eBook Gratuit

APPRENEZ React

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#reactjs

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: react

It is an unofficial and free React ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official React.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec React

Remarques

React est une bibliothèque JavaScript déclarative basée sur des composants, utilisée pour créer des interfaces utilisateur.

Pour atteindre les fonctionnalités du framework MVC dans React, les développeurs l'utilisent en conjonction avec une saveur Flux de votre choix, par exemple Redux .

Versions

Version	Date de sortie
0.3.0	2013-05-29
0.4.0	2013-07-17
0.5.0	2013-10-16
0.8.0	2013-12-19
0.9.0	2014-02-20
0.10.0	2014-03-21
0.11.0	2014-07-17
0.12.0	2014-10-28
0.13.0	2015-03-10
0,14,0	2015-10-07
15.0.0	2016-04-07
15.1.0	2016-05-20
15.2.0	2016-07-01
15.2.1	2016-07-08
15.3.0	2016-07-29
15.3.1	2016-08-19
15.3.2	2016-09-19

Version	Date de sortie
15.4.0	2016-11-16
15.4.1	2016-11-23
15.4.2	2017-01-06
15.5.0	2017-04-07
15.6.0	2017-06-13

Examples

Installation ou configuration

ReactJS est une bibliothèque JavaScript contenue dans un seul fichier `react-<version>.js` pouvant être inclus dans n'importe quelle page HTML. Les gens installent également généralement la bibliothèque React DOM `react-dom-<version>.js` avec le fichier principal React:

Inclusion de base

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script type="text/javascript">
        // Use react JavaScript code here or in a separate file
    </script>
    </body>
</html>
```

Pour obtenir les fichiers JavaScript, accédez à la page d'installation de la documentation officielle de React.

React prend également en charge la syntaxe JSX . JSX est une extension créée par Facebook qui ajoute une syntaxe XML à JavaScript. Pour utiliser JSX, vous devez inclure la bibliothèque Babel et changer `<script type="text/javascript">` en `<script type="text/babel">` afin de traduire JSX en code Javascript.

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
    <script type="text/babel">
        // Use react JSX code here or in a separate file
    </script>
```

```
    </body>
</html>
```

Installation via npm

Vous pouvez également installer React en utilisant npm en procédant comme suit:

```
npm install --save react react-dom
```

Pour utiliser React dans votre projet JavaScript, vous pouvez effectuer les opérations suivantes:

```
var React = require('react');
var ReactDOM = require('react-dom');
ReactDOM.render(<App />, ...);
```

Installation via fil

Facebook a publié son propre gestionnaire de paquets nommé Yarn , qui peut également être utilisé pour installer React. Après avoir installé Yarn, il vous suffit d'exécuter cette commande:

```
yarn add react react-dom
```

Vous pouvez ensuite utiliser React dans votre projet exactement comme si vous aviez installé React via npm.

Composant Hello World

Un composant React peut être défini comme une classe ES6 qui étend la classe `React.Component` base. Sous sa forme minimale, un composant *doit* définir une méthode de `render` spécifiant le rendu du composant dans le DOM. La méthode de `render` renvoie les noeuds React, qui peuvent être définis à l'aide de la syntaxe JSX en tant que balises de type HTML. L'exemple suivant montre comment définir un composant minimal:

```
import React from 'react'

class HelloWorld extends React.Component {
    render() {
        return <h1>Hello, World!</h1>
    }
}

export default HelloWorld
```

Un composant peut également recevoir des `props` . Ce sont des propriétés passées par son parent afin de spécifier certaines valeurs que le composant ne peut pas connaître par lui-même; une propriété peut également contenir une fonction pouvant être appelée par le composant après que certains événements se soient produits - par exemple, un bouton peut recevoir une fonction pour sa propriété `onClick` et l'appeler chaque fois que l'utilisateur clique dessus. Lors de l'écriture d'un composant, ses `props` sont accessibles via l'objet `props` sur le composant lui-même:

```
import React from 'react'

class Hello extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}!</h1>
    }
}

export default Hello
```

L'exemple ci-dessus montre comment le composant peut rendre une chaîne arbitraire transmise au `name` prop par son parent. Notez qu'un composant ne peut pas modifier les accessoires qu'il reçoit.

Un composant peut être rendu dans n'importe quel autre composant, ou directement dans le DOM s'il s'agit du composant le plus haut, en utilisant `ReactDOM.render` et en lui fournissant à la fois le composant et le noeud DOM que vous souhaitez rendre:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))
```

Vous savez maintenant comment créer un composant de base et accepter des `props` . Permet d'aller plus loin et d'introduire l' `state` .

Pour des raisons de démonstration, créons notre application Hello World, affichez uniquement le prénom si un nom complet est donné.

```
import React from 'react'

class Hello extends React.Component {

    constructor(props){

        //Since we are extending the default constructor,
        //handle default activities first.
        super(props);

        //Extract the first-name from the prop
        let firstName = this.props.name.split(" ")[0];

        //In the constructor, feel free to modify the
        //state property on the current context.
        this.state = {
            name: firstName
        }

    } //Look maa, no comma required in JSX based class defs!

    render() {
        return <h1>Hello, {this.state.name}!</h1>
    }
}
```

```
export default Hello
```

Remarque: Chaque composant peut avoir son propre état ou accepter son état parent comme prop.

Codepen Lien vers l'exemple.

Bonjour le monde

Sans JSX

Voici un exemple de base qui utilise l'API principale de React pour créer un élément React et l'API React DOM pour rendre l'élément React dans le navigateur.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>

  </head>
  <body>
    <div id="example"></div>

    <script type="text/javascript">

      // create a React element rElement
      var rElement = React.createElement('h1', null, 'Hello, world!');

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Avec JSX

Au lieu de créer un élément React à partir de chaînes, on peut utiliser JSX (une extension Javascript créée par Facebook pour ajouter de la syntaxe XML à JavaScript), ce qui permet d'écrire

```
var rElement = React.createElement('h1', null, 'Hello, world!');
```

comme équivalent (et plus facile à lire pour une personne familiarisée avec HTML)

```
var rElement = <h1>Hello, world!</h1>;
```

Le code contenant JSX doit être placé dans une `<script type="text/babel">` . Tout ce qui se trouve dans cette balise sera transformé en Javascript en utilisant la bibliothèque Babel (qui doit être ajoutée aux bibliothèques React).

Donc, finalement, l'exemple ci-dessus devient:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- Include the Babel library -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>

  </head>
  <body>
   <div id="example"></div>

    <script type="text/babel">

      // create a React element rElement using JSX
      var rElement = <h1>Hello, world!</h1>;

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Qu'est-ce que ReactJS?

ReactJS est une bibliothèque frontale open source, basée sur des composants, responsable uniquement de la **couche d'affichage** de l'application. Il est maintenu par Facebook.

ReactJS utilise un mécanisme virtuel basé sur DOM pour remplir des données (vues) dans HTML DOM. Le DOM virtuel fonctionne rapidement grâce au fait qu'il ne modifie que les éléments DOM individuels au lieu de recharger le DOM complet à chaque fois

> Une application React est composée de plusieurs **composants** , chacun responsable de la sortie d'un petit élément HTML réutilisable. Les composants peuvent être imbriqués dans d'autres composants pour permettre la création d'applications complexes à partir de blocs de construction simples. Un composant peut également conserver son état interne - par exemple, un composant TabList peut stocker une

variable correspondant à l'onglet actuellement ouvert.

React nous permet d'écrire des composants en utilisant un langage spécifique au domaine appelé JSX. JSX nous permet d'écrire nos composants à l'aide de HTML, tout en mélangeant des événements JavaScript. React le convertira en interne en un DOM virtuel et produira finalement notre code HTML pour nous.

React " *réagit* " pour indiquer rapidement et automatiquement les modifications apportées à vos composants afin de les réinstaller dans le DOM HTML en utilisant le DOM virtuel. Le DOM virtuel est une représentation en mémoire d'un DOM réel. En effectuant l'essentiel du traitement dans le DOM virtuel plutôt que directement dans le DOM du navigateur, React peut agir rapidement et n'ajouter, mettre à jour et supprimer que les composants modifiés depuis le dernier cycle de rendu.

Bonjour tout le monde avec des fonctions sans état

Les composants sans état obtiennent leur philosophie de la programmation fonctionnelle. Ce qui implique que: Une fonction renvoie tout le temps exactement la même chose à ce qui lui est donné.

Par exemple:

```
const statelessSum = (a, b) => a + b;

let a = 0;
const statefulSum = () => a++;
```

Comme vous pouvez le voir dans l'exemple ci-dessus, statelessSum retournera toujours les mêmes valeurs données a et b. Cependant, la fonction statefulSum ne renverra pas les mêmes valeurs, même sans paramètres. Ce type de comportement est également appelé *effet secondaire* . Depuis, le composant affecte quelque chose au-delà.

Ainsi, il est conseillé d'utiliser plus souvent des composants sans état, car ils sont *sans effet secondaire* et créent toujours le même comportement. C'est ce que vous voulez être dans vos applications parce que l'état fluctuant est le pire des scénarios pour un programme maintenable.

Le type de composant de réaction le plus fondamental est celui sans état. Les composants réactifs qui sont des fonctions pures de leurs accessoires et ne nécessitent aucune gestion d'état interne peuvent être écrits en tant que fonctions JavaScript simples. Ceux-ci sont considérés comme des `Stateless Functional Components` car ils ne sont fonction que des `props` , sans aucun `state` à suivre.

Voici un exemple simple pour illustrer le concept d'un `Stateless Functional Component` :

```
// In HTML
<div id="element"></div>

// In React
```

```
const MyComponent = props => {
    return <h1>Hello, {props.name}!</h1>;
};

ReactDOM.render(<MyComponent name="Arun" />, element);
// Will render <h1>Hello, Arun!</h1>
```

Notez que tout ce que fait ce composant rend un élément `h1` contenant le `name` prop. Ce composant ne suit aucun état. Voici également un exemple ES6:

```
import React from 'react'

const HelloWorld = props => (
    <h1>Hello, {props.name}!</h1>
)

HelloWorld.propTypes = {
    name: React.PropTypes.string.isRequired
}

export default HelloWorld
```

Étant donné que ces composants ne nécessitent pas d'instance de sauvegarde pour gérer l'état, React dispose de davantage d'espace pour les optimisations. L'implémentation est propre, mais jusqu'à présent, aucune optimisation de ce type n'a été implémentée .

Créer une application Réagir

create-react-app est un générateur de réactions créé par Facebook. Il fournit un environnement de développement configuré pour une facilité d'utilisation avec une configuration minimale, notamment:

- Transpilation ES6 et JSX
- Serveur de développement avec rechargement de module à chaud
- Linting code
- Préfixe CSS
- Créer un script avec JS, CSS et regroupement d'images, et des sourcemaps
- Cadre de test Jest

Installation

Tout d'abord, installez create-react-app globalement avec le gestionnaire de packages (npm).

```
npm install -g create-react-app
```

Ensuite, lancez le générateur dans le répertoire choisi.

```
create-react-app my-app
```

Accédez au répertoire nouvellement créé et exécutez le script de démarrage.

```
cd my-app/
npm start
```

Configuration

create-react-app est intentionnellement non configurable par défaut. Si une utilisation autre que celle par défaut est requise, par exemple pour utiliser un langage CSS compilé tel que Sass, la commande d'éjection peut être utilisée.

```
npm run eject
```

Cela permet de modifier tous les fichiers de configuration. NB c'est un processus irréversible.

Des alternatives

Les modèles de remplacement React incluent:

- enclave
- nwb
- mouvement
- rackt-cli
- budō
- rwb
- quik
- sagui
- roc

Construire React App

Pour créer votre application pour la production prête, exécutez la commande suivante

```
npm run build
```

Bases absolues de la création de composants réutilisables

Composants et accessoires

Comme React ne concerne que le point de vue d'une application, l'essentiel du développement dans React sera la création de composants. Un composant représente une partie de la vue de votre application. "Props" sont simplement les attributs utilisés sur un nœud JSX (par exemple, `<SomeComponent someProp="some prop's value" />`), et sont la principale manière dont notre application interagit avec nos composants. Dans l'extrait ci-dessus, à l'intérieur de SomeComponent, nous aurions accès à `this.props` , dont la valeur serait l'objet `{someProp: "some prop's value"}` .

Il peut être utile de considérer les composants de React comme des fonctions simples: ils prennent en compte les «accessoires» et produisent une sortie sous forme de balisage. Beaucoup de composants simples vont plus loin en se faisant des "fonctions pures", ce qui signifie qu'ils ne génèrent pas d'effets secondaires et sont idempotents (étant donné un ensemble d'entrées, le composant produira toujours la même sortie). Cet objectif peut être formellement appliqué en créant des composants en tant que fonctions, plutôt que des "classes". Il existe trois manières de créer un composant React:

- **Composants fonctionnels ("sans état")**

```
const FirstComponent = props => (
    <div>{props.content}</div>
);
```

- **React.createClass ()**

```
const SecondComponent = React.createClass({
    render: function () {
        return (
            <div>{this.props.content}</div>
        );
    }
});
```

- **Classes ES2015**

```
class ThirdComponent extends React.Component {
    render() {
        return (
            <div>{this.props.content}</div>
        );
    }
}
```

Ces composants sont utilisés de la même manière:

```
const ParentComponent = function (props) {
    const someText = "FooBar";
    return (
        <FirstComponent content={someText} />
        <SecondComponent content={someText} />
        <ThirdComponent content={someText} />
    );
}
```

Les exemples ci-dessus produiront tous un balisage identique.

Les composants fonctionnels ne peuvent pas avoir "d'état" en eux. Donc, si votre composant doit avoir un état, optez pour des composants basés sur des classes. Reportez-vous à la section Création de composants pour plus d'informations.

En guise de dernière remarque, les accessoires de réaction sont immuables une fois qu'ils ont été

passés, ce qui signifie qu'ils ne peuvent pas être modifiés à partir d'un composant. Si le parent d'un composant modifie la valeur d'un accessoire, React gère le remplacement des anciens accessoires par le nouveau, le composant se répète en utilisant les nouvelles valeurs.

Reportez-vous à la section Réaction et réutilisation des composants pour des plongées plus approfondies dans la relation entre les accessoires et les composants.

Lire Commencer avec React en ligne: https://riptutorial.com/fr/reactjs/topic/797/commencer-avec-react

Chapitre 2: Clés en réaction

Introduction

Les clés en réaction sont utilisées pour identifier une liste d'éléments DOM de la même hiérarchie en interne.

Donc, si vous parcourez un tableau pour afficher une liste d'éléments li, chacun des éléments li nécessite un identifiant unique spécifié par la propriété key. Cela peut généralement être l'ID de votre élément de base de données ou l'index du tableau.

Remarques

L'utilisation de l'index de tableau en tant que clé n'est généralement pas recommandée lorsque le tableau doit changer au fil du temps. À partir des documents réactifs:

> En dernier recours, vous pouvez passer l'index de l'élément dans le tableau en tant que clé. Cela peut bien fonctionner si les articles ne sont jamais réorganisés, mais les commandes seront lentes.

Un bon exemple à ce sujet: https://medium.com/@robinpokorny/index-as-a-key-is-ananti-pattern-e0349aece318

Examples

Utiliser l'id d'un élément

Nous avons ici une liste de tâches à transmettre aux accessoires de notre composant.

Chaque élément todo a une propriété text et id. Imaginez que la propriété id provienne d'un datastore de backend et soit une valeur numérique unique:

```
todos = [
  {
    id: 1,
    text: 'value 1'
  },
  {
    id: 2,
    text: 'value 2'
  },
  {
    id: 3,
    text: 'value 3'
  },
  {
    id: 4,
    text: 'value 4'
  },
```

```
];
```

Nous définissons l'attribut clé de chaque élément de liste itéré sur `todo-${todo.id}` pour que réaction puisse l'identifier en interne:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo) =>
        <li key={ `todo-${todo.id}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Utilisation de l'index du tableau

Si vous ne disposez pas d'identifiants de base de données uniques, vous pouvez également utiliser l'index numérique de votre tableau comme suit:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo, index) =>
        <li key={ `todo-${index}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Lire Clés en réaction en ligne: https://riptutorial.com/fr/reactjs/topic/9805/cles-en-reaction

Chapitre 3: Comment configurer un environnement Web de base, réagir et babel

Remarques

Ce pipeline de construction n'est pas exactement ce que vous qualifieriez de «prêt à la production», mais il vous permet de commencer à ajouter les éléments dont vous avez besoin pour obtenir l'expérience de développement que vous recherchez. L'approche que certaines personnes adoptent (y compris moi-même parfois) est de prendre un pipeline entièrement construit de Yeoman.io ou quelque part ailleurs, puis de retirer les choses qu'ils ne veulent pas jusqu'à ce que cela leur convienne. Il n'y a rien de mal à cela, mais peut-être avec l'exemple ci-dessus, vous pouvez opter pour l'approche opposée et construire à partir de rien.

Certaines choses que vous aimeriez ajouter sont des choses comme un cadre de test et des statistiques de couverture comme Karma avec Moka ou Jasmine. Linting avec ESLint. Hot module de remplacement dans webpack-dev-server afin que vous puissiez obtenir cette expérience de développement Ctrl + S, F5. De plus, le pipeline actuel ne construit qu'en mode dev, donc une tâche de construction de production serait bien.

Gotchas!

Remarquez que dans la propriété context du `webpack.config.js` nous avons utilisé le module de chemin de noeud pour définir notre chemin plutôt que de simplement concaténer `__dirname` avec la chaîne `/src` car Windows déteste les slashs avant . Donc, pour rendre la solution plus multi-plate-forme compatible utiliser le noeud de levier pour nous aider.

Explication des propriétés webpack.config.js

le contexte

Il s'agit du chemin de fichier que webpack utilisera comme chemin racine pour résoudre les chemins de fichiers relatifs. Donc, dans index.jsx où nous utilisons `require('./index.html')` ce point est résolu dans le répertoire `src/` car nous l'avons défini comme tel dans cette propriété.

entrée

Où webpack cherche d'abord à regrouper la solution. C'est pourquoi vous verrez que dans index.jsx, nous assemblons la solution avec les exigences et les importations.

sortie

C'est là que nous définissons où webpack devrait déposer les fichiers qu'il a trouvés. Nous avons également défini un nom pour le fichier dans lequel nos javascript et styles fournis seront supprimés.

devServer

Ce sont des paramètres spécifiques à webpack-dev-server. `contentBase` définit où le serveur doit créer sa racine, nous avons défini le dossier `dist/` comme base ici. Le `port` est le port sur lequel le serveur sera hébergé. `open` est ce qui est utilisé pour demander à webpack-dev-server d'ouvrir votre navigateur par défaut une fois qu'il est mis en place sur le serveur.

module> chargeurs

Ceci définit un mappage à utiliser pour Webpack afin de savoir quoi faire quand il trouve des fichiers différents. La propriété `test` donne une expression rationnelle que webpack doit utiliser pour déterminer si elle doit appliquer ce module. Dans la plupart des cas, nous avons mis en correspondance les extensions de fichier. `loader` ou `loaders` donne le nom du module de chargement que nous aimerions utiliser pour charger le fichier dans webpack et laisser ce chargeur s'occuper du regroupement de ce type de fichier. Il y a aussi une propriété de `query` sur le javascript, cela fournit juste une chaîne de requête au chargeur, donc nous aurions probablement pu utiliser une propriété de requête sur le chargeur html si nous le voulions. C'est juste une façon différente de faire les choses.

Examples

Comment construire un pipeline pour un "Hello world" personnalisé avec des images.

Étape 1: Installez Node.js

Le pipeline de génération que vous allez créer est basé sur Node.js, vous devez donc vous assurer que vous l'avez installé en premier lieu. Pour obtenir des instructions sur l'installation de Node.js, vous pouvez consulter les documents SO pour cela ici.

Etape 2: Initialisez votre projet en tant que module de noeud

Ouvrez votre dossier de projet sur la ligne de commande et utilisez la commande suivante:

```
npm init
```

Pour les besoins de cet exemple, vous pouvez prendre les paramètres par défaut ou si vous souhaitez plus d'informations sur ce que tout cela signifie, vous pouvez consulter cette documentation sur la configuration du package.

Étape 3: Installer les paquetages npm nécessaires

Exécutez la commande suivante sur la ligne de commande pour installer les packages nécessaires pour cet exemple:

```
npm install --save react react-dom
```

Ensuite, pour les dépendances de développement, exécutez cette commande:

```
npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader
css-loader style-loader file-loader image-webpack-loader
```

Enfin, webpack et webpack-dev-server sont des éléments qui valent la peine d'être installés globalement plutôt que comme une dépendance de votre projet. Si vous préférez les ajouter en tant que dépendance, cela ne fonctionnera pas. Voici la commande à exécuter:

```
npm install --global webpack webpack-dev-server
```

Étape 3: Ajouter un fichier .babelrc à la racine de votre projet

Cela permettra à babel d'utiliser les préréglages que vous venez d'installer. Votre fichier .babelrc doit ressembler à ceci:

```
{
  "presets": ["react", "es2015"]
}
```

Étape 4: Configurer la structure du répertoire du projet

Configurez-vous une structure de répertoire qui ressemble à la suivante dans la racine de votre répertoire:

```
|- node_modules
|- src/
  |- components/
  |- images/
  |- styles/
  |- index.html
  |- index.jsx
|- .babelrc
|- package.json
```

REMARQUE: Les `node_modules` , `.babelrc` et `package.json` devraient tous être déjà présents depuis les étapes précédentes. Je les ai simplement inclus pour que vous puissiez voir où ils correspondent.

Étape 5: Remplissez le projet avec les fichiers de projet Hello World

Ce n'est pas vraiment important pour le processus de construction d'un pipeline, alors je vais simplement vous donner le code et vous pouvez les copier dans:

src / components / HelloWorldComponent.jsx

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

class HelloWorldComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: 'Student'};
    this.handleChange = this.handleChange.bind(this);
  }
```

```
  handleChange(e) {
    this.setState({name: e.target.value});
  }

  render() {
    return (
      <div>
        <div className="image-container">
          <img src="./images/myImage.gif" />
        </div>
        <div className="form">
          <input type="text" onChange={this.handleChange} />
          <div>
            My name is {this.state.name} and I'm a clever cloggs because I built a React build
pipeline
          </div>
        </div>
      </div>
    );
  }
}

export default HelloWorldComponent;
```

src / images / myImage.gif

N'hésitez pas à substituer ceci à n'importe quelle image que vous souhaitez, simplement pour prouver que nous pouvons regrouper des images. Si vous fournissez votre propre image et que vous la nommez autrement, vous devrez mettre à jour `HelloWorldComponent.jsx` pour refléter vos modifications. De même, si vous choisissez une image avec une extension de fichier différente, vous devez modifier la propriété de `test` du chargeur d'image dans le fichier webpack.config.js avec une expression régulière appropriée pour correspondre à votre nouvelle extension de fichier.

src / styles / styles.css

```
.form {
  margin: 25px;
  padding: 25px;
  border: 1px solid #ddd;
  background-color: #eaeaea;
  border-radius: 10px;
}

.form div {
  padding-top: 25px;
}

.image-container {
  display: flex;
  justify-content: center;
}
```

index.html

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Learning to build a react pipeline</title>
</head>
<body>
  <div id="content"></div>
  <script src="app.js"></script>
</body>
</html>
```

index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import HelloWorldComponent from './components/HelloWorldComponent.jsx';

require('./images/myImage.gif');
require('./styles/styles.css');
require('./index.html');

render(<HelloWorldComponent />, document.getElementById('content'));
```

Étape 6: Créer une configuration Webpack

Créez un fichier appelé webpack.config.js à la racine de votre projet et copiez ce code dans celui-ci:

webpack.config.js

```
var path = require('path');

var config = {
  context: path.resolve(__dirname + '/src'),
  entry: './index.jsx',
  output: {
    filename: 'app.js',
    path: path.resolve(__dirname + '/dist'),
  },
  devServer: {
    contentBase: path.join(__dirname + '/dist'),
    port: 3000,
    open: true,
  },
  module: {
    loaders: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      },
      {
        test: /\.css$/,
        loader: "style!css"
      },
      {
        test: /\.gif$/,
        loaders: [
```

```
          'file?name=[path][name].[ext]',
          'image-webpack',
        ]
      },
      { test: /\.(html)$/,
        loader: "file?name=[path][name].[ext]"
      }
    ],
  },
};

module.exports = config;
```

Étape 7: Créez des tâches npm pour votre pipeline

Pour ce faire, vous devez ajouter deux propriétés à la clé de script du JSON définie dans le fichier package.json à la racine de votre projet. Faites en sorte que votre clé de script ressemble à ceci:

```
  "scripts": {
    "start": "webpack-dev-server",
    "build": "webpack",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

Le script de `test` aura déjà été là et vous pouvez choisir de le conserver ou non, ce n'est pas important pour cet exemple.

Étape 8: utiliser le pipeline

Depuis la ligne de commande, si vous êtes dans le répertoire racine du projet, vous devriez maintenant pouvoir exécuter la commande:

```
npm run build
```

Cela regroupera la petite application que vous avez créée et la placera dans le répertoire `dist/` qu'il créera à la racine du dossier de votre projet.

Si vous exécutez la commande:

```
npm start
```

Ensuite, l'application que vous avez créée sera servie dans votre navigateur Web par défaut à l'intérieur d'une instance de serveur de développement Web.

Lire Comment configurer un environnement Web de base, réagir et babel en ligne: https://riptutorial.com/fr/reactjs/topic/6294/comment-configurer-un-environnement-web-de-base--reagir-et-babel

Chapitre 4: Comment et pourquoi utiliser les clés dans React

Introduction

Chaque fois que vous restituez une liste de composants React, chaque composant doit avoir un attribut de `key` . La clé peut avoir n'importe quelle valeur, mais elle doit être unique pour cette liste.

Lorsque React doit effectuer des modifications sur une liste d'éléments, React réitère les deux listes d'enfants en même temps et génère une mutation chaque fois qu'il y a une différence. S'il n'y a pas de clés pour les enfants, React analyse chaque enfant. Sinon, React compare les clés pour savoir lesquelles ont été ajoutées ou supprimées de la liste.

Remarques

Pour plus d'informations, visitez ce lien pour savoir comment utiliser les clés:
https://facebook.github.io/react/docs/lists-and-keys.html

Et visitez ce lien pour savoir pourquoi il est recommandé d'utiliser des clés:
https://facebook.github.io/react/docs/reconciliation.html#recursing-on-children

Examples

Exemple de base

Pour un composant React sans classe:

```
function SomeComponent(props){

    const ITEMS = ['cat', 'dog', 'rat']
    function getItemsList(){
        return ITEMS.map(item => <li key={item}>{item}</i>);
    }

    return (
        <ul>
            {getItemsList()}
        </ul>
    );
}
```

Pour cet exemple, le composant ci-dessus résout:

```
<ul>
    <li key='cat'>cat</li>
    <li key='dog'>dog</li>
    <li key='rat'>rat</li>
```

```
<ul>
```

Lire Comment et pourquoi utiliser les clés dans React en ligne:
https://riptutorial.com/fr/reactjs/topic/9665/comment-et-pourquoi-utiliser-les-cles-dans-react

Chapitre 5: Communication entre composants

Remarques

Il y a un total de 3 cas de communication entre les composants React:

- Cas 1: Communication de parent à enfant
- Cas 2: communication enfant à parent
- Cas 3: Composants non liés (tout composant à un composant) communication

Examples

Composants parent à enfant

Que le cas le plus simple en réalité, très naturel dans le monde de la réaction et les chances sont - vous l'utilisez déjà.

Vous pouvez **transmettre des accessoires aux composants enfants** . Dans cet exemple, le `message` que nous transmettons au composant enfant, le nom du message est choisi arbitrairement, vous pouvez le nommer comme vous voulez.

```
import React from 'react';

class Parent extends React.Component {
    render() {
        const variable = 5;
        return (
            <div>
                <Child message="message for child" />
                <Child message={variable} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return <h1>{this.props.message}</h1>
    }
}

export default Parent;
```

Ici, le composant `<Parent />` affiche deux composants `<Child />` , en transmettant un `message for child` à l'intérieur du premier composant et `5` le second.

En résumé, vous avez un composant (parent) qui en rend un autre (enfant) et lui transmet des accessoires.

Composants enfant à parent

En renvoyant des données au parent, pour ce faire, nous **transmettons** simplement **une fonction en tant que prop du composant parent au composant enfant** et **le composant enfant appelle cette fonction** .

Dans cet exemple, nous allons modifier l'état Parent en transmettant une fonction au composant Child et en appelant cette fonction dans le composant Child.

```
import React from 'react';

class Parent extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };

        this.outputEvent = this.outputEvent.bind(this);
    }
    outputEvent(event) {
        // the event context comes from the Child
        this.setState({ count: this.state.count++ });
    }

    render() {
        const variable = 5;
        return (
            <div>
                Count: { this.state.count }
                <Child clickHandler={this.outputEvent} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return (
            <button onClick={this.props.clickHandler}>
                Add One More
            </button>
        );
    }
}

export default Parent;
```

Notez que la méthode `outputEvent` du `outputEvent` (qui modifie l'état parent) est appelée par l'événement `onClick` du bouton `onClick` .

Composants non liés

La seule façon si vos composants n'ont pas de relation parent-enfant (ou sont liés mais trop éloignés, comme un grand-petit-fils) est d'avoir un signal auquel un composant est abonné, et l'autre écrit.

Ce sont les 2 opérations de base de tout système d'événement: **abonnez-vous / écoutez** un événement à notifier et **envoyez / déclenchez / publiez / envoyez** un événement pour avertir ceux qui le souhaitent.

Il y a au moins 3 modèles pour faire cela. Vous pouvez trouver une comparaison ici .

Voici un bref résumé:

- Modèle 1: **Emetteur / cible / répartiteur d'événement** : les auditeurs doivent référencer la source pour s'abonner.
    - pour vous abonner: `otherObject.addEventListener('click', () => { alert('click!'); });`
    - à expédier: `this.dispatchEvent('click');`
- Pattern 2: **Publish / Subscribe** : vous n'avez pas besoin d'une référence spécifique à la source qui déclenche l'événement, il y a un objet global accessible partout qui gère tous les événements.
    - s'abonner: `globalBroadcaster.subscribe('click', () => { alert('click!'); });`
    - à envoyer: `globalBroadcaster.publish('click');`
- Pattern 3: **Signals** : similaire à Event Emitter / Target / Dispatcher mais vous n'utilisez aucune chaîne aléatoire ici. Chaque objet pouvant émettre des événements doit avoir une propriété spécifique portant ce nom. De cette façon, vous savez exactement quels événements un objet peut émettre.
    - pour vous abonner: `otherObject.clicked.add( () => { alert('click'); });`
    - à envoyer: `this.clicked.dispatch();`

Lire Communication entre composants en ligne:
https://riptutorial.com/fr/reactjs/topic/6567/communication-entre-composants

Chapitre 6: Communiquer entre les composants

Examples

Communication entre composants fonctionnels sans état

Dans cet exemple, nous utiliserons les modules `Redux` et `React Redux` pour gérer l'état de notre application et pour le rendu automatique de nos composants fonctionnels. Et ofcourse `React` et `React Dom`

Vous pouvez commander la démo terminée ici

Dans l'exemple ci-dessous, nous avons trois composants différents et un composant connecté

- **UserInputForm** : ce composant affiche un champ de saisie Et lorsque la valeur du champ change, il appelle la méthode `inputChange` sur les `props` (fournie par le composant parent) et, si les données sont également fournies, elle les affiche dans le champ de saisie.
- **UserDashboard:** Ce composant affiche un message simple et niche également `UserInputForm` composant, il passe également `inputChange` méthode pour `UserInputForm` composant, `UserInputForm` composant permet inturn utilisation de cette méthode pour communiquer avec le composant parent.
    - **UserDashboardConnected** : Ce composant **encapsule** simplement le composant `UserDashboard` aide de la méthode de `ReactRedux connect` . Cela simplifie la gestion de l'état du composant et la mise à jour du composant lorsque l'état change.
- **App** : Ce composant `UserDashboardConnected` simplement le composant `UserDashboardConnected` .

```
const UserInputForm = (props) => {

  let handleSubmit = (e) => {
    e.preventDefault();
  }

  return(
    <form action="" onSubmit={handleSubmit}>
      <label htmlFor="name">Please enter your name</label>
      <br />
      <input type="text" id="name" defaultValue={props.data.name || ''} onChange={
props.inputChange } />
    </form>
  )

}


const UserDashboard = (props) => {
```

```
  let inputChangeHandler = (event) => {
    props.updateName(event.target.value);
  }

  return(
    <div>
      <h1>Hi { props.user.name || 'User' }</h1>
      <UserInputForm data={props.user} inputChange={inputChangeHandler} />
    </div>
  )

}

const mapStateToProps = (state) => {
  return {
    user: state
  };
}
const mapDispatchToProps = (dispatch) => {
  return {
    updateName: (data) => dispatch( Action.updateName(data) ),
  };
};

const { connect, Provider } = ReactRedux;
const UserDashboardConnected = connect(
  mapStateToProps,
  mapDispatchToProps
)(UserDashboard);



const App = (props) => {
  return(
    <div>
      <h1>Communication between Stateless Functional Components</h1>
      <UserDashboardConnected />
    </div>
  )
}



const user = (state={name: 'John'}, action) => {
  switch (action.type) {
    case 'UPDATE_NAME':
      return Object.assign( {}, state, {name: action.payload}  );

    default:
      return state;
  }
};

const { createStore } = Redux;
const store = createStore(user);
const Action = {
  updateName: (data) => {
    return { type : 'UPDATE_NAME', payload: data }
  },
}
```

```
ReactDOM.render(
  <Provider store={ store }>
    <App />
  </Provider>,
  document.getElementById('application')
);
```

URL de la corbeille JS

Lire Communiquer entre les composants en ligne:
https://riptutorial.com/fr/reactjs/topic/6137/communiquer-entre-les-composants

Chapitre 7: Composants

Remarques

`React.createClass` est `React.createClass` obsolète dans la version 15.5 et devrait être supprimé dans la version 16 . Il existe un package de remplacement pour ceux qui en ont encore besoin. Les exemples qui l'utilisent doivent être mis à jour.

Examples

Composant de base

Vu le fichier HTML suivant:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Vous pouvez créer un composant de base en utilisant le code suivant dans un fichier distinct:

scripts / example.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class FirstComponent extends Component {
  render() {
    return (
      <div className="firstComponent">
        Hello, world! I am a FirstComponent.
      </div>
    );
  }
}
ReactDOM.render(
  <FirstComponent />, // Note that this is the same as the variable you stored above
  document.getElementById('content')
);
```

Vous obtiendrez le résultat suivant (notez le contenu de `div#content`):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content">
        <div className="firstComponent">
            Hello, world! I am a FirstComponent.
        </div>
    </div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Composants d'imbrication

Une grande partie de la puissance de ReactJS réside dans sa capacité à permettre l’imbrication de composants. Prenez les deux composants suivants:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = reactCreateClass({
  render: function() {
    return (
      <div className="commentForm">
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

Vous pouvez imbriquer et faire référence à ces composants dans la définition d'un composant différent:

```
var React = require('react');
var createReactClass = require('create-react-class');
```

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList /> // Which was defined above and can be reused
        <CommentForm /> // Same here
      </div>
    );
  }
});
```

La nidification supplémentaire peut être effectuée de trois manières différentes, qui ont toutes leur propre emplacement.

1. Nidification sans enfants

(suite de ci-dessus)

```
var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        <ListTitle/>
        Hello, world! I am a CommentList.
      </div>
    );
  }
});
```

C'est le style où A compose B et B compose C.

Avantages

- Facile et rapide pour séparer les éléments de l'interface utilisateur
- Facile à injecter des accessoires aux enfants en fonction de l'état du composant parent

Les inconvénients

- Moins de visibilité sur l'architecture de composition
- Moins de réutilisation

Bon si

- B et C ne sont que des composants de présentation
- B devrait être responsable du cycle de vie de C

2. Nidification à l'aide d'enfants

(suite de ci-dessus)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList>
            <ListTitle/> // child
        </CommentList>
        <CommentForm />
      </div>
    );
  }
});
```

C'est le style où A compose B et A indique à B de composer C. Plus de puissance aux composants parents.

Avantages

- Meilleure gestion du cycle de vie des composants
- Meilleure visibilité dans l'architecture de composition
- Meilleure réutilisabilité

Les inconvénients

- L'injection d'accessoires peut devenir un peu cher
- Moins de flexibilité et de puissance dans les composants enfants

Bon si

- B devrait accepter de composer quelque chose de différent de C dans le futur ou ailleurs
- A devrait contrôler le cycle de vie de C

B rendrait C en utilisant `this.props.children` , et B ne `this.props.children` aucune manière structurée de savoir à quoi servent ces enfants. Ainsi, B peut enrichir les composants enfants en donnant des accessoires supplémentaires, mais si B doit savoir exactement ce qu'ils sont, le n ° 3 pourrait être une meilleure option.

3. Nidification à l'aide d'accessoires

(suite de ci-dessus)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList title={ListTitle}/> //prop
        <CommentForm />
      </div>
    );
  }
});
```

C'est le style où A compose B et B fournit une option permettant à A de passer quelque chose à composer dans un but spécifique. Composition plus structurée.

Avantages

- Composition en tant que caractéristique
- Validation facile
- Meilleure composition

Les inconvénients

- L'injection d'accessoires peut devenir un peu cher
- Moins de flexibilité et de puissance dans les composants enfants

Bon si

- B a des fonctionnalités spécifiques définies pour composer quelque chose
- B ne devrait savoir que rendre que ne pas rendre

3 est généralement un must pour créer une bibliothèque publique de composants mais aussi une bonne pratique pour créer des composants composables et définir clairement les caractéristiques de la composition. # 1 est le plus simple et le plus rapide pour faire quelque chose qui fonctionne, mais # 2 et # 3 devraient offrir certains avantages dans divers cas d'utilisation.

Création de composants

Ceci est une extension de l'exemple de base:

Structure basique

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class FirstComponent extends Component {
    render() {
        return (
```

```
            <div>
                Hello, {this.props.name}! I am a FirstComponent.
            </div>
        );
    }
}

render(
    <FirstComponent name={ 'User' } />,
    document.getElementById('content')
);
```

L'exemple ci-dessus est appelé un composant **sans état** car il ne contient pas d' état (au sens réactif du mot).

Dans un tel cas, certaines personnes trouvent préférable d'utiliser des composants fonctionnels sans état, basés sur des fonctions fléchées ES6 .

Composants fonctionnels sans état

Dans de nombreuses applications, il existe des composants intelligents contenant des états mais rendant des composants stupides qui reçoivent simplement des objets et renvoient du code HTML au format JSX. Les composants fonctionnels sans état sont beaucoup plus réutilisables et ont un impact positif sur les performances de votre application.

Ils ont 2 caractéristiques principales:

1. Lorsqu'ils sont rendus, ils reçoivent un objet avec tous les accessoires qui ont été transmis
2. Ils doivent retourner le JSX à rendre

```
// When using JSX inside a module you must import React
import React from 'react';
import PropTypes from 'prop-types';

const FirstComponent = props => (
    <div>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

//arrow components also may have props validation
FirstComponent.propTypes = {
    name: PropTypes.string.isRequired,
}

// To use FirstComponent in another file it must be exposed through an export call:
export default FirstComponent;
```

Composants avec état

Contrairement aux composants "sans état" présentés ci-dessus, les composants "avec état" ont

un objet d'état qui peut être mis à jour avec la méthode `setState` . L'état doit être initialisé dans le `constructor` avant de pouvoir être défini:

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

class SecondComponent extends Component {
    constructor(props) {
        super(props);

        this.state = {
            toggle: true
        };

        // This is to bind context when passing onClick as a callback
        this.onClick = this.onClick.bind(this);
    }

    onClick() {
        this.setState((prevState, props) => ({
            toggle: !prevState.toggle
        }));
    }

    render() {
        return (
            <div onClick={this.onClick}>
                Hello, {this.props.name}! I am a SecondComponent.
                <br />
                Toggle is: {this.state.toggle}
            </div>
        );
    }
}
```

L'extension d'un composant avec `PureComponent` au lieu de `Component` implémentera automatiquement la méthode du cycle de vie `shouldComponentUpdate()` avec une comparaison peu profonde des propriétés et des états. Cela permet à votre application d'être plus performante en réduisant la quantité de rendus inutiles. Cela suppose que vos composants sont «purs» et rendent toujours la même sortie avec le même état et la même entrée de propriétés.

Composants d'ordre supérieur

Les composants d'ordre supérieur (HOC) permettent de partager les fonctionnalités des composants.

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

const PrintHello = ComposedComponent => class extends Component {
    onClick() {
        console.log('hello');
    }

    /* The higher order component takes another component as a parameter
    and then renders it with additional props */
    render() {
```

```
        return <ComposedComponent {...this.props } onClick={this.onClick} />
    }
}

const FirstComponent = props => (
    <div onClick={ props.onClick }>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

const ExtendedComponent = PrintHello(FirstComponent);
```

Les composants d'ordre supérieur sont utilisés lorsque vous souhaitez partager la logique entre plusieurs composants, quelle que soit la différence de rendu.

setState pièges

Vous devez faire preuve de prudence lorsque vous utilisez `setState` dans un contexte asynchrone. Par exemple, vous pouvez essayer d'appeler `setState` dans le rappel d'une requête get:

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

        this.state = {
            user: {}
        };
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });
    }

    render() {
        return <h1>{this.state.user}</h1>;
    }
}
```

Cela pourrait appeler des problèmes - si le rappel est appelé après que le `Component` est démonté, alors `this.setState` ne sera pas une fonction. Chaque fois que c'est le cas, vous devez veiller à ce que votre utilisation de `setState` soit `setState` .

Dans cet exemple, vous pouvez annuler la demande XHR lorsque le composant est démonté:

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

        this.state = {
```

```
            user: {},
            xhr: null
        };
    }

    componentWillUnmount() {
        let xhr = this.state.xhr;

        // Cancel the xhr request, so the callback is never called
        if (xhr && xhr.readyState != 4) {
            xhr.abort();
        }
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        let xhr = $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });

        this.setState({xhr: xhr});
    }
}
```

La méthode asynchrone est enregistrée en tant qu'état. Dans `componentWillUnmount` vous effectuez tout votre nettoyage, y compris l'annulation de la demande XHR.

Vous pourriez aussi faire quelque chose de plus complexe. Dans cet exemple, je crée une fonction 'stateSetter' qui accepte cet objet comme argument et empêche `this.setState` lorsque la fonction `cancel` a été appelée:

```
function stateSetter(context) {
    var cancelled = false;
    return {
        cancel: function () {
            cancelled = true;
        },
        setState(newState) {
            if (!cancelled) {
                context.setState(newState);
            }
        }
    }
}

class Component extends React.Component {
    constructor(props) {
        super(props);
        this.setter = stateSetter(this);
        this.state = {
            user: 'loading'
        };
    }
    componentWillUnmount() {
        this.setter.cancel();
```

```
    }
    componentDidMount() {
        this.fetchUser();
    }
    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setter.setState({user: user});
            });
    }
    render() {
        return <h1>{this.state.user}</h1>
    }
}
```

Cela fonctionne car la variable `cancelled` est visible dans la fermeture de `setState` nous avons créée.

Les accessoires

Les accessoires sont un moyen de transmettre des informations dans un composant React, ils peuvent avoir n'importe quel type, y compris des fonctions - parfois appelés rappels.

Dans JSX, les objets sont transmis avec la syntaxe d'attribut

```
<MyComponent userID={123} />
```

Dans la définition de MyComponent, userID sera désormais accessible depuis l'objet props

```
// The render function inside MyComponent
render() {
    return (
        <span>The user's ID is {this.props.userID}</span>
    )
}
```

Il est important de définir tous les `props` , leurs types et, le cas échéant, leur valeur par défaut:

```
// defined at the bottom of MyComponent
MyComponent.propTypes = {
    someObject: React.PropTypes.object,
    userID: React.PropTypes.number.isRequired,
    title: React.PropTypes.string
};

MyComponent.defaultProps = {
    someObject: {},
    title: 'My Default Title'
}
```

Dans cet exemple, le prop `someObject` est facultatif, mais l' `userID` prop est requis. Si vous ne parvenez pas à fournir `userID` à `MyComponent` , à l'exécution, le moteur React affiche une console vous avertissant que le prop requis n'a pas été fourni. Attention, cet avertissement ne s'affiche que

dans la version de développement de la bibliothèque React, la version de production ne enregistrera aucun avertissement.

Utiliser `defaultProps` vous permet de simplifier

```
const { title = 'My Default Title' } = this.props;
console.log(title);
```

à

```
console.log(this.props.title);
```

C'est aussi une garantie pour l'utilisation du `array object` et des `functions` . Si vous ne fournissez pas de support par défaut pour un objet, les éléments suivants génèrent une erreur si le prop n'est pas transmis:

```
if (this.props.someObject.someKey)
```

Dans l'exemple ci-dessus, `this.props.someObject` n'est `undefined` et par conséquent, la vérification de `someKey` une erreur et le code sera rompu. Avec l'utilisation de `defaultProps` vous pouvez utiliser en toute sécurité la vérification ci-dessus.

Etats des composants - Interface utilisateur dynamique

Supposons que nous voulions avoir le comportement suivant - Nous avons un en-tête (par exemple un élément h3) et en cliquant dessus, nous voulons qu'il devienne une zone de saisie pour pouvoir modifier le nom de l'en-tête. React rend cela très simple et intuitif en utilisant les états des composants et si des instructions sont utilisées. (Explication du code ci-dessous)

```
// I have used ReactBootstrap elements. But the code works with regular html elements also
var Button = ReactBootstrap.Button;
var Form = ReactBootstrap.Form;
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;

var Comment = reactCreateClass({
  getInitialState: function() {
    return {show: false, newTitle: ''};
  },

  handleTitleSubmit: function() {
     //code to handle input box submit - for example, issue an ajax request to change name in
database
  },

  handleTitleChange: function(e) {
    //code to change the name in form input box. newTitle is initialized as empty string. We
need to update it with the string currently entered by user in the form
    this.setState({newTitle: e.target.value});
  },

  changeComponent: function() {
```

```
    // this toggles the show variable which is used  for dynamic UI
    this.setState({show: !this.state.show)};
  },

  render: function() {

    var clickableTitle;

    if(this.state.show) {
        clickableTitle = <Form inline onSubmit={this.handleTitleSubmit}>
                            <FormGroup controlId="formInlineTitle">
                                <FormControl type="text" onChange={this.handleTitleChange}>
                            </FormGroup>
                        </Form>;
    } else {
        clickabletitle = <div>
                           <Button bsStyle="link" onClick={this.changeComponent}>
                               <h3> Default Text </h3>
                           </Button>
                        </div>;
    }

    return (
        <div className="comment">
            {clickableTitle}
        </div>
    );
  }
});

ReactDOM.render(
    <Comment />, document.getElementById('content')
);
```

La partie principale du code est la variable **clickableTitle** . Basé sur la variable d'état **show** , il peut s'agir d'un élément Form ou d'un élément Button. React permet l'imbrication de composants.

Nous pouvons donc ajouter un élément {clickableTitle} dans la fonction de rendu. Il recherche la variable clickableTitle. Basé sur la valeur 'this.state.show', il affiche l'élément correspondant.

Variations de composants fonctionnels sans état

```
const languages = [
  'JavaScript',
  'Python',
  'Java',
  'Elm',
  'TypeScript',
  'C#',
  'F#'
]
```

```
// one liner
const Language = ({language}) => <li>{language}</li>

Language.propTypes = {
```

```
  message: React.PropTypes.string.isRequired
}
```

```
/**
* If there are more than one line.
* Please notice that round brackets are optional here,
* However it's better to use them for readability
*/
const LanguagesList = ({languages}) => {
  <ul>
    {languages.map(language => <Language language={language} />)}
  </ul>
}

LanguagesList.PropTypes = {
  languages: React.PropTypes.array.isRequired
}
```

```
/**
 * This syntax is used if there are more work beside just JSX presentation
 * For instance some data manipulations needs to be done.
 * Please notice that round brackets after return are required,
 * Otherwise return will return nothing (undefined)
 */
const LanguageSection = ({header, languages}) => {
  // do some work
  const formattedLanguages = languages.map(language => language.toUpperCase())
  return (
    <fieldset>
      <legend>{header}</legend>
      <LanguagesList languages={formattedLanguages} />
    </fieldset>
  )
}

LanguageSection.PropTypes = {
  header: React.PropTypes.string.isRequired,
  languages: React.PropTypes.array.isRequired
}
```

```
ReactDOM.render(
  <LanguageSection
    header="Languages"
    languages={languages} />,
  document.getElementById('app')
)
```

Ici vous pouvez trouver exemple de travail de celui - ci.

Lire Composants en ligne: https://riptutorial.com/fr/reactjs/topic/1185/composants

Chapitre 8: Composants d'ordre supérieur

Introduction

Les composants d'ordre supérieur ("HOC" en bref) est un modèle de conception d'application de réaction utilisé pour améliorer les composants avec un code réutilisable. Ils permettent d'ajouter des fonctionnalités et des comportements aux classes de composants existantes.

Un HOC est une fonction javascript pure qui accepte un composant comme argument et renvoie un nouveau composant avec la fonctionnalité étendue.

Remarques

Les HOC sont assez souvent utilisés dans des bibliothèques tierces. Comme la fonction de connexion Redux.

Examples

Composant simple d'ordre supérieur

Disons que nous voulons console.log chaque fois que le composant est monté:

hocLogger.js

```
export default function hocLogger(Component) {
  return class extends React.Component {
    componentDidMount() {
      console.log('Hey, we are mounted!');
    }
    render() {
      return <Component {...this.props} />;
    }
  }
}
```

Utilisez ce HOC dans votre code:

MyLoggedComponent.js

```
import React from "react";
import {hocLogger} from "./hocLogger";

export class MyLoggedComponent extends React.Component {
    render() {
        return (
            <div>
                This component get's logged to console on each mount.
            </div>
        );
```

```
    }
}

// Now wrap MyLoggedComponent with the hocLogger function
export default hocLogger(MyLoggedComponent);
```

Composant d'ordre supérieur qui vérifie l'authentification

Disons que nous avons un composant qui ne devrait être affiché que si l'utilisateur est connecté.

Nous créons donc un HOC qui vérifie l'authentification sur chaque rendu ():

AuthenticatedComponent.js

```
import React from "react";

export function requireAuthentication(Component) {
    return class AuthenticatedComponent extends React.Component {

        /**
         * Check if the user is authenticated, this.props.isAuthenticated
         * has to be set from your application logic (or use react-redux to retrieve it from
global state).
         */
        isAuthenticated() {
            return this.props.isAuthenticated;
        }

        /**
         * Render
         */
        render() {
            const loginErrorMessage = (
                <div>
                    Please <a href="/login">login</a> in order to view this part of the
application.
                </div>
            );

            return (
                <div>
                    { this.isAuthenticated === true ? <Component {...this.props} /> :
loginErrorMessage }
                </div>
            );
        }
    };
}

export default requireAuthentication;
```

Nous utilisons ensuite ce composant d'ordre supérieur dans nos composants qui doit être caché aux utilisateurs anonymes:

MyPrivateComponent.js

```
import React from "react";
import {requireAuthentication} from "./AuthenticatedComponent";

export class MyPrivateComponent extends React.Component {
    /**
     * Render
     */
    render() {
        return (
            <div>
                My secret search, that is only viewable by authenticated users.
            </div>
        );
    }
}

// Now wrap MyPrivateComponent with the requireAuthentication function
export default requireAuthentication(MyPrivateComponent);
```

Cet exemple est décrit plus en détail ici .

Lire Composants d'ordre supérieur en ligne:
https://riptutorial.com/fr/reactjs/topic/9819/composants-d-ordre-superieur

Chapitre 9: Composants fonctionnels sans état

Remarques

Les composants fonctionnels sans état de React sont des fonctions pures des `props` passés. Ces composants ne dépendent pas de l'état et rejettent l'utilisation des méthodes de cycle de vie des composants. Vous pouvez cependant définir les `propTypes` et les `defaultPropts` .

Voir https://facebook.github.io/react/docs/reusable-components.html#stateless-functions pour plus d'informations sur les composants fonctionnels sans état.

Examples

Composant fonctionnel sans état

Les composants vous permettent de diviser l'interface utilisateur en éléments *indépendants* et *réutilisables* . C'est la beauté de React; nous pouvons séparer une page en plusieurs petits **composants** réutilisables.

Avant React v14, nous pouvions créer un composant React avec `React.Component` aide de `React.Component` (dans ES6) ou `React.createClass` (dans ES5), qu'il nécessite ou non un état pour gérer les données.

React v14 a introduit un moyen plus simple de définir des composants, généralement appelés **composants fonctionnels sans état** . Ces composants utilisent des fonctions JavaScript simples.

Par exemple:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Cette fonction est un composant React valide car elle accepte un seul argument d'objet `props` avec des données et renvoie un élément React. Nous appelons ces composants **fonctionnels** car ils sont littéralement des *fonctions* JavaScript.

Les composants fonctionnels sans état se concentrent généralement sur l'interface utilisateur; L'état doit être géré par des composants «conteneur» de niveau supérieur, ou via Flux / Redux, etc. Les composants fonctionnels sans état ne prennent pas en charge les méthodes d'état ou de cycle de vie.

Avantages:

1. Pas de surcharge de classe
2. Vous n'avez pas à vous soucier de `this` mot clé
3. Facile à écrire et facile à comprendre
4. Vous n'avez pas à vous soucier de la gestion des valeurs d'état
5. Amélioration des performances

Résumé : Si vous écrivez un composant React qui ne nécessite pas d'état et que vous souhaitez créer une interface utilisateur réutilisable, au lieu de créer un composant React standard, vous pouvez l'écrire en tant que **composant fonctionnel sans état** .

Prenons un exemple simple:

Disons que nous avons une page qui peut enregistrer un utilisateur, rechercher des utilisateurs enregistrés ou afficher une liste de tous les utilisateurs enregistrés.

C'est le point d'entrée de l'application, `index.js` :

```
import React from 'react';
import ReactDOM from 'react-dom';

import HomePage from './homepage'

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

Le composant `HomePage` fournit l'interface utilisateur pour vous enregistrer et rechercher des utilisateurs. Notez qu'il s'agit d'un composant React typique, y compris l'état, l'interface utilisateur et le code comportemental. Les données pour la liste des utilisateurs enregistrés sont stockées dans la variable d' `state` , mais notre `List` réutilisable (illustrée ci-dessous) encapsule le code de l'interface utilisateur pour la liste.

`homepage.js` :

```
import React from 'react'
import {Component} from 'react';

import List from './list';

export default class Temp extends Component{

    constructor(props) {
        super();
        this.state={users:[], showSearchResult: false, searchResult: []};
    }

    registerClick(){
        let users = this.state.users.slice();
        if(users.indexOf(this.refs.mail_id.value) == -1){
            users.push(this.refs.mail_id.value);
            this.refs.mail_id.value = '';
            this.setState({users});
        }else{
```

```
            alert('user already registered');
        }
    }

    searchClick(){
        let users = this.state.users;
        let index = users.indexOf(this.refs.search.value);
        if(index >= 0){
            this.setState({searchResult: users[index], showSearchResult: true});
        }else{
            alert('no user found with this mail id');
        }
    }

    hideSearchResult(){
        this.setState({showSearchResult: false});
    }

    render() {
        return (
            <div>
                <input placeholder='email-id' ref='mail_id'/>
                <input type='submit' value='Click here to register'
onClick={this.registerClick.bind(this)}/>
                <input style={{marginLeft: '100px'}} placeholder='search' ref='search'/>
                <input type='submit' value='Click here to register'
onClick={this.searchClick.bind(this)}/>
                {this.state.showSearchResult ?
                    <div>
                        Search Result:
                        <List users={[this.state.searchResult]}/>
                        <p onClick={this.hideSearchResult.bind(this)}>Close this</p>
                    </div>
                    :
                    <div>
                        Registered users:
                        <br/>
                        {this.state.users.length ?
                            <List users={this.state.users}/>
                            :
                            "no user is registered"
                        }
                    </div>
                }
            </div>
        );
    }
}
```

Enfin, notre `List` **composants fonctionnels sans état** , utilisée pour afficher à la fois la liste des utilisateurs enregistrés *et* les résultats de la recherche, mais sans conserver aucun état.

`list.js` :

```
import React from 'react';
var colors = ['#6A1B9A', '#76FF03', '#4527A0'];

var List = (props) => {
    return(
```

```
        <div>
            {
                props.users.map((user, i)=>{
                    return(
                        <div key={i} style={{color: colors[i%3]}}>
                            {user}
                        </div>
                    );
                })
            }
        </div>
    );
}

export default List;
```

Référence: https://facebook.github.io/react/docs/components-and-props.html

Lire Composants fonctionnels sans état en ligne:
https://riptutorial.com/fr/reactjs/topic/6588/composants-fonctionnels-sans-etat

Chapitre 10: Configuration de l'environnement réactif

Examples

Composant Simple React

Nous voulons pouvoir compiler le composant ci-dessous et le rendre dans notre page Web

Nom de fichier : src / index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';

class ToDo extends React.Component {
    render() {
        return (<div>I am working</div>);
    }
}

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

Installer toutes les dépendances

```
# install react and react-dom
$ npm i react react-dom --save

# install webpack for bundling
$ npm i webpack -g

# install babel for module loading, bundling and transpiling
$ npm i babel-core babel-loader --save

# install babel presets for react and es6
$ npm i babel-preset-react babel-preset-es2015 --save
```

Configurer le webpack

Créez un fichier `webpack.config.js` à la racine de votre répertoire de travail

Nom de fichier : webpack.config.js

```
module.exports = {
    entry: __dirname + "/src/index.jsx",
    devtool: "source-map",
    output: {
        path: __dirname + "/build",
        filename: "bundle.js"
    },
```

```
    module: {
        loaders: [
            {test: /\.jsx?$/, exclude: /node_modules/, loader: "babel-loader"}
        ]
    }
}
```

Configurer babel

Créez un fichier `.babelrc` à la racine de notre répertoire de travail

Nom de fichier : .babelrc

```
{
    "presets": ["es2015","react"]
}
```

Fichier HTML à utiliser pour réagir

Configurer un simple fichier html à la racine du répertoire du projet

Nom de fichier : index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="App"></div>
    <script src="build/bundle.js" charset="utf-8"></script>
  </body>
</html>
```

Transpile et regroupe ton composant

En utilisant Webpack, vous pouvez regrouper votre composant:

```
$ webpack
```

Cela va créer notre fichier de sortie dans le répertoire de `build` .

Ouvrez la page HTML dans un navigateur pour voir le composant en action

Lire Configuration de l'environnement réactif en ligne:
https://riptutorial.com/fr/reactjs/topic/7480/configuration-de-l-environnement-reactif

Chapitre 11: Cycle de vie des composants réactifs

Introduction

Les méthodes de cycle de vie doivent être utilisées pour exécuter du code et interagir avec votre composant à différents moments de la vie des composants. Ces méthodes sont basées sur un composant Montage, Mise à jour et Démontage.

Examples

Création de composants

Lorsqu'un composant React est créé, plusieurs fonctions sont appelées:

- Si vous utilisez `React.createClass` (ES5), 5 fonctions définies par l'utilisateur sont appelées
- Si vous utilisez la `class Component extends React.Component` (ES6), 3 fonctions définies par l'utilisateur sont appelées

`getDefaultProps()` (ES5 uniquement)

C'est la **première** méthode appelée.

Les valeurs de prop renvoyées par cette fonction seront utilisées par défaut si elles ne sont pas définies lors de l'instanciation du composant.

Dans l'exemple suivant, `this.props.name` sera `this.props.name` par défaut sur `Bob` s'il n'est pas spécifié autrement:

```
getDefaultProps() {
  return {
    initialCount: 0,
    name: 'Bob'
  };
}
```

`getInitialState()` (ES5 uniquement)

C'est la **deuxième** méthode appelée.

La valeur de retour de `getInitialState()` définit l'état initial du composant React. Le framework React appelle cette fonction et attribue la valeur de retour à `this.state` .

Dans l'exemple suivant, `this.state.count` sera initialisé avec la valeur de `this.props.initialCount` :

```
getInitialState() {
  return {
    count : this.props.initialCount
  };
}
```

`componentWillMount()` (ES5 et ES6)

C'est la **troisième** méthode appelée.

Cette fonction peut être utilisée pour apporter des modifications finales au composant avant qu'il ne soit ajouté au DOM.

```
componentWillMount() {
  ...
}
```

`render()` (ES5 et ES6)

C'est la **quatrième** méthode appelée.

La fonction `render()` devrait être une fonction pure de l'état et des propriétés du composant. Il retourne un seul élément qui représente le composant pendant le processus de rendu et doit soit représenter un composant DOM natif (par exemple, `<p />`), soit un composant composite. Si rien ne doit être rendu, il peut renvoyer `null` ou `undefined` .

Cette fonction sera rappelée après toute modification des accessoires ou de l'état du composant.

```
render() {
  return (
    <div>
      Hello, {this.props.name}!
    </div>
  );
}
```

`componentDidMount()` (ES5 et ES6)

C'est la **cinquième** méthode appelée.

Le composant a été monté et vous pouvez désormais accéder aux noeuds DOM du composant, par exemple via les `refs` .

Cette méthode doit être utilisée pour:

- Préparer des minuteries
- Récupération des données
- Ajouter des écouteurs d'événement
- Manipulation des éléments DOM

```
componentDidMount() {
  ...
}
```

ES6 Syntaxe

Si le composant est défini à l'aide de la syntaxe de classe ES6, les fonctions `getDefaultProps()` et `getInitialState()` ne peuvent pas être utilisées.

Au lieu de cela, nous déclarons nos `defaultProps` comme une propriété statique sur la classe et déclarons la forme d'état et l'état initial dans le constructeur de notre classe. Celles-ci sont toutes deux définies sur l'instance de la classe au moment de la construction, avant toute autre fonction de cycle de vie React.

L'exemple suivant illustre cette approche alternative:

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

    this.state = {
      count: this.props.initialCount
    };
  }

  upCount() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
        Hello, {this.props.name}!<br />
        You clicked the button {this.state.count} times.<br />
        <button onClick={this.upCount}>Click here!</button>
      </div>
    );
  }
}

MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Remplacement de `getDefaultProps()`

Les valeurs par défaut des composants du composant sont spécifiées en définissant la propriété `defaultProps` de la classe:

```
MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Remplacement de `getInitialState()`

Le moyen idiomatique de configurer l'état initial du composant est de définir `this.state` dans le constructeur:

```
constructor(props){
  super(props);

  this.state = {
    count: this.props.initialCount
  };
}
```

Mise à jour des composants

`componentWillReceiveProps(nextProps)`

C'est la **première fonction appelée sur les modifications de propriétés** .

Lorsque **les propriétés du composant changent** , React appelle cette fonction avec les **nouvelles propriétés** . Vous pouvez accéder aux anciens accessoires avec *this.props* et aux nouveaux accessoires avec *nextProps* .

Avec ces variables, vous pouvez effectuer des opérations de comparaison entre les anciens et les nouveaux accessoires, ou la fonction d'appel, car une propriété change, etc.

```
componentWillReceiveProps(nextProps){
  if (nextProps.initialCount && nextProps.initialCount > this.state.count){
    this.setState({
      count : nextProps.initialCount
    });
  }
}
```

`shouldComponentUpdate(nextProps, nextState)`

C'est la **deuxième fonction appelée sur les modifications de propriétés et la première sur les changements d'état** .

Par défaut, si un autre composant / votre composant modifie une propriété / un état de votre composant, **React** affichera une nouvelle version de votre composant. Dans ce cas, cette fonction renvoie toujours true.

Vous pouvez remplacer cette fonction et **choisir plus précisément si votre composant doit être mis à jour ou non** .

Cette fonction est principalement utilisée pour l' **optimisation** .

Si la fonction renvoie **false** , le **pipeline de mise à jour s'arrête immédiatement** .

```
componentShouldUpdate(nextProps, nextState){
  return this.props.name !== nextProps.name ||
    this.state.count !== nextState.count;
}
```

componentWillUpdate(nextProps, nextState)

Cette fonction fonctionne comme `componentWillMount()` . **Les modifications ne sont pas dans DOM** , vous pouvez donc apporter certaines modifications juste avant la mise à jour.

/! \: vous ne pouvez pas utiliser **this.setState ()** .

```
componentWillUpdate(nextProps, nextState){}
```

render()

Il y a quelques modifications, alors restaurez le composant.

componentDidUpdate(prevProps, prevState)

Même chose que `componentDidMount()` : **DOM est rafraîchi** , vous pouvez donc travailler sur le DOM ici.

```
componentDidUpdate(prevProps, prevState){}
```

Enlèvement de composant

componentWillUnmount()

Cette méthode est appelée **avant** qu'un composant ne soit démonté du DOM.

C'est un bon endroit pour effectuer des opérations de nettoyage comme:

- Suppression des écouteurs d'événements
- Effacer les minuteries.
- Arrêt des prises
- Nettoyage des états de redux.

```
componentWillUnmount(){
  ...
}
```

Exemple de suppression d'un écouteur d'événement attaché dans `componentWillUnMount`

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

export default class SideMenu extends Component {

  constructor(props) {
    super(props);
    this.state = {
        ...
      };
    this.openMenu = this.openMenu.bind(this);
    this.closeMenu = this.closeMenu.bind(this);
  }

  componentDidMount() {
    document.addEventListener("click", this.closeMenu);
  }

  componentWillUnmount() {
    document.removeEventListener("click", this.closeMenu);
  }

  openMenu() {
    ...
  }

  closeMenu() {
    ...
  }

  render() {
    return (
      <div>
        <a
          href      = "javascript:void(0)"
          className = "closebtn"
          onClick   = {this.closeMenu}
        >
          x
        </a>
        <div>
          Some other structure
        </div>
      </div>
    );
  }
}
```

Réaction du conteneur de composants

Lors de la création d'une application React, il est souvent souhaitable de diviser les composants en fonction de leur responsabilité principale, en composants de présentation et de conteneur. Les composants de présentation ne concernent que l'affichage des données - ils peuvent être

considérés comme des fonctions qui convertissent souvent un modèle en vue. En règle générale, ils ne maintiennent aucun état interne. Les composants de conteneur concernent la gestion des données. Cela peut être fait en interne par leur propre état, ou en agissant comme intermédiaires avec une bibliothèque de gestion d'état telle que Redux. Le composant conteneur n'affichera pas directement les données, mais les transmettra à un composant de présentation.

```
// Container component
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
    constructor() {
        super();
        // Set initial state
        this.state = { comments: [] }
    }

    componentDidMount() {
        // Make API call and update state with returned comments
        Api.getComments().then(comments => this.setState({ comments }));
    }

    render() {
        // Pass our state comments to the presentational component
        return (
            <CommentsList comments={this.state.comments} />;
        );
    }
}

// Presentational Component
const CommentsList = ({ comments }) => (
    <div>
        {comments.map(comment => (
            <div>{comment}</div>
        )}
    </div>
);

CommentsList.propTypes = {
    comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```

Appel de méthode de cycle de vie dans différents états

Cet exemple sert de complément à d'autres exemples qui expliquent comment utiliser les méthodes du cycle de vie et quand la méthode sera appelée.

Cet exemple résume quelles méthodes (componentWillMount, componentWillReceiveProps, etc.) seront appelées et dans quelle séquence sera différent pour un composant **dans différents états** :

Lorsqu'un composant est initialisé:

1. getDefaultProps
2. getInitialState

3. componentWillMount
4. rendre
5. componentDidMount

Lorsqu'un état a changé d'état:

1. shouldComponentUpdate
2. componentWillUpdate
3. rendre
4. componentDidUpdate

Quand un composant a des accessoires changés:

1. componentWillReceiveProps
2. shouldComponentUpdate
3. componentWillUpdate
4. rendre
5. componentDidUpdate

Lorsqu'un composant se démonte:

1. componentWillUnmount

Lire Cycle de vie des composants réactifs en ligne:
https://riptutorial.com/fr/reactjs/topic/2750/cycle-de-vie-des-composants-reactifs

Chapitre 12: Etat en réaction

Examples

État de base

L'état dans les composants React est essentiel pour gérer et communiquer les données dans votre application. Il est représenté comme un objet JavaScript et a une portée au *niveau des composants* , il peut être considéré comme les données privées de votre composant.

Dans l'exemple ci-dessous, nous définissons un état initial dans la fonction `constructor` de notre composant et l'utilisons dans la fonction de `render` .

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

    // Set-up our initial state
    this.state = {
      greeting: 'Hiya Buddy!'
    };
  }

  render() {
    // We can access the greeting property through this.state
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

setState ()

La principale méthode de mise à jour de l'interface utilisateur pour vos applications React consiste à appeler la fonction `setState()` . Cette fonction effectuera une *fusion superficielle* entre le nouvel état que vous fournissez et l'état précédent, et déclenchera un re-rendu de votre composant et de tous les décédés.

Paramètres

1. `updater` : il peut s'agir d'un objet avec un nombre de paires clé-valeur à fusionner dans l'état ou une fonction renvoyant un tel objet.
2. `callback (optional)` : une fonction qui sera exécutée après l'exécution de `setState()` . En raison du fait que React ne garantit pas que les appels à `setState()` sont atomiques, cela peut parfois être utile si vous souhaitez effectuer une action après avoir été `setState()` que `setState()` a été exécuté avec succès.

Usage:

Le `setState` procédé accepte une `updater` à `prevState props updater` argument qui peut être soit un objet avec un nombre de clé-valeur paires qui doit être fusionné dans l'état, ou une fonction qui renvoie un tel objet calculé à partir de `prevState` et `props` .

Utiliser `setState()` avec un objet comme `updater` de `updater` à `updater`

```
//
// An example ES6 style component, updating the state on a simple button click.
// Also demonstrates where the state can be set directly and where setState should be used.
//
class Greeting extends React.Component {
    constructor(props) {
        super(props);
        this.click = this.click.bind(this);
        // Set initial state (ONLY ALLOWED IN CONSTRUCTOR)
        this.state = {
            greeting: 'Hello!'
        };
    }
    click(e) {
        this.setState({
              greeting: 'Hello World!'
        });
    }
    render() {
        return(
            <div>
                <p>{this.state.greeting}</p>
                <button onClick={this.click}>Click me</button>
            </div>
        );
    }

}
```

Utiliser `setState()` avec une fonction comme `updater` de `updater` à `updater`

```
//
// This is most often used when you want to check or make use
// of previous state before updating any values.
//

this.setState(function(previousState, currentProps) {
  return {
    counter: previousState.counter + 1
  };
});
```

Cela peut être plus sûr que d'utiliser un argument d'objet dans lequel plusieurs appels à `setState()` sont utilisés, car plusieurs appels peuvent être regroupés par React et exécutés en même temps.

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

Ces appels peuvent être regroupés par React en utilisant `Object.assign()` , ce qui fait que le compteur est incrémenté de 1 plutôt que de 3.

L'approche fonctionnelle peut également être utilisée pour déplacer la logique de réglage d'état en dehors des composants. Cela permet d'isoler et de réutiliser la logique d'état.

```
// Outside of component class, potentially in another file/module

function incrementCounter(previousState, currentProps) {
    return {
        counter: previousState.counter + 1
    };
}

// Within component

this.setState(incrementCounter);
```

Appel de `setState()` avec un objet et une fonction de rappel

```
//
// 'Hi There' will be logged to the console after setState completes
//

this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

Antipattern commun

Vous ne devriez pas enregistrer les `props` dans l' `state` . C'est considéré comme un anti-modèle . Par exemple:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            url: ''
        }

        this.onChange = this.onChange.bind(this);
    }
```

```
    onChange(e) {
        this.setState({
            url: this.props.url + '/days=?' + e.target.value
        });
    }

    componentWillMount() {
        this.setState({url: this.props.url});
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.state.url}
            </div>
        )
    }
}
```

L' `url` prop est enregistrée dans l' `state` , puis modifiée. Au lieu de cela, choisissez d'enregistrer les modifications dans un état, puis créez le chemin d'accès complet en utilisant à la fois l' `state` et les `props` :

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            days: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            days: e.target.value
        });
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.props.url + '/days?=' + this.state.days}
            </div>
        )
    }
}
```

En effet, dans une application React, nous voulons avoir une seule source de vérité - c’est-à-dire que toutes les données sont la responsabilité d’un seul composant et d’un seul composant. Il est de la responsabilité de ce composant de stocker les données dans son état et de distribuer les données à d'autres composants via des accessoires.

Dans le premier exemple, la classe MyComponent et son parent conservent "url" dans leur état. Si nous mettons à jour state.url dans MyComponent, ces modifications ne sont pas reflétées dans le parent. Nous avons perdu notre source unique de vérité et il devient de plus en plus difficile de suivre le flux de données grâce à notre application. En comparaison avec le deuxième exemple - l'url est uniquement maintenu dans l'état du composant parent et utilisé comme un prop dans MyComponent - nous conservons donc une source unique de vérité.

État, événements et contrôles gérés

Voici un exemple d'un composant React avec un champ de saisie "géré". Chaque fois que la valeur du champ d'entrée change, un gestionnaire d'événements est appelé et met à jour l'état du composant avec la nouvelle valeur du champ d'entrée. L'appel à `setState` dans le gestionnaire d'événements déclenchera un appel pour `render` mise à jour du composant dans le dom.

```
import React from 'react';
import {render} from 'react-dom';


class ManagedControlDemo extends React.Component {

  constructor(props){
    super(props);
    this.state = {message: ""};
  }

  handleChange(e){
    this.setState({message: e.target.value});
  }

  render() {
    return (
      <div>
        <legend>Type something here</legend>
          <input
            onChange={this.handleChange.bind(this)}
            value={this.state.message}
            autoFocus />
        <h1>{this.state.message}</h1>
      </div>
   );
  }
}


render(<ManagedControlDemo/>, document.querySelector('#app'));
```

Il est très important de noter le comportement à l'exécution. Chaque fois qu'un utilisateur modifie la valeur dans le champ de saisie

- `handleChange` sera appelé et ainsi
- `setState` sera appelé et ainsi
- `render` s'appellera

Pop quiz, après avoir tapé un caractère dans le champ de saisie, quels éléments DOM changent

1. tout cela - le div, la légende, l’entrée, le h1
2. seulement l'entrée et h1
3. rien
4. Qu'est-ce qu'un DOM?

Vous pouvez expérimenter cela plus ici pour trouver la réponse

Lire Etat en réaction en ligne: https://riptutorial.com/fr/reactjs/topic/1816/etat-en-reaction

Chapitre 13: Formulaires et saisie utilisateur

Examples

Composants contrôlés

Les composants de formulaire contrôlés sont définis avec une propriété `value` . La valeur des entrées contrôlées est gérée par React, les entrées utilisateur n'auront aucune influence directe sur l'entrée rendue. Au lieu de cela, une modification de la propriété de `value` doit refléter ce changement.

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

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: ''
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          value={this.state.name} />
      </div>
    )
  }
}
```

L'exemple ci - dessus montre comment la `value` propriété définit la valeur actuelle de l'entrée et le `onChange` gestionnaire d'événements met à jour l'état du composant avec l'entrée de l'utilisateur.

Les entrées de formulaire doivent être définies comme des composants contrôlés dans la mesure du possible. Cela garantit que l'état du composant et la valeur d'entrée sont synchronisés à tout moment, même si la valeur est modifiée par un déclencheur autre qu'une entrée utilisateur.

Composants non contrôlés

Les composants non contrôlés sont des entrées sans propriété de `value` . Contrairement aux composants contrôlés, il est de la responsabilité de l'application de maintenir l'état du composant

et la valeur d'entrée synchronisés.

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

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: 'John'
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          defaultValue={this.state.name} />
      </div>
    )
  }
}
```

Ici, l'état du composant est mis à jour via le `onChange` gestionnaire d'événements, tout comme pour les composants contrôlés. Toutefois, au lieu d'une propriété `value` , une propriété `defaultValue` est fournie. Cela détermine la valeur initiale de l'entrée lors du premier rendu. Toute modification ultérieure de l'état du composant n'est pas automatiquement reflétée par la valeur d'entrée. Si cela est nécessaire, un composant contrôlé doit être utilisé à la place.

Lire Formulaires et saisie utilisateur en ligne:
https://riptutorial.com/fr/reactjs/topic/2884/formulaires-et-saisie-utilisateur

Chapitre 14: Installation

Examples

Configuration simple

Configuration des dossiers

Cet exemple suppose que le code est dans `src/` et que la sortie doit être `out/` . En tant que telle, la structure des dossiers devrait ressembler à

```
example/
|-- src/
|   |-- index.js
|   `-- ...
|-- out/
`-- package.json
```

Mise en place des packages

En supposant un environnement npm d'installation, nous devons d'abord configurer babel afin de transposer le code React dans un code conforme à es5.

```
$npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

La commande ci-dessus indiquera à npm d'installer les bibliothèques principales de Babel ainsi que le module de chargement à utiliser avec webpack. Nous installons également les es6 et les presets de réaction pour que babel comprenne le code du module JSX et es6. (Plus d'informations sur les presets peuvent être trouvés ici Presets Babel)

```
$npm i -D webpack
```

Cette commande installera webpack en tant que dépendance de développement. (**je suis** le raccourci pour installer et **-D** le raccourci pour --save-dev)

Vous pouvez également installer d'autres packages Webpack (tels que des chargeurs supplémentaires ou l'extension webpack-dev-server).

Enfin, nous aurons besoin du code de réaction actuel

```
$npm i -D react react-dom
```

Mise en place du webpack

Avec la configuration des dépendances, nous aurons besoin d'un fichier webpack.config.js pour indiquer quoi faire à webpack

webpack.config.js simple:

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  }
};
```

Ce fichier demande à webpack de démarrer avec le fichier index.js (supposé être dans src /) et de le convertir en un seul fichier bundle.js dans le répertoire `out` .

Le bloc `module` indique à webpack de tester tous les fichiers rencontrés sur l'expression régulière et, s'ils correspondent, invoque le chargeur spécifié. (`babel-loader` dans ce cas) En outre, le regex l' `exclude` indique à webpack d'ignorer ce chargeur spécial pour tous les modules du dossier `node_modules` , ce qui accélère le processus de transpilation. Enfin, l'option `query` indique à webpack les paramètres à transmettre à babel et permet de transmettre les presets installés précédemment.

Tester la configuration

Il ne reste plus qu'à créer le fichier `src/index.js` et essayer de `src/index.js` l'application

src / index.js:

```
'use strict'

import React from 'react'
import { render } from 'react-dom'

const App = () => {
  return <h1>Hello world!</h1>
}

render(
```

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

Ce fichier devrait normalement afficher un simple en-tête `<h1>Hello world!</h1>` dans la balise html avec l'identifiant "app", mais pour le moment, cela devrait suffire à transposer le code une fois.

`$./node_modules/.bin/webpack` . Exécutera la version installée localement de webpack (utilisez `$webpack` si vous avez installé globalement Webpack avec -g)

Cela devrait créer le fichier `out/bundle.js` avec le code transpiled à l'intérieur et conclure l'exemple.

Utiliser webpack-dev-server

Installer

Après avoir configuré un projet simple pour utiliser webpack, babel et réagir en émettant `$npm i -g webpack-dev-server` installera le serveur http de développement pour un développement plus rapide.

Modification de webpack.config.js

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    publicPath: '/public/',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  },
  devServer: {
    contentBase: path.resolve(__dirname, 'public'),
    hot: true
  }
};
```

Les modifications sont dans

- `output.publicPath` qui configure un chemin d'accès `output.publicPath` lequel notre bundle doit être servi (voir les fichiers de configuration Webpack pour plus d'informations)
- `devServer`
    - `contentBase` le chemin de base pour servir les fichiers statiques (par exemple `index.html`)
    - `hot` définit le serveur webpack-dev pour recharger à chaud lorsque des modifications sont apportées aux fichiers sur le disque

Et enfin, nous avons juste besoin d'un index.html simple pour tester notre application.

index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>React Sandbox</title>
  </head>
  <body>

    <div id="app" />

    <script src="public/bundle.js"></script>
  </body>
</html>
```

Avec cette configuration exécutant `$webpack-dev-server` devrait démarrer un serveur http local sur le port 8080 et lors de la connexion devrait rendre une page contenant un `<h1>Hello world!</h1>` .

Lire Installation en ligne: https://riptutorial.com/fr/reactjs/topic/6441/installation

Chapitre 15: Installation React, Webpack & Typescript

Remarques

Pour obtenir une mise en évidence de la syntaxe dans votre éditeur (par exemple, le code VS), vous devez télécharger les informations de frappe pour les modules que vous utilisez dans votre projet.

Dites par exemple que vous utilisez React et ReactDOM dans votre projet, et que vous souhaitez mettre en évidence et Intellisense pour eux. Vous devrez ajouter les types à votre projet en utilisant cette commande:

```
npm install --save @types/react @types/react-dom
```

Votre éditeur devrait maintenant saisir automatiquement ces informations de saisie et vous fournir une saisie semi-automatique et Intellisense pour ces modules.

Examples

webpack.config.js

```
module.exports = {
    entry: './src/index',
    output: {
        path: __dirname + '/build',
        filename: 'bundle.js'
    },
    module: {
        rules: [{
            test: /\.tsx?$/,
            loader: 'ts-loader',
            exclude: /node_modules/
        }]
    },
    resolve: {
        extensions: ['.ts', '.tsx']
    }
};
```

Les composants principaux sont (en plus de l' `entry` standard, de la `output` et des autres propriétés du Webpack):

Le chargeur

Pour cela, vous devez créer une règle qui teste les extensions de fichier `.ts` et `.tsx` , spécifiez `ts-`

`loader` comme chargeur.

Résoudre les extensions TS

Vous devez également ajouter les extensions `.ts` et `.tsx` dans le tableau de `resolve` ou les webpack ne les verront pas.

tsconfig.json

Ceci est un tsconfig minimal pour vous permettre de démarrer.

```
{
    "include": [
        "src/*"
    ],
    "compilerOptions": {
        "target": "es5",
        "jsx": "react",
        "allowSyntheticDefaultImports": true
    }
}
```

Passons en revue les propriétés une par une:

`include`

Ceci est un tableau de code source. Ici, nous n'avons qu'une seule entrée, `src/*` , qui spécifie que tout ce qui se trouve dans le répertoire `src` doit être inclus dans la compilation.

`compilerOptions.target`

Spécifie que nous voulons compiler vers la cible ES5

`compilerOptions.jsx`

Si vous définissez ceci sur `true` , TypeScript compilera automatiquement votre syntaxe tsx de `<div />` à `React.createElement("div")` .

`compilerOptions.allowSyntheticDefaultImports`

Propriété pratique qui vous permettra d'importer des modules de nœuds comme s'ils étaient des modules ES6, donc au lieu de faire

```
import * as React from 'react'
const { Component } = React
```

tu peux juste faire

```
import React, { Component } from 'react'
```

sans aucune erreur vous indiquant que React n'a aucune exportation par défaut.

Mon premier composant

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


interface AppProps {
    name: string;
}
interface AppState {
    words: string[];
}

class App extends Component<AppProps, AppState> {
    constructor() {
        super();
        this.state = {
            words: ['foo', 'bar']
        };
    }

    render() {
        const { name } = this.props;
        return (<h1>Hello {name}!</h1>);
    }
}

const root = document.getElementById('root');
ReactDOM.render(<App name="Foo Bar" />, root);
```

Lorsque vous utilisez TypeScript with React, une fois que vous avez téléchargé les définitions de type React DefinitelyTyped (`npm install --save @types/react`), vous devez ajouter des annotations de type à chaque composant.

Vous faites cela comme ça:

```
class App extends Component<AppProps, AppState> { }
```

où `AppProps` et `AppState` sont des interfaces (ou des alias de type) pour les accessoires et l'état de vos composants, respectivement.

Lire Installation React, Webpack & Typescript en ligne:
https://riptutorial.com/fr/reactjs/topic/9590/installation-react--webpack--amp--typescript

Chapitre 16: Introduction au rendu côté serveur

Examples

Composants de rendu

Il existe deux options pour rendre les composants sur le serveur: `renderToString` et `renderToStaticMarkup` .

renderToString

Cela rendra les composants React au format HTML sur le serveur. Cette fonction ajoutera également `data-react-` propriétés de `data-react-` aux `data-react-` aux éléments HTML afin que React on client ne soit plus obligé de rendre les éléments.

```
import { renderToString } from "react-dom/server";
renderToString(<App />);
```

renderToStaticMarkup

Cela rendra les composants React au format HTML, mais sans `data-react-` propriétés de `data-react-` , il est déconseillé d'utiliser des composants qui seront rendus sur le client, car les composants seront réexécutés.

```
import { renderToStaticMarkup } from "react-dom/server";
renderToStaticMarkup(<App />);
```

Lire Introduction au rendu côté serveur en ligne:
https://riptutorial.com/fr/reactjs/topic/7478/introduction-au-rendu-cote-serveur

Chapitre 17: JSX

Remarques

JSX est une **étape de préprocesseur** qui ajoute la syntaxe XML à JavaScript. Vous pouvez certainement utiliser React sans JSX, mais JSX rend React beaucoup plus élégant.

Tout comme XML, les balises JSX ont un nom de balise, des attributs et des enfants. Si une valeur d'attribut est entre guillemets, la valeur est une chaîne. Sinon, placez la valeur entre accolades et la valeur est l'expression JavaScript incluse.

Fondamentalement, JSX ne fournit que du sucre syntaxique pour la fonction `React.createElement(component, props, ...children)` .

Donc, le code JSX suivant:

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

ReactDOM.render(<HelloMessage name="Kalo" />, mountNode);
```

Compile jusqu'au code JavaScript suivant:

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
}

ReactDOM.render(React.createElement(HelloMessage, { name: "Kalo" }), mountNode);
```

En conclusion, notez que **la ligne suivante dans JSX n'est ni une chaîne ni un HTML** :

```
const element = <h1>Hello, world!</h1>;
```

Il s'appelle JSX, et c'est une **extension de syntaxe pour JavaScript** . JSX peut vous rappeler un langage de modèle, mais il est livré avec toute la puissance de JavaScript.

L'équipe React indique dans ses documents qu'ils recommandent de l'utiliser pour décrire ce que l'interface utilisateur devrait ressembler.

Examples

Accessoires dans JSX

Il existe plusieurs façons de spécifier des accessoires dans JSX.

Expressions JavaScript

Vous pouvez transmettre **n'importe quelle expression JavaScript** en prop, en l'entourant de `{}` . Par exemple, dans cette JSX:

```
<MyComponent count={1 + 2 + 3 + 4} />
```

Dans le `MyComponent` , la valeur de `props.count` sera `10` , car l'expression `1 + 2 + 3 + 4` est évaluée.

Si les instructions et les boucles ne sont pas des expressions en JavaScript, elles ne peuvent pas être utilisées directement dans JSX.

Littéraux de chaîne

Bien sûr, vous pouvez simplement transmettre n'importe quelle `string literal` comme un `prop` . Ces deux expressions JSX sont équivalentes:

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

Lorsque vous passez un littéral de chaîne, sa valeur est HTML-sans-échappement. Donc, ces deux expressions JSX sont équivalentes:

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

Ce comportement n'est généralement pas pertinent. C'est seulement mentionné ici pour être complet.

Valeur par défaut des accessoires

Si vous ne transmettez aucune valeur pour un accessoire, la valeur par **défaut est** `true` . Ces deux expressions JSX sont équivalentes:

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

Cependant, l'équipe React affirme que **cette approche n'est pas recommandée** dans ses documents, car elle peut être confondue avec l'objet abrégé ES6 `{foo}` qui est l'abréviation de `{foo: foo}` plutôt que `{foo: true}` . Ils disent que ce comportement est juste là pour qu'il corresponde au comportement de HTML.

Attributs de diffusion

Si vous avez déjà des objets en tant qu'objet et que vous souhaitez les transmettre dans JSX, vous pouvez utiliser `...` tant qu'opérateur de diffusion pour transmettre l'objet de l'ensemble des objets. Ces deux composants sont équivalents:

```
function Case1() {
  return <Greeting firstName="Kaloyab" lastName="Kosev" />;
}

function Case2() {
  const person = {firstName: 'Kaloyan', lastName: 'Kosev'};
  return <Greeting {...person} />;
}
```

Enfants à JSX

Dans les expressions JSX qui contiennent à la fois une balise d'ouverture et une balise de fermeture, le contenu entre ces balises est transmis en tant que support spécial: `props.children` . Il y a plusieurs façons différentes de transmettre les enfants:

Littéraux de chaîne

Vous pouvez mettre une chaîne entre les balises d'ouverture et de fermeture et `props.children` sera simplement cette chaîne. Ceci est utile pour la plupart des éléments HTML intégrés. Par exemple:

```
<MyComponent>
    <h1>Hello world!</h1>
</MyComponent>
```

Ceci est valide JSX, et `props.children` dans MyComponent sera simplement `<h1>Hello world!</h1>` .

Notez que **le HTML n'est pas échappé** , vous pouvez donc généralement écrire JSX comme si vous écriviez du HTML.

N'oubliez pas que dans ce cas, JSX:

- supprime les espaces au début et à la fin d'une ligne;
- supprime les lignes vierges;
- les nouvelles lignes adjacentes aux étiquettes sont supprimées;
- les nouvelles lignes qui apparaissent au milieu des littéraux de chaîne sont condensées dans un seul espace.

JSX Enfants

Vous pouvez fournir plus d'éléments JSX que les enfants. Ceci est utile pour afficher les composants imbriqués:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

Vous pouvez **mélanger différents types d'enfants, vous pouvez donc utiliser des littéraux de chaîne avec des enfants JSX** . Ceci est une autre façon dont JSX est comme HTML, de sorte que ce soit à la fois JSX valide et HTML valide:

```
<div>
  <h2>Here is a list</h2>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Notez qu'un composant React **ne peut pas renvoyer plusieurs éléments React, mais qu'une seule expression JSX peut avoir plusieurs enfants** . Donc, si vous voulez qu'un composant rende plusieurs choses, vous pouvez les encapsuler dans un `div` comme l'exemple ci-dessus.

Expressions JavaScript

Vous pouvez transmettre toute expression JavaScript en tant qu'enfant, en la plaçant dans `{}` . Par exemple, ces expressions sont équivalentes:

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

Ceci est souvent utile pour rendre une liste d'expressions JSX de longueur arbitraire. Par exemple, cela affiche une liste HTML:

```
const Item = ({ message }) => (
  <li>{ message }</li>
);

const TodoList = () => {
  const todos = ['finish doc', 'submit review', 'wait stackoverflow review'];
  return (
    <ul>
      { todos.map(message => (<Item key={message} message={message} />)) }
    </ul>
  );
};
```

Notez que les expressions JavaScript peuvent être mélangées avec d'autres types d'enfants.

Fonctionne comme des enfants

Normalement, les expressions JavaScript insérées dans JSX seront évaluées par une chaîne, un élément React ou une liste de ces éléments. Cependant, `props.children` fonctionne comme n'importe quel autre accessoire en ce sens qu'il peut transmettre n'importe quelle sorte de données, pas seulement celles que React sait rendre. Par exemple, si vous avez un composant personnalisé, vous pouvez le demander en tant que `props.children` :

```
const ListOfTenThings = () => (
  <Repeat numTimes={10}>
    {(index) => <div key={index}>This is item {index} in the list</div>}
  </Repeat>
);

// Calls the children callback numTimes to produce a repeated component
const Repeat = ({ numTimes, children }) => {
  let items = [];
  for (let i = 0; i < numTimes; i++) {
    items.push(children(i));
  }
  return <div>{items}</div>;
};
```

Les enfants passés à un composant personnalisé peuvent être n'importe quoi, tant que ce composant les transforme en quelque chose que React peut comprendre avant le rendu. Cet usage n'est pas courant, mais cela fonctionne si vous voulez étirer ce que JSX est capable de faire.

Valeurs ignorées

Notez que `false` , `null` , `undefined` et `true` sont des enfants valides. Mais ils ne rendent tout simplement pas. Ces expressions JSX rendront toutes la même chose:

```
<MyComponent />

<MyComponent></MyComponent>

<MyComponent>{false}</MyComponent>

<MyComponent>{null}</MyComponent>

<MyComponent>{true}</MyComponent>
```

Ceci est extrêmement utile pour rendre conditionnellement des éléments React. Ce JSX ne rend qu'un si `showHeader` est vrai:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

Une mise en garde importante est que certaines valeurs "fausses", telles que le nombre `0` , sont toujours rendues par React. Par exemple, ce code ne se comportera pas comme prévu, car `0` sera imprimé lorsque `props.messages` est un tableau vide:

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

Une approche pour résoudre ce problème consiste à s'assurer que l'expression avant le `&&` est toujours booléenne:

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

Enfin, gardez à l'esprit que si vous voulez qu'une valeur comme `false` , `true` , `null` ou `undefined` apparaisse dans la sortie, vous devez d'abord la convertir en chaîne:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

Lire JSX en ligne: https://riptutorial.com/fr/reactjs/topic/8027/jsx

Chapitre 18: Les accessoires en réaction

Remarques

REMARQUE: à partir de React 15.5 et plus, le composant PropTypes réside dans son propre package npm, à savoir «prop-types» et nécessite sa propre instruction d'importation lors de l'utilisation de PropTypes. Voir la documentation officielle de la réaction pour la rupture: https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html

Examples

introduction

`props` sont utilisées pour transmettre des données et des méthodes d'un composant parent à un composant enfant.

Des choses intéressantes sur les `props`

1. Ils sont immuables.
2. Ils nous permettent de créer des composants réutilisables.

Exemple de base

```
class Parent extends React.Component{
  doSomething(){
     console.log("Parent component");
  }
  render() {
    return <div>
         <Child
           text="This is the child number 1"
           title="Title 1"
           onClick={this.doSomething} />
         <Child
           text="This is the child number 2"
           title="Title 2"
           onClick={this.doSomething} />
      </div>
  }
}

class Child extends React.Component{
  render() {
    return <div>
       <h1>{this.props.title}</h1>
       <h2>{this.props.text}</h2>
      </div>
  }
}
```

Comme vous pouvez le voir dans l'exemple, grâce aux `props` nous pouvons créer des composants réutilisables.

Accessoires par défaut

`defaultProps` vous permet de définir des valeurs par défaut ou de secours pour vos `props` composant. `defaultProps` est utile lorsque vous appelez des composants de différentes vues avec des objets fixes, mais dans certaines vues, vous devez transmettre des valeurs différentes.

Syntaxe

ES5

```
var MyClass = React.createClass({
  getDefaultProps: function() {
    return {
      randomObject: {},
      ...
    };
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.defaultProps = {
    randomObject: {},
    ...
}
```

ES7

```
class MyClass extends React.Component {
    static defaultProps = {
        randomObject: {},
        ...
    };
}
```

Le résultat de `getDefaultProps()` ou `defaultProps` sera mis en cache et utilisé pour garantir que `this.props.randomObject` aura une valeur s'il n'était pas spécifié par le composant parent.

PropTypes

`propTypes` vous permet de spécifier les `props` votre composant a besoin et le type qu'il devrait être. Votre composant fonctionnera sans définir `propTypes` , mais il est `propTypes` de le définir car il rendra votre composant plus lisible, servira de documentation à d’autres développeurs qui lisent votre composant, et pendant le développement, React vous avertira si vous essayez de le faire.

définir un accessoire qui est un type différent de la définition que vous avez définie pour lui.

Certains primitifs `propTypes` et souvent utilisables `propTypes` sont -

```
  optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol
```

Si vous `isRequired` à un `propType` ce prop doit être fourni lors de la création de l'instance de ce composant. Si vous ne fournissez pas les `propTypes` **requis** , l'instance de composant ne peut pas être créée.

Syntaxe

ES5

```
var MyClass = React.createClass({
  propTypes: {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.propTypes = {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
};
```

ES7

```
class MyClass extends React.Component {
     static propTypes = {
        randomObject: React.PropTypes.object,
        callback: React.PropTypes.func.isRequired,
        ...
    };
}
```

Validation des accessoires plus complexes

De même, `PropTypes` vous permet de spécifier une validation plus complexe

Valider un objet

```
...
    randomObject: React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    }).isRequired,
...
```

Validation sur un tableau d'objets

```
...
    arrayOfObjects: React.PropTypes.arrayOf(React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    })).isRequired,
...
```

Faire passer des accessoires à l'aide d'un opérateur de répartition

Au lieu de

```
var component = <Component foo={this.props.x} bar={this.props.y} />;
```

Lorsque chaque propriété doit être transmise en tant que valeur unique, vous pouvez utiliser l'opérateur de propagation `...` pris en charge pour les baies dans ES6 afin de transmettre toutes vos valeurs. Le composant va maintenant ressembler à ceci.

```
var component = <Component {...props} />;
```

Rappelez-vous que les propriétés de l'objet que vous transmettez sont copiées sur les accessoires du composant.

La commande est importante. Les attributs ultérieurs remplacent les précédents.

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

Un autre cas est que vous pouvez également utiliser un opérateur de propagation pour ne transmettre que des parties des accessoires aux composants enfants, puis vous pouvez utiliser à nouveau la syntaxe de déstructuration des accessoires.

C'est très utile lorsque les composants enfants ont besoin de beaucoup d'accessoires, mais ne veulent pas les dépasser un par un.

```
const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };
var component = <Component {...{foo, bar}} />;
```

```
const { foo, bar } = component.props
console.log(foo, bar); // 'foo bar'
```

Props.children et composition des composants

Les composants "enfants" d'un composant sont disponibles sur un accessoire spécial, `props.children` . Ce prop est très utile pour les composants "Compositing", et peut rendre le balisage JSX plus intuitif ou refléter la structure finale prévue du DOM:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {this.props.children}
            </div>
        </article>
    );
}
```

Ce qui nous permet d'inclure un nombre arbitraire de sous-éléments lors de l'utilisation ultérieure du composant:

```
var ParentComponent = function () {
    return (
        <SomeComponent heading="Amazing Article Box" >
            <p className="first"> Lots of content </p>
            <p> Or not </p>
        </SomeComponent>
    );
}
```

Props.children peut également être manipulé par le composant. Parce que props.children peut ou peut ne pas être un tableau , React fournit des fonctions utilitaires comme React.Children . Considérons dans l'exemple précédent si nous avions voulu envelopper chaque paragraphe dans son propre élément `<section>` :

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {React.Children.map(this.props.children, function (child) {
                    return (
                        <section className={child.props.className}>
                            React.cloneElement(child)
                        </section>
                    );
                })}
            </div>
        </article>
    );
}
```

Notez l'utilisation de React.cloneElement pour supprimer les objets de la `<p>` child `<p>` - car les accessoires sont immuables, ces valeurs ne peuvent pas être modifiées directement. Au lieu de cela, un clone sans ces accessoires doit être utilisé.

De plus, lors de l'ajout d'éléments dans des boucles, sachez comment React réconcilie les enfants pendant un réexamen , et pensez fortement à inclure un `key` unique au niveau global sur les éléments enfants ajoutés dans une boucle.

Détecter le type de composants enfants

Parfois, il est très utile de connaître le type de composant enfant lors d'une itération à travers eux. Pour parcourir les composants enfants, vous pouvez utiliser la fonction React `Children.map` util:

```
React.Children.map(this.props.children, (child) => {
  if (child.type === MyComponentType) {
    ...
  }
});
```

L'objet enfant expose la propriété `type` que vous pouvez comparer à un composant spécifique.

Lire Les accessoires en réaction en ligne: https://riptutorial.com/fr/reactjs/topic/2749/les-accessoires-en-reaction

Chapitre 19: Performance

Examples

Les bases - DOM DOM vs DOM virtuel

HTML DOM est cher

Chaque page Web est représentée en interne en tant qu'arbre d'objets. Cette représentation est appelée *modèle d'objet de document* . En outre, il s'agit d'une interface indépendante du langage qui permet aux langages de programmation (tels que JavaScript) d'accéder aux éléments HTML.

En d'autres termes

> Le HTML DOM est un standard pour obtenir, modifier, ajouter ou supprimer des éléments HTML.

Cependant, ces **opérations DOM** sont extrêmement **coûteuses** .

Virtual DOM est une solution

L'équipe de React a donc eu l'idée d'abstraire le *DOM HTML* et de créer son propre *DOM virtuel* afin de calculer le nombre minimum d'opérations à appliquer sur le *DOM HTML* pour répliquer l'état actuel de notre application.

Le DOM virtuel permet de gagner du temps grâce aux modifications inutiles du DOM.

De quelle façon précisément?

À chaque instant, React a l'état de l'application représenté sous la forme d'un `Virtual DOM` . Chaque fois que l'état de l'application change, voici les étapes que React effectue pour optimiser les performances.

1. Générer un nouveau *DOM virtuel* qui représente le nouvel état de notre application
2. Comparer l'ancien DOM virtuel (qui représente le DOM HTML actuel) au nouveau DOM virtuel
3. Basé sur 2. trouver le nombre minimum d'opérations pour transformer l'ancien DOM virtuel (qui représente le DOM HTML actuel) dans le nouveau DOM virtuel

- pour en savoir plus à ce sujet - lisez l'algorithme Diff de React

4. Une fois ces opérations trouvées, elles sont mappées dans leurs opérations *DOM HTML* équivalentes

- rappelez-vous, le *DOM virtuel* est seulement une abstraction du *DOM HTML* et il existe une relation isomorphe entre eux

5. Maintenant, le nombre minimum d'opérations qui ont été trouvées et transférées vers leurs opérations *HTML* équivalentes est maintenant appliqué directement sur le *DOM HTML de* l'application, ce qui permet de gagner du temps en modifiant inutilement le *DOM HTML* .

Remarque: les opérations appliquées sur le DOM virtuel sont peu coûteuses, car le DOM virtuel est un objet JavaScript.

Algorithme diff de React

La génération du nombre minimum d'opérations pour transformer un arbre en un autre a une complexité de l'ordre de O (n ^ 3) où n est le nombre de nœuds dans l'arbre. React repose sur deux hypothèses pour résoudre ce problème dans un temps linéaire - O (n)

1. Deux composants de la même classe généreront des arbres similaires et deux composants de classes différentes généreront des arbres différents.

2. Il est possible de fournir une clé unique pour les éléments stables sur différents rendus.

Pour décider si deux nœuds sont différents, React différencie 3 cas

1. Deux nœuds sont différents s'ils ont des types différents.

- Par exemple, `<div>...</div>` est différent de `<span>...</span>`

2. Chaque fois que deux nœuds ont des clés différentes

- Par exemple, `<div key="1">...</div>` est différent de `<div key="2">...</div>`

De plus, **ce qui suit est crucial et extrêmement important pour comprendre** si vous souhaitez optimiser les performances.

> S'ils [deux nœuds] ne sont pas du même type, React ne va même pas essayer de faire correspondre ce qu'ils rendent. Il va juste supprimer le premier du DOM et insérer le second.

Voici pourquoi

> Il est très peu probable qu'un élément génère un DOM qui ressemblerait à ce que générerait un élément. Au lieu de passer du temps à essayer de faire correspondre ces deux structures, React ne fait que reconstruire l'arbre à partir de zéro.

Conseils & Astuces

Lorsque deux nœuds ne sont pas du même type, React ne tente pas de les faire correspondre - il supprime simplement le premier nœud du DOM et insère le second. C'est pourquoi la première astuce dit

1. Si vous vous voyez alterner entre deux classes de composants avec une sortie très similaire, vous pouvez en faire la même classe.

---

2. Utilisez shouldComponentUpdate pour empêcher le composant de réexécuter, si vous savez qu'il ne va pas changer, par exemple

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

Mesure de performance avec ReactJS

Vous ne pouvez pas améliorer quelque chose que vous ne pouvez pas mesurer . Pour améliorer les performances des composants React, vous devriez pouvoir le mesurer. ReactJS fournit des outils *addon* pour mesurer la performance. Importer le module `react-addons-perf` pour mesurer les performances

```
import Perf from 'react-addons-perf' // ES6
var Perf = require('react-addons-perf') // ES5 with npm
var Perf = React.addons.Perf; // ES5 with react-with-addons.js
```

Vous pouvez utiliser les méthodes ci-dessous à partir du module `Perf` importé:

- Perf.printInclusive ()
- Perf.printExclusive ()
- Perf.printWasted ()
- Perf.printOperations ()
- Perf.printDOM ()

`Perf.printWasted()` est la plus importante dont vous aurez besoin la plupart du temps. `Perf.printWasted()` vous donne la représentation sous forme de tableau du temps perdu de votre composant individuel.

(index)	Owner > component	Waste
0	"Todos > TodoItem"	102.7

Total time: 132.71 ms

Vous pouvez noter la colonne **Temps** perdu dans le tableau et améliorer les performances du composant en utilisant la section **Conseils et astuces** ci-dessus.

Reportez-vous au guide officiel React et à l'excellent article de Benchling Engg. sur la performance de réaction

Lire Performance en ligne: https://riptutorial.com/fr/reactjs/topic/6875/performance

Chapitre 20: React Boilerplate [React + Babel + Webpack]

Examples

Mise en place du projet

Vous avez besoin du gestionnaire de package Node pour installer les dépendances du projet. Nœud de téléchargement pour votre système d'exploitation à partir de Nodejs.org . Node Package Manager est fourni avec le noeud.

Vous pouvez également utiliser Node Version Manager pour mieux gérer vos versions de nœud et npm. C'est génial pour tester votre projet sur différentes versions de nœuds. Cependant, il n'est pas recommandé pour l'environnement de production.

Une fois que vous avez installé le nœud sur votre système, installez-vous et installez des packages essentiels pour dynamiser votre premier projet React en utilisant Babel et Webpack.

Avant de commencer à frapper des commandes dans le terminal. Jetez un oeil à ce que Babel et Webpack sont utilisés.

Vous pouvez démarrer votre projet en exécutant `npm init` dans votre terminal. Suivez la configuration initiale. Après cela, exécutez les commandes suivantes dans votre terminal -

Dépendances:

```
npm install react react-dom --save
```

Dépendances de Dev:

```
npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0
webpack webpack-dev-server react-hot-loader --save-dev
```

Dépendances optionnelles du dev:

```
npm install eslint eslint-plugin-react babel-eslint --save-dev
```

Vous pouvez vous référer à cet exemple de package.json

Créez `.babelrc` dans la racine de votre projet avec les contenus suivants:

```
{
  "presets": ["es2015", "stage-0", "react"]
}
```

Vous pouvez éventuellement créer `.eslintrc` dans la racine de votre projet avec le contenu suivant:

```
{
  "ecmaFeatures": {
    "jsx": true,
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
  },
  "plugins": [
    "react"
  ]
}
```

Créez un fichier `.gitignore` pour empêcher le téléchargement des fichiers générés sur votre dépôt git.

```
node_modules
npm-debug.log
.DS_Store
dist
```

Créez le fichier `webpack.config.js` avec le contenu minimum suivant.

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  module: {
    loaders: [{
      test: /\.js$/,
      loaders: ['react-hot', 'babel'],
      include: path.join(__dirname, 'src')
    }]
  }
};
```

Et enfin, créez un fichier `sever.js` pour pouvoir exécuter `npm start` , avec le contenu suivant:

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('./webpack.config');

new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
  if (err) {
    return console.log(err);
  }

  console.log('Serving your awesome project at http://localhost:3000/');
});
```

Créez le fichier `src/app.js` pour voir votre projet React faire quelque chose.

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

export default class App extends Component {
  render() {
    return (
      <h1>Hello, world.</h1>
    );
  }
}
```

Lancez `node server.js` ou `npm start` dans le terminal si vous avez défini ce que signifie `start` dans votre `package.json`

projet réactif

A propos de ce projet

Ceci est un projet simple passe-partout. Cet article vous guidera pour configurer l'environnement pour ReactJs + Webpack + Bable.

Commençons

nous aurons besoin d'un gestionnaire de paquets nodaux pour lancer le serveur express et gérer les dépendances tout au long du projet. Si vous êtes nouveau sur le gestionnaire de paquets, vous pouvez vérifier ici . Remarque: L'installation du gestionnaire de packages de nœuds est requise ici.

Créez un dossier avec le nom approprié et naviguez depuis celui-ci depuis le terminal ou par l'interface graphique. Allez ensuite au terminal et tapez `npm init` cela créera un fichier package.json. description, point d'entrée, dépôt git, auteur, licence, etc. Ici, le point d'entrée est important car le nœud le recherchera initialement lorsque vous exécuterez le projet. À la fin, il vous demandera de vérifier les informations que vous fournissez. Vous pouvez taper *oui* ou le modifier. Eh bien c'est ça, notre fichier *package.json* est prêt.

La configuration du serveur Express est exécutée *npm install express @ 4 --save* . C'est toutes

les dépendances dont nous avions besoin pour ce projet. Il est important de sauvegarder l'indicateur, sans *lequel le* fichier *package.js* ne sera pas mis à jour. La tâche principale de *package.json* est de stocker la liste des dépendances. Il ajoutera express version 4. Votre *package.json* ressemblera à `"dependencies": { "express": "^4.13.4", ..... },`

Une fois le téléchargement terminé, vous pouvez voir qu'il existe *un* dossier *node_modules* et un sous-dossier de nos dépendances. Maintenant, à la racine du projet, créez un nouveau fichier *server.js* . Maintenant, nous mettons en place un serveur express. Je vais passer tout le code et l'expliquer plus tard.

```
var express = require('express');
// Create our app
var app = express();

app.use(express.static('public'));

app.listen(3000, function () {
  console.log('Express server is using port:3000');
});
```

var express = require ('express'); cela vous donnera accès à l'intégralité de l'api express.

var app = express (); appellera la bibliothèque express comme fonction. *app.use ();* laissez la fonctionnalité ajouter à votre application express. *app.use (express.static ('public'));* spécifiera le nom du dossier qui sera exposé dans notre serveur Web. *app.listen (port, function () {})* ici sera notre port *3000* et la fonction que nous appelons vérifiera que le serveur web est exécuté correctement. C'est ça le serveur express est mis en place.

Allez maintenant dans notre projet et créez un nouveau dossier public et créez le fichier *index.html* . *index.html* est le fichier par défaut pour votre application et Express server recherchera ce fichier. Le *fichier index.html* est un simple fichier html qui ressemble à

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
</head>

<body>
  <h1>hello World</h1>
</body>

</html>
```

Et accédez au chemin du projet via le terminal et tapez *node server.js* . Ensuite, vous verrez * console.log ('le serveur Express utilise le port: 3000'); *.

Allez dans le navigateur et tapez *http: // localhost: 3000* dans la barre de navigation, vous verrez *hello World* .

Allez maintenant dans le dossier public et créez un nouveau fichier *app.jsx* . JSX est une étape de

préprocesseur qui ajoute une syntaxe XML à votre code JavaScript. Vous pouvez certainement utiliser React sans JSX, mais JSX rend React beaucoup plus élégant. Voici l'exemple de code pour *app.jsx*

```
ReactDOM.render(
  <h1>Hello World!!!</h1>,
  document.getElementById('app')
);
```

Maintenant, allez dans *index.html* et modifiez le code, il devrait ressembler à ceci

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23
/browser.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js">
</script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js">    </script>
</head>

<body>
  <div id="app"></div>

  <script type="text/babel" src="app.jsx"></script>
</body>

</html>
```

Avec ceci en place vous êtes tous finis, j'espère que vous trouvez cela simple.

Lire React Boilerplate [React + Babel + Webpack] en ligne:
https://riptutorial.com/fr/reactjs/topic/5969/react-boilerplate--react-plus-babel-plus-webpack-

Chapitre 21: React.createClass vs extensions React.Component

Syntaxe

- Cas 1: React.createClass ({})
- Cas 2: la classe MyComponent étend React.Component {}

Remarques

`React.createClass` est `React.createClass` obsolète dans la version 15.5 et devrait être supprimé dans la version 16 . Il existe un package de remplacement pour ceux qui en ont encore besoin. Les exemples qui l'utilisent doivent être mis à jour.

Examples

Créer un composant réactif

Explorons les différences de syntaxe en comparant deux exemples de code.

React.createClass (obsolète)

Nous avons ici un `const` avec une classe React affectée, avec la fonction de `render` suivant pour compléter une définition de composant de base typique.

```
import React from 'react';

const MyComponent = React.createClass({
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Prenons la définition React.createClass ci-dessus et convertissons-la pour utiliser une classe ES6.

```
import React from 'react';
```

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

export default MyComponent;
```

Dans cet exemple, nous utilisons maintenant les classes ES6. Pour les modifications de React, nous créons maintenant une classe appelée **MyComponent** et nous étendons à partir de React.Component au lieu d'accéder directement à React.createClass. De cette façon, nous utilisons moins de Reactplate React et plus de JavaScript.

PS: Typiquement, cela serait utilisé avec quelque chose comme Babel pour compiler l'ES6 en ES5 pour fonctionner dans d'autres navigateurs.

Déclarer les accessoires par défaut et les propTypes

Il y a des changements importants dans la façon dont nous utilisons et déclarons les accessoires par défaut et leurs types.

React.createClass

Dans cette version, la propriété `propTypes` est un objet dans lequel nous pouvons déclarer le type de chaque accessoire. La propriété `getDefaultProps` est une fonction qui renvoie un objet pour créer les accessoires initiaux.

```
import React from 'react';

const MyComponent = React.createClass({
  propTypes: {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  },
  getDefaultProps() {
    return {
      name: 'Home',
      position: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Cette version utilise `propTypes` tant que propriété sur la classe **MyComponent** réelle au lieu d'une propriété faisant partie de l' `createClass` définition `createClass` .

La `getDefaultProps` a été remplacée par une propriété Object sur la classe appelée defaultProps, car ce n'est plus une fonction "get", c'est juste un objet. Cela évite plus de réactions, c'est tout simplement JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
MyComponent.propTypes = {
  name: React.PropTypes.string,
  position: React.PropTypes.number
};
MyComponent.defaultProps = {
  name: 'Home',
  position: 1
};

export default MyComponent;
```

En outre, il existe une autre syntaxe pour `propTypes` et `defaultProps` . Il s'agit d'un raccourci si les initialiseurs de propriété ES7 de votre version sont activés:

```
import React from 'react';

class MyComponent extends React.Component {
  static propTypes = {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  };
  static defaultProps = {
    name: 'Home',
    position: 1
  };
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
```

```
export default MyComponent;
```

Définir l'état initial

Il y a des changements dans la façon dont nous définissons les états initiaux.

React.createClass

Nous avons une fonction `getInitialState` , qui renvoie simplement un objet des états initiaux.

```
import React from 'react';

const MyComponent = React.createClass({
  getInitialState () {
    return {
      activePage: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Dans cette version, nous déclarons tous les états comme une simple **propriété d'initialisation dans le constructeur** , au lieu d'utiliser la fonction `getInitialState` . Il se sent moins "réactif API" car il s'agit simplement de JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activePage: 1
    };
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

Mixins

Nous ne pouvons utiliser les `mixins` qu'avec la méthode React.createClass.

React.createClass

Dans cette version, nous pouvons ajouter des `mixins` aux composants en utilisant la propriété mixins qui prend un tableau de mixins disponibles. Ceux-ci étendent alors la classe de composant.

```
import React from 'react';

var MyMixin = {
  doSomething() {

  }
};
const MyComponent = React.createClass({
  mixins: [MyMixin],
  handleClick() {
    this.doSomething(); // invoke mixin's method
  },
  render() {
    return (
      <button onClick={this.handleClick}>Do Something</button>
    );
  }
});

export default MyComponent;
```

React.Component

Les mix React ne sont pas pris en charge lors de l'utilisation de composants React écrits en ES6. De plus, ils ne prendront pas en charge les classes ES6 dans React. La raison en est qu'ils sont considérés comme dangereux .

"this" Contexte

L'utilisation de React.createClass lie automatiquement `this` contexte (valeurs) correctement, mais ce n'est pas le cas lors de l'utilisation de classes ES6.

React.createClass

Notez la déclaration `onClick` avec la méthode `this.handleClick` liée. Lorsque cette méthode est appelée, React appliquera le bon contexte d'exécution au `handleClick` .

```
import React from 'react';
```

```
const MyComponent = React.createClass({
  handleClick() {
    console.log(this); // the React Component instance
  },
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
});

export default MyComponent;
```

React.Component

Avec les classes ES6, `this` est `null` par défaut, les propriétés de la classe ne sont pas automatiquement liées à l'instance de la classe React (composant).

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // null
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Il y a quelques façons dont nous pourrions lier le droit `this` contexte.

Cas 1: Lier en ligne:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick.bind(this)}></div>
```

```
    );
  }
}

export default MyComponent;
```

Cas 2: Liaison dans le constructeur de classe

Une autre approche consiste à changer le contexte de `this.handleClick` dans le `constructor` . De cette façon, nous évitons les répétitions en ligne. Considéré par beaucoup comme une meilleure approche qui évite de toucher à JSX:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Cas 3: Utiliser la fonction anonyme ES6

Vous pouvez également utiliser la fonction anonyme ES6 sans avoir à vous lier explicitement:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick = () => {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

ES6 / Réagir le mot-clé "this" avec ajax pour récupérer les données du

serveur

```
import React from 'react';

class SearchEs6 extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            searchResults: []
        };
    }

    showResults(response){
        this.setState({
            searchResults: response.results
        })
    }

    search(url){
        $.ajax({
            type: "GET",
            dataType: 'jsonp',
            url: url,
            success: (data) => {
                this.showResults(data);
            },
            error: (xhr, status, err) => {
                console.error(url, status, err.toString());
            }
        });
    }

    render() {
        return (
            <div>
                <SearchBox search={this.search.bind(this)} />
                <Results searchResults={this.state.searchResults} />
            </div>
        );
    }
}
```

Lire React.createClass vs extensions React.Component en ligne:
https://riptutorial.com/fr/reactjs/topic/6371/react-createclass-vs-extensions-react-component

Chapitre 22: Réagir au routage

Examples

Exemple de fichier Routes.js, suivi de l'utilisation du lien du routeur dans le composant

Placez un fichier comme le suivant dans votre répertoire de niveau supérieur. Il définit les composants à rendre pour quels chemins

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';

import Index from './containers/home';
import App from './components/app';

export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="posts/new" component={New} />
    <Route path="posts/:id" component={Show} />

  </Route>
);
```

Maintenant, dans votre index.js de niveau supérieur qui est votre point d'entrée vers l'application, il vous suffit de rendre ce composant de routeur comme suit:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// import the routes component we created in routes.js
import routes from './routes';


// entry point
ReactDOM.render(
    <Router history={browserHistory} routes={routes} />
  , document.getElementById('main'));
```

Il ne vous reste plus qu'à utiliser `Link` au lieu de `<a>` tags dans votre application. Utiliser Link communiquera avec React Router pour changer la route React Router vers le lien spécifié, ce qui à son tour rendra le composant correct défini dans routes.js

```
import React from 'react';
import { Link } from 'react-router';

export default function PostButton(props) {
  return (
```

```
    <Link to={`posts/${props.postId}`}>
      <div className="post-button" >
        {props.title}
        <span>{props.tags}</span>
      </div>
    </Link>
  );
}
```

Réagir au routage asynchrone

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';

import Index from './containers/home';
import App from './components/app';

//for single Component lazy load use this
const ContactComponent = () => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          callback(null, require('./components/Contact')["default"]);
        }, 'Contact');
      }
    }
};

//for multiple componnets
 const groupedComponents = (pageName) => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          switch(pageName){
            case 'about' :
                callback(null, require( "./components/about" )["default"]);
                    break ;
            case 'tos' :
                callback(null, require( "./components/tos" )["default"]);
                    break ;
          }
        }, "groupedComponents");
      }
    }
};
export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="/contact" {...ContactComponent()} />
    <Route path="/about" {...groupedComponents('about')} />
    <Route path="/tos" {...groupedComponents('tos')} />
  </Route>
);
```

Lire Réagir au routage en ligne: https://riptutorial.com/fr/reactjs/topic/6096/reagir-au-routage

Chapitre 23: Réagir aux formes

Examples

Composants contrôlés

Un composant contrôlé est lié à une valeur et ses modifications sont traitées dans le code à l'aide de rappels basés sur des événements.

```
class CustomForm extends React.Component {
constructor() {
    super();
    this.state = {
        person: {
            firstName: '',
            lastName: ''
        }
    }
}

handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}

render() {
    return (
        <form>
            <input
            type="text"
            name="firstName"
            value={this.state.firstName}
            onChange={this.handleChange.bind(this)} />

            <input
            type="text"
            name="lastName"
            value={this.state.lastName}
            onChange={this.handleChange.bind(this)} />
        </form>
    )
}
```

}

Dans cet exemple, nous initialisons l'état avec un objet personne vide. Nous lions ensuite les valeurs des 2 entrées aux clés individuelles de l'objet personne. Ensuite, en tant qu'utilisateur, nous capturons chaque valeur de la fonction `handleChange` . Étant donné que les valeurs des composants sont liées à l'état, nous pouvons réexécuter en tant que type d'utilisateur en appelant `setState()` .

REMARQUE: Ne pas appeler `setState()` avec des composants contrôlés provoquera la saisie de

l'utilisateur, mais ne verra pas l'entrée car React ne restitue que les modifications quand il lui est demandé de le faire.

Il est également important de noter que les noms des entrées sont identiques aux noms des clés dans l'objet personne. Cela nous permet de capturer la valeur dans la forme du dictionnaire comme vu ici.

```
handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}
```

`person[event.target.name]` est la même chose que `person.firstName || person.lastName` Bien sûr, cela dépend de l'entrée en cours de saisie. Comme nous ne savons pas où l'utilisateur va taper, en utilisant un dictionnaire et en faisant correspondre les noms d'entrée avec les noms des clés, cela nous permet de capturer la saisie utilisateur. Peu importe où onChange est appelé.

Lire Réagir aux formes en ligne: https://riptutorial.com/fr/reactjs/topic/8047/reagir-aux-formes

Chapitre 24: Réagir aux outils

Examples

Liens

Endroits où trouver des composants et des bibliothèques React;

- Catalogue des composants de réaction
- JS.coach

Lire Réagir aux outils en ligne: https://riptutorial.com/fr/reactjs/topic/6595/reagir-aux-outils

Chapitre 25: Réagir avec Redux

Introduction

Redux est devenu le statu quo pour la gestion de l'état au niveau de l'application au début, et ceux qui travaillent sur des «applications à grande échelle» ne jurent que par lui. Cette rubrique explique pourquoi et comment utiliser la bibliothèque de gestion des états, Redux, dans vos applications React.

Remarques

Bien que l'architecture pilotée par composants de React soit fantastique pour décomposer l'application en petites pièces modulaires et encapsulées, elle présente certains défis pour la gestion de l'état de l'application dans son ensemble. Le temps d'utiliser Redux, c'est lorsque vous devez afficher les mêmes données sur plusieurs composants ou pages (route connue). À ce stade, vous ne pouvez plus stocker les données dans des variables locales à un composant ou à un autre, et l'envoi de messages entre les composants devient rapidement désordonné. Avec Redux, vos composants sont tous abonnés aux mêmes données partagées dans le magasin et, par conséquent, l'état peut être facilement répercuté de manière cohérente sur l'ensemble de l'application.

Examples

Utiliser Connect

Créez un magasin Redux avec *createStore* .

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp, { inistialStateVariable: "derp"})
```

Utilisez *connect* pour connecter le composant au magasin Redux et extrayez les accessoires d'un magasin à l'autre.

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Définissez les actions permettant à vos composants d'envoyer des messages au magasin Redux.

```
/*
```

```
 * action types
 */

export const ADD_TODO = 'ADD_TODO'

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

Gérez ces messages et créez un nouvel état pour le magasin dans les fonctions du réducteur.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Lire Réagir avec Redux en ligne: https://riptutorial.com/fr/reactjs/topic/10856/reagir-avec-redux

Chapitre 26: Réagissez à l'appel AJAX

Examples

Demande HTTP GET

Parfois, un composant doit afficher des données à partir d'un point de terminaison distant (par exemple, une API REST). Une pratique courante consiste à effectuer de tels appels dans la méthode `componentDidMount` .

Voici un exemple d'utilisation de superagent en tant qu'assistant AJAX:

```
import React from 'react'
import request from 'superagent'

class App extends React.Component {
  constructor () {
    super()
    this.state = {}
  }
  componentDidMount () {
    request
      .get('/search')
      .query({ query: 'Manny' })
      .query({ range: '1..5' })
      .query({ order: 'desc' })
      .set('API-Key', 'foobar')
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({someData: resp.text})
        }
      })
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

React.render(<App />, document.getElementById('root'))
```

Une demande peut être lancée en appelant la méthode appropriée sur l'objet de `request` , puis en appelant `.end()` pour envoyer la demande. Définir des champs d'en-tête est simple, `.set()` avec un nom de champ et une valeur.

La méthode `.query()` accepte les objets qui, utilisés avec la méthode GET, formeront une chaîne de requête. Ce qui suit produira le chemin `/search?query=Manny&range=1..5&order=desc` .

Demandes **POST**

```
request.post('/user')
  .set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
  .end(callback)
```

Reportez- vous à la documentation de Superagent pour plus de détails.

Ajax in React sans une bibliothèque tierce - aka avec VanillaJS

Ce qui suit fonctionnerait dans IE9 +

```
import React from 'react'

class App extends React.Component {
  constructor () {
    super()
    this.state = {someData: null}
  }
  componentDidMount () {
    var request = new XMLHttpRequest();
    request.open('GET', '/my/url', true);

    request.onload = () => {
      if (request.status >= 200 && request.status < 400) {
        // Success!
        this.setState({someData: request.responseText})
      } else {
        // We reached our target server, but it returned an error
        // Possibly handle the error by changing your state.
      }
    };

    request.onerror = () => {
      // There was a connection error of some sort.
      // Possibly handle the error by changing your state.
    };

    request.send();
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

React.render(<App />, document.getElementById('root'))
```

Demande HTTP GET et mise en boucle des données

L'exemple suivant montre comment un ensemble de données provenant d'une source distante peut être rendu dans un composant.

Nous effectuons une requête AJAX en utilisant `fetch` , qui est intégrée à la plupart des navigateurs. Utilisez un polyfill d' `fetch` en production pour prendre en charge les anciens navigateurs. Vous pouvez également utiliser toute autre bibliothèque pour effectuer des requêtes

(par exemple, axios , SuperAgent , ou même Javascript).

Nous définissons les données que nous recevons comme état du composant, nous pouvons donc y accéder à l'intérieur de la méthode de rendu. Là, nous parcourons les données en utilisant la `map` . N'oubliez pas de toujours ajouter un attribut de `key` unique (ou prop) à l'élément en boucle, ce qui est important pour les performances de rendu de React.

```
import React from 'react';

class Users extends React.Component {
  constructor() {
    super();
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('/api/users')
      .then(response => response.json())
      .then(json => this.setState({ users: json.data }));
  }

  render() {
    return (
      <div>
        <h1>Users</h1>
        {
          this.state.users.length == 0
            ? 'Loading users...'
            : this.state.users.map(user => (
              <figure key={user.id}>
                <img src={user.avatar} />
                <figcaption>
                  {user.name}
                </figcaption>
              </figure>
            ))
        }
      </div>
    );
  }
}

ReactDOM.render(<Users />, document.getElementById('root'));
```

Exemple de travail sur JSBin .

Lire Réagissez à l'appel AJAX en ligne: https://riptutorial.com/fr/reactjs/topic/6432/reagissez-a-l-appel-ajax

Chapitre 27: Solutions d'interface utilisateur

Introduction

Disons que nous nous inspirons des idées des interfaces utilisateur modernes utilisées dans les programmes et que nous les convertissons en composants React. C'est en cela que consiste le sujet " **Solutions d'interface utilisateur** ". L'attribution est appréciée.

Examples

Volet de base

```
import React from 'react';

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

    render() {
        return React.createElement(
            'section', this.props
        );
    }
}
```

Panneau

```
import React from 'react';

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

    render(...elements) {
        var props = Object.assign({
            className: this.props.active ? 'active' : '',
            tabIndex: -1
        }, this.props);

        var css = this.css();
        if (css != '') {
            elements.unshift(React.createElement(
                'style', null,
                css
            ));
        }

        return React.createElement(
            'div', props,
            ...elements
```

```
        );
    }

    static title() {
        return '';
    }
    static css() {
        return '';
    }
}
```

Les principales différences par rapport au volet simple sont les suivantes:

- le panneau a le focus par exemple lorsqu'il est appelé par script ou cliqué par la souris;
- panneau a `title` méthode statique par composant, de sorte qu'il peut être prolongé par d' autres composants du panneau avec redéfinie `title` (la raison ici est que la fonction peut alors être appelé à nouveau sur le rendu à des fins de localisation, mais dans les limites de cet exemple le `title` n'a pas de sens) ;
- il peut contenir des feuilles de style individuelles déclarées dans la méthode statique `css` (vous pouvez pré-charger le contenu des fichiers depuis `PANEL.css`).

Languette

```
import React from 'react';

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

    render() {
        var props = Object.assign({
            className: this.props.active ? 'active' : ''
        }, this.props);
        return React.createElement(
            'li', props,
            React.createElement(
                'span', props,
                props.panelClass.title()
            )
        );
    }
}
```

`panelClass` propriété `panelClass` de l'occurrence `Tab` doit contenir la classe de *panneau* utilisée pour la description.

PanelGroup

```
import React from 'react';
import Tab from './Tab.js';

class PanelGroup extends React.Component {
    constructor(props) {
```

```
        super(props);
        this.setState({
            panels: props.panels
        });
    }

    render() {
        this.tabSet = [];
        this.panelSet = [];
        for (let panelData of this.state.panels) {
            var tabIsActive = this.state.activeTab == panelData.name;
            this.tabSet.push(React.createElement(
                Tab, {
                    name: panelData.name,
                    active: tabIsActive,
                    panelClass: panelData.class,
                    onMouseDown: () => this.openTab(panelData.name)
                }
            ));
            this.panelSet.push(React.createElement(
                panelData.class, {
                    id: panelData.name,
                    active: tabIsActive,
                    ref: tabIsActive ? 'activePanel' : null
                }
            ));
        }
        return React.createElement(
            'div', { className: 'PanelGroup' },
            React.createElement(
                'nav', null,
                React.createElement(
                    'ul', null,
                    ...this.tabSet
                )
            ),
            ...this.panelSet
        );
    }

    openTab(name) {
        this.setState({ activeTab: name });
        this.findDOMNode(this.refs.activePanel).focus();
    }
}
```

`panels` propriété `panels` de l'instance `PanelGroup` doit contenir un tableau avec des objets. Chaque objet déclare des données importantes sur les panneaux:

- **`name`** - identifiant du panneau utilisé par le script du contrôleur;
- **`class`** - **`class`** panel.

N'oubliez pas de définir la propriété `activeTab` au nom de l'onglet requis.

Clarification

Lorsque l'onglet est en bas, le panneau requis obtient le nom de la classe `active` sur l'élément

DOM (cela signifie qu'il sera visible) et il est maintenant concentré.

Exemple de vue avec `PanelGroup`s

```
import React from 'react';
import Pane from './components/Pane.js';
import Panel from './components/Panel.js';
import PanelGroup from './components/PanelGroup.js';

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

    render() {
        return React.createElement(
            'main', null,
            React.createElement(
                Pane, { id: 'common' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'console',
                                panelClass: ConsolePanel
                            },
                            {
                                name: 'figures',
                                panelClass: FiguresPanel
                            }
                        ],
                        activeTab: 'console'
                    }
                )
            ),
            React.createElement(
                Pane, { id: 'side' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'properties',
                                panelClass: PropertiesPanel
                            }
                        ],
                        activeTab: 'properties'
                    }
                )
            )
        );
    }
}

class ConsolePanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Console';
```

```
    }
}

class FiguresPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Figures';
    }
}

class PropertiesPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Properties';
    }
}
```

Lire Solutions d'interface utilisateur en ligne: https://riptutorial.com/fr/reactjs/topic/8112/solutions-d-interface-utilisateur

Chapitre 28: Utilisation de React avec Flow

Introduction

Comment utiliser le vérificateur de type Flow pour vérifier les types de composants React.

Remarques

Flux | Réagir

Examples

Utilisation de Flow pour vérifier les types de composants fonctionnels sans état

```
type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

const AppContainer =
  ({ posts, dispatch, children }: Props) => (
    <div className="main-app">
      <Header {...{ posts, dispatch }} />
      {children}
    </div>
  )
```

Utiliser Flow pour vérifier les types de prop

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

type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

class Posts extends Component {
  props: Props;

  render () {
    // rest of the code goes here
  }
}
```

Lire Utilisation de React avec Flow en ligne: https://riptutorial.com/fr/reactjs/topic/7918/utilisation-de-react-avec-flow

Chapitre 29: Utiliser ReactJS avec jQuery

Examples

ReactJS avec jQuery

Tout d'abord, vous devez importer la bibliothèque jquery. Nous devons également importer findDOmNode car nous allons manipuler le dom. Et évidemment, nous importons également React.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';
```

Nous configurons une fonction de flèche «handleToggle» qui se déclenchera quand une icône sera cliquée. Nous montrons et masquons simplement un div avec un nom de référence "basculer" surCliquez sur une icône.

```
handleToggle = () => {
    const el = findDOMNode(this.refs.toggle);
    $(el).slideToggle();
};
```

Définissons maintenant le nom de référence 'toggle'

```
<ul className="profile-info additional-profile-info-list" ref="toggle">
  <li>
    <span className="info-email">Office Email</span>   me@shuvohabib.com
  </li>
</ul>
```

L'élément div où nous allons lancer le "handleToggle" sur onClick.

```
 <div className="ellipsis-click" onClick={this.handleToggle}>
   <i className="fa-ellipsis-h"/>
 </div>
```

Laissez le code complet ci-dessous, à quoi il ressemble.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';

export default class FullDesc extends React.Component {
    constructor() {
        super();
    }

    handleToggle = () => {
        const el = findDOMNode(this.refs.toggle);
```

```
        $(el).slideToggle();
    };

    render() {
        return (
            <div className="long-desc">
                <ul className="profile-info">
                    <li>
                        <span className="info-title">User Name : </span> Shuvo Habib
                    </li>
                </ul>

                <ul className="profile-info additional-profile-info-list" ref="toggle">
                    <li>
                        <span className="info-email">Office Email</span> me@shuvohabib.com
                    </li>
                </ul>

                <div className="ellipsis-click" onClick={this.handleToggle}>
                    <i className="fa-ellipsis-h"/>
                </div>
            </div>
        );
    }
}
```

Nous avons fini! Voici comment nous pouvons utiliser **jQuery dans le** composant **React** .

Lire Utiliser ReactJS avec jQuery en ligne: https://riptutorial.com/fr/reactjs/topic/6009/utiliser-reactjs-avec-jquery

Chapitre 30: Utiliser ReactJS avec Typescript

Examples

Composant ReactJS écrit en Typescript

En fait, vous pouvez utiliser les composants de ReactJS dans Typescript comme dans l'exemple de Facebook. Remplacez simplement l'extension du fichier 'jsx' par 'tsx':

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Mais pour utiliser pleinement la fonctionnalité principale de Typescript (vérification de type statique), il faut faire deux choses:

1) convertir l'exemple React.createClass en classe ES6:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

2) ajouter ensuite les interfaces Props et State:

```
interface IHelloMessageProps {
    name:string;
}

interface IHelloMessageState {
  //empty in our case
}

class HelloMessage extends React.Component<IHelloMessageProps, IHelloMessageState> {
  constructor(){
    super();
  }
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

Maintenant, Typescript affichera une erreur si le programmeur oublie de transmettre des accessoires. Ou s'ils ont ajouté des accessoires qui ne sont pas définis dans l'interface.

Composants réactifs sans état dans Typescript

Les composants en réaction qui sont des fonctions pures de leurs accessoires et qui ne nécessitent aucun état interne peuvent être écrits en tant que fonctions JavaScript au lieu d'utiliser la syntaxe de classe standard, comme suit:

```
import React from 'react'

const HelloWorld = (props) => (
    <h1>Hello, {props.name}!</h1>
);
```

La même chose peut être obtenue dans Typescript en utilisant la classe `React.SFC` :

```
import * as React from 'react';

class GreeterProps {
   name: string
}

const Greeter : React.SFC<GreeterProps> = props =>
    <h1>Hello, {props.name}!</h1>;
```

Notez que le nom `React.SFC` est un alias pour `React.StatelessComponent` So, soit peut être utilisé.

Installation et configuration

Pour utiliser typecript avec react dans un projet de noeud, vous devez d'abord avoir un répertoire de projet initialisé avec npm. Pour initialiser le répertoire avec `npm init`

Installation via npm ou yarn

Vous pouvez installer React en utilisant npm en procédant comme suit:

```
npm install --save react react-dom
```

Facebook a publié son propre gestionnaire de paquets nommé Yarn , qui peut également être utilisé pour installer React. Après avoir installé Yarn, il vous suffit d'exécuter cette commande:

```
yarn add react react-dom
```

Vous pouvez ensuite utiliser React dans votre projet exactement comme si vous aviez installé React via npm.

Installer les définitions de type de réaction dans Typescript 2.0+

Pour compiler votre code en utilisant dactylographié, ajoutez / installez des fichiers de définition de type en utilisant npm ou yarn.

```
npm install --save-dev @types/react @types/react-dom
```

ou, en utilisant du fil

```
yarn add --dev @types/react @types/react-dom
```

Installation des définitions de type de réaction dans les anciennes versions de Typescript

Vous devez utiliser un paquet séparé appelé tsd

```
tsd install react react-dom --save
```

Ajouter ou modifier la configuration de TypeScript

Pour utiliser JSX , un langage mélangeant le javascript avec html / xml, vous devez modifier la configuration du compilateur de typecript. Dans le fichier de configuration dactylographié du projet (généralement appelé `tsconfig.json`), vous devrez ajouter l'option JSX en tant que:

```
"compilerOptions": {
    "jsx": "react"
},
```

Cette option du compilateur demande essentiellement au compilateur typecript de traduire les balises JSX dans le code en appels de fonction javascript.

Pour éviter que le compilateur de typecript convertisse JSX en appels de fonctions javascript, utilisez

```
"compilerOptions": {
    "jsx": "preserve"
},
```

Composants sans état et sans propriété

Le composant de réaction le plus simple sans état et sans propriétés peut être écrit comme suit:

```
import * as React from 'react';

const Greeter = () => <span>Hello, World!</span>
```

Ce composant, cependant, ne peut pas accéder à `this.props` car `this.props` ne peut pas dire s'il s'agit d'un composant de réaction. Pour accéder à ses accessoires, utilisez:

```
import * as React from 'react';

const Greeter: React.SFC<{}> = props => () => <span>Hello, World!</span>
```

Même si le composant ne possède pas de propriétés explicitement définies, il peut désormais accéder à **`props.children`** car tous les composants ont des enfants par nature.

Un autre bon usage similaire des composants sans état et sans propriété réside dans la

modélisation simple des pages. Ce qui suit est un composant `Page` simple, en supposant qu'il existe des composants `Container` , `NavTop` et `NavBottom` hypothétiques dans le projet:

```
import * as React from 'react';

const Page: React.SFC<{}> = props => () =>
    <Container>
        <NavTop />
        {props.children}
        <NavBottom />
    </Container>

const LoginPage: React.SFC<{}> = props => () =>
    <Page>
        Login Pass: <input type="password" />
    </Page>
```

Dans cet exemple, le composant `Page` peut ensuite être utilisé par toute autre page réelle en tant que modèle de base.

Lire Utiliser ReactJS avec Typescript en ligne: https://riptutorial.com/fr/reactjs/topic/1419/utiliser-reactjs-avec-typescript

Chapitre 31: Utiliser ReactJS de manière Flux

Introduction

Il est très pratique d'utiliser l'approche Flux, lorsque votre application avec ReactJS sur frontend est prévue pour croître, en raison des structures limitées et d'un peu de nouveau code pour faciliter les changements d'état lors de l'exécution.

Remarques

Flux est l'architecture d'application utilisée par Facebook pour créer des applications Web côté client. Il complète les composants de la vue composable de React en utilisant un flux de données unidirectionnel. C'est plus un modèle qu'un cadre formel, et vous pouvez commencer à utiliser Flux immédiatement sans beaucoup de nouveau code.

Les applications flux ont trois parties principales: *le répartiteur* , *les magasins* et *les vues* (composants réactifs). Celles-ci ne doivent pas être confondues avec Model-View-Controller. Les contrôleurs existent dans une application Flux, mais ce sont des vues de contrôleurs, des vues souvent situées en haut de la hiérarchie, qui récupèrent les données des magasins et transmettent ces données à leurs enfants. De plus, les créateurs d'actions - méthodes d'assistance du répartiteur - sont utilisés pour prendre en charge une API sémantique qui décrit toutes les modifications possibles dans l'application. Il peut être utile de les considérer comme une quatrième partie du cycle de mise à jour de Flux.

Flux évite MVC en faveur d'un flux de données unidirectionnel. Lorsqu'un utilisateur interagit avec une vue Réaction, la vue propage une action via un répartiteur central vers les différents magasins contenant les données et la logique métier de l'application, qui met à jour toutes les vues affectées. Cela fonctionne particulièrement bien avec le style de programmation déclaratif de React, qui permet au magasin d'envoyer des mises à jour sans spécifier comment effectuer la transition entre les vues.

Examples

Flux de données

> Ceci est un aperçu de la vue d'ensemble complète.

Le modèle de flux suppose l'utilisation d'un flux de données unidirectionnel.

1. **Action** - objet simple décrivant le `type` action et d'autres données d'entrée.
2. **Dispatcher** - récepteur à action unique et contrôleur de rappel. Imaginez que c'est un hub central de votre application.
3. **Store** - contient l'état et la logique de l'application. Il enregistre les rappels dans le répartiteur

et émet des événements à afficher lorsque des modifications ont été apportées à la couche de données.

4. **View** - Réagir à un composant qui reçoit un événement de modification et des données du magasin. Cela provoque un rendu lorsque quelque chose est changé.

   > À partir du flux de données Flux, les vues peuvent également **créer des actions** et les transmettre au répartiteur pour les interactions utilisateur.

Renversé

Pour le rendre plus clair, nous pouvons commencer à la fin.

- Différents composants React (*vues*) obtiennent des données de différents magasins à propos des modifications apportées.

  > Peu de composants peuvent être appelés **contrôleurs-vues** , car ils fournissent le code de colle pour obtenir les données des magasins et pour transmettre des données dans la chaîne de leurs descendants. Les vues de contrôleur représentent toute section importante de la page.

- *Les magasins* peuvent être considérés comme des rappels qui comparent le type d'action et d'autres données d'entrée pour la logique métier de votre application.

- *Dispatcher* est un conteneur d'actions et de rappels d'actions communes.

- *Les actions* ne sont rien que des objets simples avec une propriété de `type` requise.

  > Auparavant, vous souhaiterez utiliser des constantes pour les types d'action et les méthodes d'assistance (appelées **créateurs d'action**).

Lire Utiliser ReactJS de manière Flux en ligne: https://riptutorial.com/fr/reactjs/topic/8158/utiliser-reactjs-de-maniere-flux

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec React	Adam, Adrián Daraš, Alex, Alex Young, Anuj, Bart Riordan, Cassidy, Community, Daksh Gupta, Dave Kaye, diabolicfreak, DMan, Donald, Everettss, Gianluca Esposito, himanshuIIITian, hyde, Ilya Lyamkin, Inanc Gumus, ivarni, jengeb, jolyonruss, Jon Chan, JordanHendrix, juandemarco, Kaloyan Kosev, Konstantin Grushetsky, Maksim, Marty, MaxPRafferty, Md. Nahiduzzaman Rose, Md.Sifatul Islam, Ming Soon, MMachinegun, Nick Bartlett, orvi, paqash, Prakash, rossipedia, Shabin Hashim, Simplans, Sunny R Gupta, TheShadowbyte, Timo, Tushar Khanna, user2314737
2	Clés en réaction	Dennis Stücken, thibmaek
3	Comment configurer un environnement Web de base, réagir et babel	Bart Riordan, Tien Do, Zac Braddy
4	Comment et pourquoi utiliser les clés dans React	Sammy I.
5	Communication entre composants	David, Kaloyan Kosev
6	Communiquer entre les composants	Random User
7	Composants	akashrajkn, Anuj, Bart Riordan, Bond, Brandon Roberts, Denis Ivanov, Diego V, DMan, Evan Hammer, Everettss, goldbullet, GordyD, hmnzr, Ilya Lyamkin, ivarni, Jagadish Upadhyay, jbmartinez, John Ruddell, jolyonruss, Jon Chan, jonathangoodman, JordanHendrix, justabuzz, k170, Kousha, Kyle Richardson, m_callens, Maayan Glikser, Michael Peyper, Paul Graffam, philpee2, QoP, Radu Brehar, Sai Vikas, sjmarshy , Timo, Vlad Bezden, WooCaSh, Zakaria Ridouh, zurfyx
8	Composants d'ordre supérieur	Dennis Stücken
9	Composants fonctionnels sans	Adam, Mark Lapierre, Mayank Shukla, Valter Júnior

	état	
10	Configuration de l'environnement réactif	ghostffcode, Tien Do
11	Cycle de vie des composants réactifs	Alex Young, Alexg2195, Anuj, Ashari, Everettss, F. Kauder, irrigator, John Ruddell, QoP, Salman Saleem, Saravana, Siddharth, skav, Timo, ultrasamad, Vivian, WitVault
12	Etat en réaction	Alex Young, Alexander, Brad Colthurst, Everettss, Kousha, Kyle Richardson, QoP, skav, Timo
13	Formulaires et saisie utilisateur	Everettss, Henrik Karlsson, ivarni, Timo
14	Installation	Rene R, Ruairi O'Brien
15	Installation React, Webpack & Typescript	Aron
16	Introduction au rendu côté serveur	Adrián Daraš, MauroPorrasP
17	JSX	Kaloyan Kosev, Ming Soon
18	Les accessoires en réaction	Ahmad, Anuj, Danillo Corvalan, Everettss, Faktor 10, Fellow Stranger, hansn, Ilya Lyamkin, Jack7, Jagadish Upadhyay, JimmyLv, MaxPRafferty, QoP, Sergii Bishyr, vintproykt, WitVault , zbynour
19	Performance	Aditya Singh, lustoykov, thibmaek
20	React Boilerplate [React + Babel + Webpack]	Mihir, parlad neupane, Tien Do
21	React.createClass vs extensions React.Component	Kaloyan Kosev, leonardoborges, Michael Peyper, pwolaq, Qianyue, sqzaman
22	Réagir au routage	abhirathore2006, Robeen
23	Réagir aux formes	promisified
24	Réagir aux outils	brillout
25	Réagir avec Redux	Jim

26	Réagissez à l'appel AJAX	adamboro, Fabian Schultz, Jason Bourne, lifeiscontent, McGrady, Sunny R Gupta
27	Solutions d'interface utilisateur	vintproykt
28	Utilisation de React avec Flow	JimmyLv, lifeiscontent, Rifat, Rory O'Kane
29	Utiliser ReactJS avec jQuery	Kousha, Shuvo Habib
30	Utiliser ReactJS avec Typescript	Everettss, John Ruddell, kevgathuku, Leone, Rajab Shakirov
31	Utiliser ReactJS de manière Flux	vintproykt