite le clavier des chiffres uniquement
16         keyboardType={'numeric'}
17       />
18       <Counter step={step} />
19     </View>
20   );
21 };
22
23 const Counter = (props) => {
24   const [counter, setCounter] = React.useState(0);
25
26   // Si la props step est un nombre on prends ce nombre sinon le pas défaut à
27   const stepValidated = typeof props.step === "number" ? props.step : 1
28
29   return (
30     <View>
31       <Text>{"Le compteur vaut : " + counter}</Text>
32       <Button
33         onPress={() => {
34           setCounter((previousValue) => previousValue + stepValidated);
35         }}
36         title={`+` + stepValidated}
37       />
38       <Button

```

```
39     onPress={() => {
40         setCounter((previousValue) => previousValue - stepValidated);
41     }}
42     title={`'- ' + stepValidated}
43   />
44 </View>
45 );
46 };
47
48 Counter.propTypes = {
49   step: propTypes.number
50 };
51
52 export default MyApplicationRoot
```