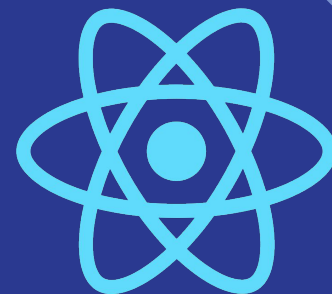


Formation React-Native Composants personnalisés




Bastien NICOLAS
bastien.nicolas@edu.esiee-it.fr

CODING
FACTORY
by ESIEE[tech]

Introduction

- Créer un composant
- Les props
- Cycle de vie du composant
- Les states
- Les hooks
- Exercices

Créer un composant

- Il existe deux type de composants : les composants classes et les composants fonctions
 - Ces deux type de composants permettent de faire la même chose
 - Les composants fonctions ont été introduits plus tard avec les hooks pour améliorer les performances (c'est moins coûteux d'appeler une fonction de d'instancier une classe)
 - Il est recommandé d'utiliser les composants fonctions avec des hooks même si l'on peut régulièrement retrouver des composants classes dans certains projets/modules
 - Pour un composant fonction, la fonction qui le définit est appelée à chaque rendu du composant (ça peut se produire plusieurs fois par secondes), attention à ce qu'on met dedans pour ne pas dégrader les performances !
 - On met un composant par fichier
- 

Créer un composant

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

const FunctionComponent = () => {
  return (
    <View>
      <Text>Mon composant fonction</Text>
    </View>
  );
};

export default FunctionComponent;
```

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

class ClassComponent extends React.Component {
  render() {
    return (
      <View>
        <Text>Mon composant classe</Text>
      </View>
    );
  }
}

export default ClassComponent;
```

```
<>
  <FunctionComponent />
  <ClassComponent />
</>
```

Les props

- Les props sont des données qui sont passées à un composant lors de son appel et qui sont utilisés dans le composant (vue ou traitement)
- Les props sont en lecture seul au sein du composant
- Si un props change, le composant est automatiquement mise à jour



Les props

```
const FunctionComponent = props => {  
  return (  
    <View>  
      <Text>{props.name}</Text>  
      <Text>Mon composant fonction</Text>  
    </View>  
  );  
};
```

```
export default FunctionComponent;
```

```
<FunctionComponent name={'Toto'} />
```

```
class ClassComponent extends React.Component {  
  render() {  
    return (  
      <View>  
        <Text>{this.props.name}</Text>  
        <Text>Mon composant classe</Text>  
      </View>  
    );  
  }  
}
```

```
export default ClassComponent;
```

```
<ClassComponent name={'Toto'} />
```

Les props

On peut passer en props tout type de données comme des fonction pour déclencher des callbacks par exemple

```
const FunctionComponent = props => {  
  const {onCustomAction} = props;  
  
  return (  
    <TouchableOpacity onPress={onCustomAction}>  
      <Text>Mon composant fonction</Text>  
    </TouchableOpacity>  
  );  
};
```

```
<FunctionComponent  
  onCustomAction={() => console.log('Component pressed')}  
>
```

Les props

L'enfant d'un composant peut être récupéré avec le props "children"

```
const FunctionComponent = props => {  
  const {children} = props;  
  
  return <View>{children}</View>;  
};
```

```
<FunctionComponent>  
  <Text>Toto</Text>  
</FunctionComponent>
```


Les props

On peut faire hériter toutes les props d'un composant à un composant enfant de cette manière

```
const FunctionComponent = props => {  
  return (  
    <View>  
      <Image {...props} />  
    </View>  
  );  
};
```

```
<FunctionComponent  
  source={require(id: './src/assets/no-profile-picture.png')}  
  style={styles.image}  
>
```



Cycle de vie du composant

- Les composants classes utilisent les méthodes de classe “componentDidMount” et “componentWillUnmount”
- Les composants fonctions utilisent le hook “useEffect” (on ne peut pas déclarer de méthode dans une fonction)
 - Ce hook useEffect (pour cet usage précis) prend comme premier paramètre fonction qui est appelée à la création du composant
 - Cette fonction renvoie une autre fonction qui est appelée avant la destruction du composant
 - Le second paramètre de useEffect est un tableau vide (important)



Cycle de vie du composant

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

const FunctionComponent = props => {
  useEffect( effect: () => {
    console.log('le composant est opérationnel');

    return () => {
      console.log('le composant va être détruit');
    };
  }, deps: []);

  return (
    <View>
      <Text>Mon composant fonction</Text>
    </View>
  );
};
```

```
class ClassComponent extends React.Component {
  componentDidMount() {
    console.log('le composant est opérationnel');
  }

  componentWillUnmount() {
    console.log('le composant va être détruit');
  }

  render() {
    return (
      <View>
        <Text>Mon composant classe</Text>
      </View>
    );
  }
}
```

Les states

- Les states sont des variables qui vont permettrent de faire évoluer le composant en temps réel quand leurs valeurs changent
- On peut changer leurs valeurs au sein du composant avec des fonctions prévues à cet effet
- Les states peuvent contenir tout type de données



Les states

```
import React, {useState} from 'react';
import {View, Text, TouchableOpacity} from 'react-native';

const FunctionComponent = () => {
  const [value, setValue] = useState( initialState: 0);

  const onTouchablePress = () => {
    setValue(value + 1);
  };

  return (
    <View>
      <TouchableOpacity onPress={onTouchablePress}>
        <Text>{value}</Text>
      </TouchableOpacity>
    </View>
  );
};
```

```
class ClassComponent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      value: 0,
    };
  }

  onTouchablePress = () => {
    this.setState( state: {
      value: this.state.value + 1,
    });
  };

  render() {
    return (
      <View>
        <TouchableOpacity onPress={this.onTouchablePress}>
          <Text>{this.state.value}</Text>
        </TouchableOpacity>
        <Text>Mon composant classe</Text>
      </View>
    );
  }
}
```

Les hooks

- useState
- useEffect
- useCallback
- useMemo

```
import {useState, useEffect, useCallback, useMemo} from 'react';
```

Ce sont les principaux hook fournis par React mais il en existe d'autres, certains modules en proposent également

Nous pouvons également créer les nôtres



Les hooks - useState

```
const [value, setValue] = useState( initialState: 0);  
const [isValid, setIsValid] = useState( initialState: false);  
const [myArray, setMyArray] = useState( initialState: [1, 2, 3]);  
const [data, setData] = useState(props.data);
```

```
const [email, setEmail] = useState( initialState: '');  
const [isValid, setIsValid] = useState( initialState: true);  
  
return (  
  <>  
    <TextInput  
      value={email}  
      //onChangeText={value => setEmail(value)}  
      onChangeText={setEmail}  
      placeholder={'Email'}  
      style={isValid ? styles.input : styles.inputError}  
    />  
    <Text>Email : {email}</Text>  
  </>  
)
```

Les hooks - useEffect

- Ce hook permet d'exécuter une fonction lorsqu'une variable change de valeur (props ou state en général)
- Il prend deux paramètres :
 - La fonction à exécuter
 - Un tableau de dépendances qui sont les variables à écouter et qui déclenchent ce hook quand elles changent de valeur
- Il permet également de suivre le cycle de vie d'un composant comme vu précédemment mais le tableau de dépendance doit être vide



Les hooks - useEffect

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

const FunctionComponent = props => {
  useEffect( effect: () => {
    console.log('le composant est opérationnel');

    return () => {
      console.log('le composant va être détruit');
    };
  }, deps: []);

  return (
    <View>
      <Text>Mon composant fonction</Text>
    </View>
  );
};
```

```
const FunctionComponent = props => {
  const [value, setValue] = useState();

  useEffect( effect: () => {
    console.log('le props name a changé');
  }, deps: [props.name]);

  useEffect( effect: () => {
    console.log('le state value a changé');
  }, deps: [value]);
};
```

Les hooks - useCallback

- Ce hook permet de définir des fonctions de callback
- Il permet d'améliorer les performances car la fonction n'est pas redéfinie à chaque rendu du composant
- Il prend deux paramètres :
 - La fonction de callback
 - Un tableau de dépendances qui quand elles changes permettent de redéfinir la fonction de callback
- Si on fait une fonction de callback sans ce hook ça fonctionne mais ce n'est pas optimisé !



Les hooks - useCallback

```
const FunctionComponent = () => {  
  const [value, setValue] = useState( initialState: 0);  
  
  const onTouchablePress = () => {  
    setValue(value + 1);  
  };  
  
  return (  
    <View>  
      <TouchableOpacity onPress={onTouchablePress}>  
        <Text>{value}</Text>  
      </TouchableOpacity>  
    </View>  
  );  
};
```



```
const FunctionComponent = () => {  
  const [value, setValue] = useState( initialState: 0);  
  
  const onTouchablePress = useCallback( callback: () => {  
    setValue(value + 1);  
  }, deps: [value]);  
  
  return (  
    <View>  
      <TouchableOpacity onPress={onTouchablePress}>  
        <Text>{value}</Text>  
      </TouchableOpacity>  
    </View>  
  );  
};
```



Les hooks - useMemo

- Ce hook permet de stocker une donnée calculée à partir de variables
- Il permet également d'optimiser les performances en ne calculant pas la valeur à chaque rendu mais seulement quand les dépendances changent
- Il prend deux paramètres :
 - Une fonction qui calcule la valeur à mémoriser
 - Un tableau de dépendances qui quand elles changes permettent de recalculer la valeur à mémoriser
- Il fonctionne de la même manière que useCallback (useCallback est en fait un useMemo qui renvoie une fonction)



Les hooks - useMemo

```
const [data, setData] = useState( initialState: []);  
const [search, setSearch] = useState( initialState: '');  
  
const dataFiltered = () => {  
  return data.filter(o => o.text.includes(search));  
};
```




```
const [data, setData] = useState( initialState: []);  
const [search, setSearch] = useState( initialState: '');  
  
const dataFiltered = useMemo( factory: () => {  
  return data.filter(o => o.text.includes(search));  
}, deps: [data, search]);
```



Exercice 1 - Formulaire avec validation

- Gérer les valeurs des champs avec des `useState`
- Si le mot de passe ne contient pas au moins 3 caractères, afficher les bordures du champ en rouge (utiliser `useState` et `useEffect`)
- Si la confirmation du mot de passe est différente du mot de passe, afficher les bordures du champ en rouge (utiliser `useMemo`)
- Le clic sur le bouton Envoyer doit afficher une alerte “Bonjour <prénom> <nom>, votre mot de passe est <mot de passe>” (utiliser `useCallback`)
- Bonus : gérer les erreurs seulement à la fin de la saisie de chaque champ, voir la fonction `onEndEditing` de `TextInput`

Inscription



Exercice 2 - Todo list

- Faire un composant `TodoListScreen` (la page) et un composant `TodoListItem` (une Todo de la liste)
- On doit pouvoir ajouter des Todo à la liste
- On doit pouvoir supprimer les Todo (l'action est remontée via un props à la page qui gère la suppression)
- On doit pouvoir rechercher les Todo par titre (ça filtre la liste qui s'affiche mais les Todo ne doivent pas être effacées)

The image shows a mobile application interface for a todo list. At the top, there is a search bar with the placeholder text "Rechercher ...". Below this, a list of three items is displayed: "Faire les courses", "Sortir le chien", and "Appeler maman". Each item is contained within a light gray rectangular box and has a red button labeled "Supprimer" (Delete) to its right. At the bottom of the screen, there is a text input field with the placeholder "A faire ..." and a green button labeled "Ajouter" (Add) to its right.