ReactJS
Maîtriser le framework

Sommaire

Hooks

Les Hooks sont une nouveauté de React 16.8.

Ils permettent de bénéficier d'un état local et d'autres fonctionnalités de React sans avoir à écrire de classes.

Ils ne peuvent donc s'utiliser que dans des composants « fonction » qui ne bénéficiaient pas d'un state auparavant.

Hooks

Les Hooks sont des fonctions JavaScript, mais ils imposent deux règles supplémentaires :

- Appelez les Hooks uniquement au niveau racine. N’appelez pas de Hooks à l’intérieur de boucles, de code conditionnel ou de fonctions imbriquées.
- Appelez les Hooks uniquement depuis des composants « fonctions ». N’appelez pas les Hooks depuis des fonctions JavaScript classiques.

Hook useState

```
import React, { useState } from 'react';
function Example() {
  // Déclaration d'une nouvelle variable d'état, que l'on appellera “count”
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Vous avez cliqué {count} fois</p>
      <button onClick={() => setCount(count + 1)}>
        Cliquez ici
      </button>
    </div>
  );
}
```

Hook useEffect

Le Hook d’effet, useEffect, permet aux fonctions composants de gérer des effets de bord.

Il joue le même rôle que componentDidMount, componentDidUpdate, et componentWillUnmount dans les classes React, mais au travers d’une API unique.

Hook useEffect

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);
  // Équivalent à componentDidMount plus componentDidUpdate :
  useEffect(() => {
    // Mettre à jour le titre du document en utilisant l'API du navigateur
    document.title = `Vous avez cliqué ${count} fois`;
  });
  return (
    <div>
      <p>Vous avez cliqué {count} fois</p>
      <button onClick={() => setCount(count + 1)}>
        Cliquez ici
      </button>
    </div>
  );
}
```

Améliorer une application React

Les périmètres d'erreurs (error boundaries, NdT) sont des composants React qui interceptent toute erreur JavaScript survenant dans l'arbre de composants de leurs enfants, loguent ces erreurs, et affichent une UI de remplacement au lieu de l'arbre de composants qui a planté.

Les périmètres d'erreurs capturent les erreurs survenant dans le rendu, les méthodes de cycle de vie, et les constructeurs de tout l'arbre en-dessous d'eux.

Améliorer une application React

Un composant basé classe devient un périmètre d'erreur s'il définit au moins une des méthodes de cycle de vie static getDerivedStateFromError() ou componentDidCatch().

Mettre à jour votre état local au sein de ces méthodes vous permet d'intercepter une erreur JavaScript non gérée dans l'arbre en-dessous de vous, et d'afficher à la place une UI de remplacement.

Améliorer une application React


```
export default class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError(error) {
    // On met à jour l'état afin que le prochain rendu affiche  l'UI de remplacement.
    return { hasError: true };
  }
  render() {
    if (this.state.hasError) {
      // Vous pouvez afficher ici n'importe quelle UI de secours
      return <h1>Ça sent le brûlé.</h1>;
    }
    return this.props.children;
  }
}
```

Améliorer une application React


```
function App() {
return (
    <div className="App">
      <ErrorBoundary>
        <Navigation />
        <h1>Accueil</h1>
      </ErrorBoundary>
    </div>
  );
}
```

Améliorer une application React

En React, il est courant pour un composant de renvoyer plusieurs éléments.

Les fragments nous permettent de grouper une liste d’enfants sans ajouter de nœud supplémentaire au DOM.

Améliorer une application React


```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

Améliorer une application React


```
class Columns extends React.Component {
  render() {
    return (
      <div>
        <td>Bonjour</td>
        <td>Monde</td>
      </div>
    );
  }
}
```

Améliorer une application React


```
<table>
  <tr>
    <div>
      <td>Bonjour</td>
      <td>Monde</td>
    </div>
  </tr>
</table>
```

Améliorer une application React


```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Bonjour</td>
        <td>Monde</td>
      </React.Fragment>
    );
  }
}
```

Améliorer une application React


```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Bonjour</td>
        <td>Monde</td>
      </>
    );
  }
}
```

Améliorer une application React

Dans une application React typique, les données sont passées de haut en bas (du parent à l'enfant) via les props, mais cela peut devenir lourd pour certains types de props (ex. les préférences régionales, le thème de l'interface utilisateur) qui s'avèrent nécessaires pour de nombreux composants au sein d'une application.

Le Contexte offre un moyen de partager des valeurs comme celles-ci entre des composants sans avoir à explicitement passer une prop à chaque niveau de l'arborescence.

Améliorer une application React

```
// Le Contexte nous permet de transmettre une prop profondément dans l'arbre des composants sans la faire passer explicitement à travers tous les composants.

// Crée un contexte pour le thème (avec "light" comme valeur par défaut).


export const ThemeContext = React.createContext('light');
```

Améliorer une application React

```
class App extends React.Component {
  render() {
    // Utilise un Provider pour passer le thème plus bas dans l'arbre.
    // N'importe quel composant peut le lire, quelle que soit sa profondeur.
    // Dans cet exemple, nous passons "dark" comme valeur actuelle.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
```

Améliorer une application React

```
// Un composant au milieu n'a plus à transmettre
explicitement le thème
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
```

Améliorer une application React


```
class ThemedButton extends React.Component {
  // Définit un contextType pour lire le contexte de thème actuel.  React va trouver le Provider de thème ancêtre le plus proche et utiliser sa valeur.
  // Dans cet exemple, le thème actuel est “dark”.

static contextType = ThemeContext;
render() {
    return <Button theme={this.context} />;
  }
}
```

TypeScript

TypeScript est un langage de programmation développé par Microsoft.

C’est un sur-ensemble typé de JavaScript, et il fournit son propre compilateur.

Étant un langage typé, TypeScript peut trouver des erreurs et bugs lors de la compilation, bien avant que l’application ne soit déployée. Vous trouverez plus d’informations sur l’utilisation de TypeScript avec React:

https://github.com/Microsoft/TypeScript-React-Starter#typescript-react-starter

```
npx create-react-app my-app --template typescript
```

TypeScript

Pour utiliser TypeScript, vous devez :

- Ajouter la dépendance TypeScript dans votre projet
- Configurer les options du compilateur TypeScript
- Utiliser les extensions appropriées pour vos fichiers
- Ajouter les définitions de type pour les bibliothèques que vous utilisez

Lift State Up

Plusieurs composants ont souvent besoin de refléter les mêmes données dynamiques.

Nous conseillons de faire remonter l'état partagé dans leur ancêtre commun le plus proche.

L'élément enfant peut retourner des valeurs à l'élément parent via une fonction dans les « props ».

Higher-Order Components

Un composant d'ordre supérieur (Higher-Order Component ou HOC, NdT) est une technique avancée de React qui permet de réutiliser la logique de composants.

Les HOC ne font pas partie de l'API de React à proprement parler, mais découlent de sa nature compositionnelle.

Concrètement, un composant d'ordre supérieur est une fonction qui accepte un composant et renvoie un nouveau composant.

https://github.com/Sidibedev/highOrderComponent

REDUX

Redux permet une gestion de "states globaux". Grâce à Redux, vous palliez les faiblesses des props qui ne sont accessibles qu'en lecture seule, ou des states qui sont liés à un unique composant, et donc seulement localement.

Redux permet donc de faciliter le développement de toute application dès que celle-ci implique l'existence de composants dépendant les uns des autres.

https://react-redux.js.org/introduction/getting-started

```
npm install @reduxjs/toolkit react-redux
npx create-react-app my-app --template redux
```

REDUX


```
import React from 'react'
import ReactDOM from 'react-dom/client'

import { Provider } from 'react-redux'
import store from './store'

import App from './App'

// As of React 18
const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(
  <Provider store={store}>
    <App />
  </Provider>
)
```

Tests Unitaires

React Testing Library fournit un ensemble de fonctions utilitaires pour tester des composants React sans dépendre de leurs détails d'implémentation.

Cette approche facilite le changement de conception interne et vous aiguille vers de meilleures pratiques en termes d'accessibilité.

Même s'il ne fournit pas de moyen pour réaliser le rendu « superficiel » d'un composant (sans ses enfants), on peut y arriver avec un harnais tel que Jest et ses mécanismes d'isolation.

Tests Unitaires

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders Calculatrice', () => {
  render(<App />);
  const linkElement = screen.getByText(/calculatrice/i);
  expect(linkElement).toBeInTheDocument();
});
```

https://testing-library.com/docs/react-testing-library/intro/

ReactDOMServer

L’objet ReactDOMServer vous permet de produire sous forme de texte statique le balisage nécessaire à l’affichage de composants.

En règle générale, on l’utilise avec un serveur Node.

Il va donc falloir développer:

- Une partie côté client en React.
- Une partie côté serveur avec Node (et Express).

React Native

React Native est un framework d'applications mobiles open source créé par Facebook.

Il est utilisé pour développer des applications pour Android , iOS et UWP en permettant aux développeurs d’utiliser React avec les fonctionnalités natives de ces plateformes.

React Native

Les principes de fonctionnement de React Native sont pratiquement identiques à ceux de React, à la différence que React Native ne manipule pas le DOM via le DOM virtuel.

Il s'exécute dans un processus en arrière-plan (qui interprète le code JavaScript écrit par les développeurs) directement sur le terminal et communique avec la plate-forme native via une passerelle de sérialisation, asynchrone et par lots.

React Native n'utilise pas HTML. Au lieu de cela, les messages du thread JavaScript sont utilisés pour manipuler des vues natives.

React Native


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

const HelloWorldApp = () => {
  return (
    <Text>Hello world!</Text>
  );
}
export default HelloWorldApp;

// Skip this line if using Create React Native App
AppRegistry.registerComponent('HelloWorld', () => HelloWorldApp);

// The ReactJS code can also be imported into another component with the following code:
import HelloWorldApp from './HelloWorldApp';
```