Développement mobile avec React Js

Du 06 au 10 septembre 2021

* INSTALLATION
* Les PROPS
* Les COMPOSANTS
* LE CYCLE DE VIE
* REACT JS ET ROUTING
* REACT HOOKS
* OPTIMISATION
* REDUX
* TP1: UN EXEMPLE D'APPLICATION CRUD SIMPLE
* TP2: DEVELOPPEMENT D'UNE APPLICATION AVEC RE CT JS OU ANGULAR JS

1- INSTALLATION

* Pour créer une nouvelle application, vous pouvez choisir l'une des méthodes suivantes :

- **Utiliser npx:** npx create-react-app react-frontend
- **Utiliser npm:** npm init react-app react-frontend
- **Utiliser du fil:** yarn create react-app react-frontend

*Utilisation de JSX

JSX ressemble à un HTML normal dans la plupart des cas. Nous l'avons déjà utilisé dans la partie Configuration de l'environnement. Regardez le code de **App.jsx** où nous retournons **div** .

App.jsx

```jsx
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            <h1>Header</h1>
            <h2>Content</h2>
            <p>This is the content!!!</p>
         </div>
      );
   }
}
export default App;
```

Utilisation de l'état

L'état est le lieu d'où proviennent les données. L'exemple de code suivant montre comment créer un composant avec état à l'aide de la syntaxe EcmaScript2016.
On va créer le fichier main.js

```
import React from 'react';

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

      this.state = {
         header: "Header from state...",
         content: "Content from state..."
      }
   }
   render() {
      return (
         <div>
            <h1>{this.state.header}</h1>
            <h2>{this.state.content}</h2>
         </div>
      );
   }
}
export default App;
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

ReactDOM.render(<App />, document.getElementById('app'));
```

Header from state...

Content from state...

Utiliser des accessoires(PROPS)

* Lorsque nous avons besoin de données immuables dans notre composant, nous pouvons simplement ajouter des accessoires à la fonction **reactDOM.render()** dans **main.js** et les utiliser dans notre composant.

App.jsx

```
import React from 'react';

class App extends React.Component {
    render() {
        return (
            <div>
                <h1>{this.props.headerProp}</h1>
                <h2>{this.props.contentProp}</h2>
            </div>
        );
    }
}
export default App;
```

* LES PROPS

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

ReactDOM.render(<App headerProp = "Header from props..." contentProp = "Conte
   from props..."/>, document.getElementById('app'));

export default App;
```

Accessoires par défaut

Vous pouvez également définir les valeurs de propriété par défaut directement sur le constructeur du composant au lieu de l'ajouter à l' élément **reactDom.render()** .

App.jsx

```jsx
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            <h1>{this.props.headerProp}</h1>
            <h2>{this.props.contentProp}</h2>
         </div>
      );
   }
}
App.defaultProps = {
   headerProp: "Header from props...",
   contentProp:"Content from props..."
}
export default App;
```

main.js

```jsx
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

ReactDOM.render(<App/>, document.getElementById('app'));
```

Header from props...

Content from props...

État et accessoires

* L'exemple suivant montre comment combiner l' **état** et les accessoires dans votre application. Nous définissons l'état dans notre composant parent et le transmettons à l'arborescence des composants à l'aide de **props** . A l' intérieur du **rendu** fonction, nous fixons **headerProp** et **contentProp** utilisés dans les composants de l' enfant.

main.js

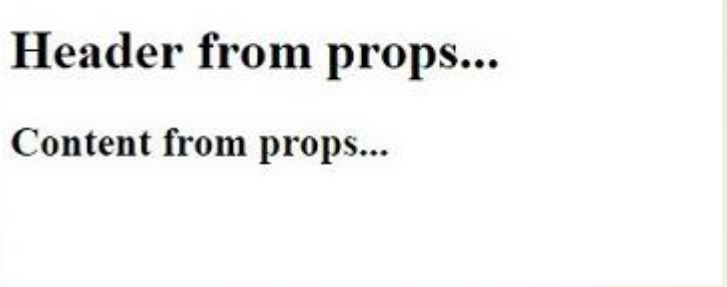
```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

ReactDOM.render(<App/>, document.getElementById('app'));
```

App.jsx

```jsx
import React from 'react';

class App extends React.Component {
   constructor(props) {
      super(props);
      this.state = {
         header: "Header from props...",
         content: "Content from props..."
      }
   }
   render() {
      return (
         <div>
            <Header headerProp = {this.state.header}/>
            <Content contentProp = {this.state.content}/>
         </div>
      );
   }
}
class Header extends React.Component {
   render() {
      return (
         <div>
            <h1>{this.props.headerProp}</h1>
         </div>
      );
   }
}
class Content extends React.Component {
   render() {
      return (
         <div>
            <h2>{this.props.contentProp}</h2>
         </div>
      );
   }
}
export default App;
```



Cycle de vie des composants

* haque composant de React a un cycle de vie que vous pouvez surveiller et manipuler au cours de ses trois phases principales.
* Les trois phases sont : le **montage** , la **mise à jour** et le **démontage.**

componentWillMount est exécuté avant le rendu, à la fois côté serveur et côté client.

componentDidMount est exécuté après le premier rendu uniquement côté client. C'est là que les requêtes AJAX et les mises à jour DOM ou d'état doivent se produire.

componentWillReceiveProps est invoqué dès que les accessoires sont mis à jour avant qu'un autre rendu ne soit appelé. Nous l'avons déclenché à partir de **setNewNumber** lorsque nous avons mis à jour l'état.

* Cycle de vie des composants

- **shouldComponentUpdate** doit renvoyer **une** valeur **vraie** ou **fausse** . Cela déterminera si le composant sera mis à jour ou non. Ceci est défini sur **true** par défaut. Si vous êtes sûr que le composant n'a pas besoin d'être rendu après la mise à jour de l' **état** ou des **accessoires** , vous pouvez renvoyer **une** valeur **false** .
- **componentWillUpdate** est appelé juste avant le rendu.
- **componentDidUpdate** est appelé juste après le rendu.
- **componentWillUnmount** est appelé après le démontage du composant du dom. Nous **démontons** notre composant dans **main.js** .

* Dans l'exemple suivant, nous allons définir l' **état** initial dans la fonction constructeur. Le **setNewnumber** est utilisé pour mettre à jour l' **état** . Toutes les méthodes de cycle de vie se trouvent à l'intérieur du composant Content.

App.jsx

```jsx
import React from 'react';

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

        this.state = {
            data: 0
        }
        this.setNewNumber = this.setNewNumber.bind(this)
    };
    setNewNumber() {
        this.setState({data: this.state.data + 1})
    }
    render() {
        return (
            <div>
                <button onClick = {this.setNewNumber}>INCREMENT</button>
                <Content myNumber = {this.state.data}></Content>
            </div>
        );
    }
}
```

```
class Content extends React.Component {
   componentWillMount() {
      console.log('Component WILL MOUNT!')
   }
   componentDidMount() {
      console.log('Component DID MOUNT!')
   }
   componentWillReceiveProps(newProps) {
      console.log('Component WILL RECIEVE PROPS!')
   }
   shouldComponentUpdate(newProps, newState) {
      return true;
   }
   componentWillUpdate(nextProps, nextState) {
      console.log('Component WILL UPDATE!');
   }
   componentDidUpdate(prevProps, prevState) {
      console.log('Component DID UPDATE!')
   }
   componentWillUnmount() {
      console.log('Component WILL UNMOUNT!')
   }
   render() {
      return (
         <div>
            <h3>{this.props.myNumber}</h3>
         </div>
      );
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

ReactDOM.render(<App/>, document.getElementById('app'));

setTimeout(() => {
   ReactDOM.unmountComponentAtNode(document.getElementById('app'));}, 10000);
```

66

Installer un routeur React

Un moyen simple d'installer le **react-router** consiste à exécuter l'extrait de code suivant dans la fenêtre d' **invite de commande**

npm install react-router

Dans cette étape, nous allons créer quatre composants. Le composant **App** sera utilisé comme un menu d'onglet. Les trois autres composants **(Accueil), (À propos)** et **(Contact)** sont rendus une fois que l'itinéraire a changé.

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, Route, Link, browserHistory, IndexRoute } from 'react-router

class App extends React.Component {
   render() {
      return (
         <div>
            <ul>
            <li>Home</li>
            <li>About</li>
            <li>Contact</li>
            </ul>
            {this.props.children}
         </div>
      )
   }
}
```

```
export default App;

class Home extends React.Component {
   render() {
      return (
         <div>
            <h1>Home...</h1>
         </div>
      )
   }
}
export default Home;

class About extends React.Component {
   render() {
      return (
         <div>
            <h1>About...</h1>
         </div>
      )
   }
}
```

```
export default Home;

class About extends React.Component {
   render() {
      return (
         <div>
            <h1>About...</h1>
         </div>
      )
   }
}
export default About;

class Contact extends React.Component {
   render() {
      return (
         <div>
            <h1>Contact...</h1>
         </div>
      )
   }
}
export default Contact;
```

main.js

```
ReactDOM.render((
   <Router history = {browserHistory}>
      <Route path = "/" component = {App}>
         <IndexRoute component = {Home} />
         <Route path = "home" component = {Home} />
         <Route path = "about" component = {About} />
         <Route path = "contact" component = {Contact} />
      </Route>
   </Router>
), document.getElementById('app'))
```

Introduction aux Hooks

Les *Hooks* sont arrivés avec React 16.8. Ils vous permettent de bénéficier d'un état local et d'autres fonctionnalités de React sans avoir à écrire une classe. Cet exemple est un hook d'état et il affiche un compteur. Quand vous faites click sur le bouton, la valeur augmente.

```
import React, { useState } from 'react';

function Example() {
  // Déclare une nouvelle variable d'état, qu'on va appeler « 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 d’état

Dans le code ci-dessus, useState est un *Hook*. Nous l’appelons au sein d’une fonction composant pour y ajouter un état local. React va préserver cet état d’un affichage à l’autre. useState retourne une paire : la valeur de l’état *actuel* et une fonction qui vous permet de la mettre à jour. Vous pouvez appeler cette fonction depuis un gestionnaire d’événements, par exemple. Elle est similaire à this.setState dans une classe, à ceci près qu’elle ne fusionne pas l’ancien état et le nouveau.

Le seul argument de useState est l’état initial. Dans l’exemple précédent, c’est 0 puisque notre compteur démarre à zéro. Remarquez que contrairement à this.state, ici l’état n’est pas nécessairement un objet, même si ça reste possible. L’argument d’état initial n’est utilisé que pour le premier affichage

Hook d’effet

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.

```
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>
  );
}
```

Lorsque vous utilisez UseEffect, vous dites à React de lancer votre fonction « d'éffet » après avoir mis à jour le DOM. Les éffects étant déclaré au sein du composant, ils ont accès à ses PROPS et son état. Par défaut, React execute les effects après chaque affichage, y compris le premier.

Les effets peuvent aussi préciser comment les « nettoyer » en renvoyant une fonction. Par exemple, ce composant utilise un effet pour s'abonner au statut de connexion d'un ami, et se nettoie en résiliant l'abonnement :

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Chargement...';
  }
  return isOnline ? 'En ligne' : 'Hors-ligne';
}
```

Tout comme avec UseState, vous pouvez utiliser plus d'un seul effect dans un composant:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `Vous avez cliqué ${count} fois`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  // ...
```

Construire vos propres Hooks

Un peu plus tôt sur cette page nous avons présenté un composant FriendStatus qui s'appelle les HOOKS UseState et UseEffect pour s'aborner à l'état de connexion d'un amis. On veut utiliser cette logique d'abornement dans un autre composant.

tout d'abord, nous allons extraire cette logique dans un HOOK personnalisé UseFriendStatus

```
import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

Nous pouvons l’utiliser dans les deux composants

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Chargement...';
  }
  return isOnline ? 'En ligne' : 'Hors-ligne';
}
```

```
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

Premiers pas avec React Redux

React Redux est la couche officielle de liaisons d'interface utilisateur React pour Redux . Il permet à vos composants React de lire les données d'un magasin Redux et d'envoyer des actions au magasin pour mettre à jour l'état.

Utilisation de Create React App

```
# Redux + Plain JS template
npx create-react-app my-app --template redux

# Redux + TypeScript template
npx create-react-app my-app --template redux-typescript
```

Une application React existante

```
# If you use npm:
npm install react-redux

# Or if you use Yarn:
yarn add react-redux
```

Vous devrez également installer Redux et configurer un magasin Redux dans votre application.

Si vous utilisez TypeScript, les types React Redux sont conservés séparément dans DefinitelyTyped, mais inclus en tant que dépendance du react-reduxpackage, ils doivent donc être installés automatiquement.
Si vous devez toujours les installer manuellement, exécutez :

```
npm install @types/react-redux
```

Techniques pour optimiser les performances sur une application React

À utiliser React.Fragment pour éviter d'ajouter des nœuds supplémentaires au DOM

Lorsque vous travaillez avec React, il existe des cas où vous devrez rendre plusieurs éléments ou renvoyer un groupe d'éléments associés. Voici un exemple :

```
function App() {

 return (

        <h1>Hello React!</h1>
         <h1>Hello React Again!</h1>
        );


          }
```

Si vous essayez d'exécuter votre application avec le code ci-dessus, vous rencontrerez une erreur indiquant que Adjacent JSX elements must be wrapped in an enclosing tag cela implique que vous devez encapsuler les deux éléments dans un div parent.

```
function App() {
  return (
    <div>
      <h1>Hello React!</h1>
      <h1>Hello React Again!</h1>
    </div>
  );
}
```

Utiliser la version de production

Une autre façon d'optimiser une application React consiste à vous assurer de regrouper votre application pour la production avant de la déployer.

Par défaut, votre application est en mode développement, ce qui signifie que React inclura des avertissements utiles. Cela peut être très utile pendant que vous développez, mais cela peut augmenter la taille de votre application et ralentir les réponses par rapport à d'habitude. Si votre projet est généré avec create-react-app, vous pouvez résoudre ce problème en l'exécutant npm run buil davant le déploiement, ce qui créera une version prête pour la production de votre application dans un build/dossier que vous pourrez ensuite déployer. Vous pouvez confirmer si votre application est en mode développement ou production à l'aide des outils de développement React .

Utilisez React.Suspense et React.Lazy pour les composants de chargement différé

Le chargement paresseux est une excellente technique pour optimiser et accélérer le temps de rendu de votre application.
L'idée du chargement différé est de charger un composant uniquement lorsque cela est nécessaire.
React est fourni avec l' React.lazyAPI afin que vous puissiez effectuer une importation dynamique en tant que composant standard. Ici, au lieu de charger votre composant habituel comme ceci :

```
import LazyComponent from './LazyComponent';
```

Vous pouvez réduire le risque de performances en utilisant la méthode paresseuse pour rendre un composant.

```
const LazyComponent = React.lazy(() => import('./LazyComponent'));
```

React.lazy prend une fonction qui doit appeler un dynamic import(). Cela renverra ensuite un Promise qui se résout en un module avec une defaultexportation contenant un composant React. Le composant paresseux doit être rendu à l'intérieur d'un Suspensecomposant, ce qui vous permet d'ajouter du contenu de secours en tant qu'état de chargement en attendant que le composant paresseux se charge.

```
import React, { Suspense } from 'react';

const LazyComponent = React.lazy(() => import('./LazyComponent'));

function MyComponent() {
  return (
  <div>
    <Suspense fallback={<div>Loading....</div>}>
      <LazyComponent />
    </Suspense>
  </div>
 );
}
```

Utilisez React.memo pour la mémorisation des composants

React.memo est un excellent moyen d'optimiser les performances car il aide à mettre en cache les composants fonctionnels.

Voici comment cela fonctionne : lorsqu'une fonction est rendue à l'aide de cette technique, elle enregistre le résultat en mémoire, et la prochaine fois que la fonction avec les mêmes arguments est appelée, elle renvoie le résultat enregistré sans exécuter à nouveau la fonction, ce qui vous permet d'économiser de la bande passante.

Dans le contexte de React, les **fonctions** sont les **composants fonctionnels** et les **arguments** sont les **props** . Voici un exemple :

```
import React from 'react';


const MyComponent = React.memo(props =>  {
  /* render only if the props changed */
});
```

React.memo est un composant d'ordre supérieur et il est similaire
à **React.PureComponent** mais pour utiliser des composants de fonction au lieu de classes

Virtualiser une grande liste à l'aide de la fenêtre de réaction

Lorsque vous souhaitez afficher une énorme table ou une liste de données, cela peut considérablement ralentir les performances de votre application. La virtualisation peut aider dans un scénario comme celui-ci à l'aide d'une bibliothèque telle que react-window . react-window aide à résoudre ce problème en ne restituant que les éléments de la liste qui sont actuellement visibles, ce qui permet de restituer efficacement des listes de toute taille.

```
import React from 'react';
import { FixedSizeList as List } from 'react-window';
import './style.css';

const Row = ({ index, style }) => (
  <div className={index % 2 ? 'ListItemOdd' : 'ListItemEven'} style={style
    Row {index}
  </div>
);
const Example = () => (
  <List
    className="List"
    height={150}
    itemCount={1000}
    itemSize={35}
    width={300}
  >
    {Row}
  </List>
);
```

Emballer

Les techniques décrites ci-dessus sont toutes d'excellents moyens pour vous de mettre en pratique l'optimisation des performances de votre application. S'il y a une chose que vous retenez de cet article, c'est que vous devriez toujours faire un effort pour créer une application performante qui améliorera considérablement la vitesse de votre application et l'expérience de votre utilisateur en créant d'abord le projet, puis en optimisant les performances où nécessaire; en faisant cela, vous n'avez qu'une longueur d'avance pour rendre vos utilisateurs heureux. Merci d'avoir lu et j'espère que vous avez trouvé cet