Marquer comme terminé

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

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

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

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

Ce *hook* est constitué\ :

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

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

- à chaque nouveau rendu

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

- à l'initialisation du composant

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

- en cas de modification d'une variable

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

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

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

Mise à jour du compteur de tâche

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

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

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

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

Mise en page

Cette page liste tous les attributs des différents composants.

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

Utilisation de `flex`

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

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

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

Répartition des tailles (*main axis*)

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

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

Alignement selon la direction principale

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

Alignement selon la direction secondaire (*cross axis*)

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

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

Démo

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

Utilisation de la taille de l'écran

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

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

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

Pour aller plus loin

Je vous conseille cet article qui explore les possibilités suivantes\ :

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

Modifié le: lundi 9 septembre 2024, 11:28

◄ Correction TP 2

Choisir un élément

Aller à…

TP ?

?