React.js

javascript et son écosystème

Jean-Christophe Routier
Licence mention Informatique
Université Lille – Sciences et Technologies

Université de Lille

Faculté des sciences et technologies
Département Informatique

- orienté vues, approche déclarative
- orienté composants
- efficacité basée sur un *DOM virtuel*

- installation

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

+ dans navigateur *React DevTools*

premier contact

react v0.1

```
// dans /src/main.js
import React from 'react';
import ReactDOM from 'react-dom';

var bootstrapReact =
  () => ReactDOM.render(
    React.createElement('h3', null, 'React in action !'),
    document.getElementById('insertReactHere')
  );

window.addEventListener( 'DOMContentLoaded', bootstrapReact );
```

react v0.2

webpack

mise en place

```
// dans webpack.config.js
module.exports = {
  entry: './src/main.js',

  output: {
    path: path.resolve(__dirname, 'public'),
    filename: 'javascripts/bundle.js'
  },...

// dans /public/index.html
...
<head>
  <script src="javascripts/bundle.js"></script>
 ...
<body>
  ...
  <div id="insertReactHere"></div>
```

jsx

une extension syntaxique à JavaScript

react v0.3

```
// dans /src/main.js
ReactDOM.render(
   <article>
     <h3>React in action, using jsx</h3>
     <div>this div is in an article</div>
     <div>this div is in the same article</div>
   </article>,
   document.getElementById('insertReactHere')
);
```

- une expression JSX n'est ni du javascript ni une chaîne de caractères

  $\Longrightarrow$ nécessite un *transpilage*

- une expression placée entre accolades { } est du code javascript interprété
- attribut : `class` $\rightsquigarrow$ `className`

babel

- **transpiler** le code (transformer/compiler)
- installation

  `npm install @babel/core babel-loader --save-dev`
- permettre l'utilisation d'évolutions de javascript non encore intégrées
  → exemple : les modules es6

  `npm install @babel/preset-env --save-dev`
- créer le fichier de configuration `.babelrc`

```
{
  "presets": ["@babel/preset-env"]
}
```

avec webpack

- application du transpilage avec webpack avec le `babel-loader`
  `npm install babel-loader --save-dev`
- ajout d'une règle dans `webpack.config.js`

```
  // dans webpack.config.js
module: {
   rules: [
     {
       test: /\.js$/,                    // pour les fichiers .js
       exclude: (/node_modules/),
       use: [
         { loader : 'babel-loader' }
       ]
     },
     ... // autres règles .css, images, etc.
   ]
 }
};
```

babel et jsx

transpilage JSX avec Babel :

- installation du module de transpilage pour react
  `npm install --save-dev @babel/preset-react`
- dans `.babelrc`, ajouter le module à `presets`
  `"presets": ["@babel/preset-env", "@babel/preset-react"]`
- application via `npm run build`

react v0.3

premiers composants

react v0.4
react v0.4.5

- composant **sans état** :
  fonction dont le résultat est le contenu du composant.
- le nom d'un composant commence pas une majuscule.

```
// dans /components/first.js
import React from 'react';
const First =
  () =>
    <article>
      ...
    </article>
export default First;
```

```
// dans /src/main.js
import First from '../components/first.js';
ReactDOM.render(<First />,
                document.getElementById('insertReactHere'));
```

propriétés

react v0.5

- via le paramètre `props` de la fonction constructeur
  JSX : valeurs passées comme des attributs du composant
- objet dont les propriétés sont non mutables

```
// dans /components/person.js
const Person =
  (props) =>
    <div>I am :
      <ul>
        <li>name => {props.name}</li>
        <li>age => {props.age}</li>
      </ul>
    </div>
```

```
// dans /src/main.js
  <Person name="timoleon" age = "12" />
  <Person name="Bilbo Baggins" age = "111" />
```

- les propriétés peuvent être des tableaux, des objets, etc.

react v0.5.5

```
// dans /components/listing.js
var Listing =
  (props) =>
    <div>
      <Person { ...props.persons[0] } />
      <Person { ...props.persons[1] } />
    </div>
```

NB : utilisation du *spread operator*

```
// dans /src/main.js
import persons from '../data/persons.js';          // un tableau d'objets
import Listing from '../components/listing.js';
var bootstrapReact =
  () => ReactDOM.render(
      <Listing persons={persons} />,
      ...
```

props.children

- `props.children` désigne les nœuds enfants fournis au composant lors de son utilisation

```
// dans /components/personListing.js
const PersonListing =
  (props) => <div>
                <Person { ...props.persons[0] } />
                <Person { ...props.persons[1] } />
                { props.children }
              </div>
```

```
// dans /src/main.js
const bootstrapReact =  () => ReactDOM.render(
                                <PersonListing persons={persons}>
                                  <h3>Liste des personnes</h3>
                                  <Person name="Nouveau" age=1 />
                                </PersonListing>, ...
```

react v0.6

classes de composants

- `extends React.component`
- les propriétés `props` sont en paramètre du constructeur
  - appel de `super(props)`
  - `props` devient `this.props`
- c'est la méthode `render()` qui renvoie la vue du composant
  le résultat ne peut avoir qu'un seul composant racine

react v1

```
// dans /components/person.js
import React from 'react';

export default class Person extends React.Component {
  constructor(props)
    super(props);
  }
  render() {
     return(
        <div className="person">I am :
          <dl>
            <dt>name</dt><dd> { this.props.name } </dd>
            <dt>age</dt><dd> { this.props.age } </dd>
          </dl>
        </div>
     );
  }
}
```

approche déclarative : on décrit dans `render()` ce que l'on veut avoir

defaultProps et propTypes

react v1.1

- valeur par défaut et contraintes sur les propriétés
- validation des valeurs des *props* à l'exécution, phase de développement
- message *warning* dans la console en cas de non respect

```
// dans /components/person.js
import PropTypes from 'prop-types';
export default class Person extends React.Component {
  ...
}
Person.defaultProps = {
  name : 'Anonymous'
}

Person.propTypes = {
  name : PropTypes.string,
  age : PropTypes.number.isRequired
}
```

composants à état

react v2

- rappel : `this.props` non mutable
- `this.state` représente l'état d'un composant : un objet initialisé dans le constructeur
- les modifications de l'état doivent être réalisées par des appels à la méthode `setState()` qui prend en paramètre un objet décrivant les modifications de `state`
- les modifications sont ensuite répercutées sur la vue générée par `render()`

événements

Les modifications de l'état peuvent résulter d'un évènement.

- les noms des événéments sont en notation "camelCase"
  `onclick` ⇝ `onClick`
- avec JSX c'est la fonction qui est fournie comme valeur
  `bind()` probablement nécessaire
  `onClick = {this.handleClick.bind(this)}`

react v2.1

```
// dans /components/star.js
export default class Star extends React.Component {
  constructor(props) {
    super(props);
    this.state = { on :  false };
  }
  selectSource() {
      if (this.state.on) return ... else return ...;
  }
  handleClick(event) {
      this.setState({ on :  true });
  }
  render() {
    return(
      <img src = { this.selectSource() }
           onClick= { this.handleClick.bind(this) }
      /> );
    }
}
```

state et props

- il ne faut utiliser `this.props` pour initialiser l'état uniquement si la *prop* utilisée a été explicitement définie uniquement dans le but de servir "valeur d'initialisation"

react v2.2

setState et état précédent

react v2.3

- les mises à jour de l'état (et de `props`) peuvent être asynchrones
- pour des raisons de performance React peut exécuter plusieurs `setState` en une fois
- donc, il ne faut pas baser la nouvelle valeur de l'état sur sa "valeur courante".
  Ceci peut ne pas être correct :

```
this.setState({on : ! this.state.on}); // peut poser un problème de
    cohérence
```

- dans un tel cas il faut utiliser la version de `setState` qui prend en paramètre une fonction qui sera appelée avec les valeurs qu'avaient de `state` et `props` au moment de la mise à jour :

```
this.setState( (prevstate, props) => ({ on : ! prevstate.on }) );
```

fusion des mises à jour

- si l'état est composé de plusieurs données, on peut les mettre à jour séparément,
  React se charge de faire la *fusion des mises à jour*

```
  constructor(props) {
    super(props);
    this.state = {
      name : "anonymous",
      age : 18
    };
  }
```

Les mises à jour peuvent être faites indépendamment si besoin :

```
changeValues(newName) {
   this.setState({ name : newName });
   ... some operations ...
   let newAge = ...;
   this.setState({ age : newAge });
}
```

liste de composants

- on peut créer des collections de composants et les inclure dans JSX avec { }.
- lors de la production d'une collection de composants, il est nécessaire d'attribuer à chaque composant une clé unique dans la liste

react v2.4-bad

```
render() {
  let stars = new Array(5).fill(0).map (
          (e,i) => <Star on={ i < this.state.rating } />
      );
  return(
    <div className="rating">
      { stars }
    </div>
    );
  }
```

```
Warning:  Each child in an array or iterator should have a unique
"key" prop
```

attribut key

- il faut définir l'attribut `key`
  permet d'identifier les éléments ajoutés, supprimés ou modifiés

react v2.4-ok

```
render() {
  let stars = new Array(5).fill(0).map(
          (e,i) => <Star on= { i < this.state.rating } key={i} />
      );
  return(
    <div className="rating">
      { stars }
    </div>
  );
}
```

typiquement une clef unique type *id* d'une base de données, à défaut numéro d'ordre dans la liste

placer l'état au plus haut

- dans React les valeurs sont transmises uniquement de « haut en bas » entre les composants
- une modification de l'état d'un enfant n'a pas d'effet sur un parent
- quand les changements d'une valeur impactent plusieurs composants, il faut la définir dans l'état du composant « le plus haut placé » c'est-à-dire dans le composant racine commun à tous les composants concernés
- les demandes de changement par un enfant sont gérées par une fonction du parent concerné
  cette fonction est transmise par le parent en tant que `props`

application

react v2.5

C'est la valeur de « *rating* » qui détermine les valeurs des « *stars* ». Il faut gérer les modifications d'état au niveau de ce composant.

- l'état `value` est défini dans `Rating`
- dans `Star` la valeur de `on` est passée comme `props`, elle dépend de la valeur de `value` dans le composant `Rating` parent
- `Rating` transmet à `Star` la référence de la fonction qui permet de modifier son état (`onStarClicked`)
- au lieu d'exécuter `setState`, dans `Star` on exécute la fonction passée pour `onStarClicked`

react v2.5

```
// dans /components/rating.js
export default class Rating extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value : this.props.value };
  }

  handleStarClicked(starNum) {
    ...
    this.setState(  value :  newValue  );
  }

  render() {
    let stars = new Array(5).fill(0).map(
      (e,i) =>
        <Star
          on = { i < this.state.value }
          onStarClicked = { () => this.handleStarClicked(i) }
          key = {i}
        />
    );
    return(
      <div className="rating">
       ...
```

react v2.5

```
// dans /components/star.js
class Star extends React.Component {
  constructor(props) {
    super(props);
  }

  selectSource() {
    if (this.props.on) { ...
  }

  handleClick() {
    this.props.onStarClicked();
  }

  render() {
    return(
      <img src={this.selectSource()}
        onClick={this.handleClick.bind(this)}
      />
      );
    }
}
```

placer l'état au plus haut (bis)

- ajout du composant `Book`
- ajout du composant `BookList`
  constitué à partir de plusieurs composants `Book`
- l'état est placé au niveau de `BookList`
- les éléments enfants remontent les modifications via des fonctions fournies par une propriété

react v2.6

voir dans le code

```
//dans /components/BookList
let allbooks = this.state.books.map(
  book => <Book
             { ...book }
             onBookChange = {b => this.handleBookChange(b)}
             key = { book.id }
          /> )
```

- utilisation de l'opérateur *spread* « { ...book } » avec JSX pour éviter de répéter toutes les propriétés :
  équivaut à : `title={book.title} author={book.author} ...`
- l'attribut `key` dans la construction de la liste de composants
- transmission par la propriété `onBookChange` de la fonction à appeler par le composant enfant en cas de changement
  noter que c'est bien `handleBookChange` qui appelle `setState()`
- le *rating moyen* qui est affiché (et donc « mis à jour » si besoin)

voir dans le code

dans /component/book.js (pas de « state »)

```
//dans /component/book.js
handleRatingChange(newRating) {
  let book = { ...this.props, rating : newRating};
  this.props.onBookChange(book);
}
```

- utilisation de l'opérateur *spread* dans `...this.props`
  et construction du nouvel objet par "remplacement" de `rating`
- utilisation de la fonction passée en propriété pour transmettre les modifications au parent :
  `this.props.onBookChange(book)`
- création du lien par la propriété `onRatingChange` du composant `Rating`

voir dans le code

dans `/components/Rating.js`

- pas de « `state` »
- utilisation de `this.props.onRatingChange` dans `handleStarClicked`
- mise en place de la propriété `onStarChange` de `Star`

voir dans le code

dans `/components/Star.js`

- voir `onClick` sur l'image et l'utilisation de `this.props.onStarChange`
- suivre le chemin de propagation du "message" en cas de clic sur l'image d'un composant `Star` :
  `onClick` ⇝ `handleClick` ⇝ `this.props.onStarChange` ⇝ `handleStarClicked` ⇝ `this.props.onRatingChange` ⇝ `handleRatingChange` ⇝ `this.props.onBookChange` ⇝ `handleBookChange` ⇝ `setState`

méthode du « *cycle de vie* »

- `componentWillMount` : exécutée **avant** que le composant ne soit inséré dans le DOM `render` n'a pas été appelée
- `componentDidMount` : exécutée **après** que le composant ait été inséré dans le DOM `render` a été appelée
- `componentWillUpdate` : exécutée **avant** que le composant ne soit mis à jour dans le DOM `render` n'a pas encore été ré-appelée
- `componentDidUpdate` : exécutée **après** que le composant ait été mis à jour dans le DOM `render` a déjà été ré-appelée
- `componentWillUnMount` : exécutée juste **avant** que le composant ne soit retiré du DOM

react v2.6

```
// dans /components/booklist.js
export default class BookList extends React.Component {
  constructor(props) {
    super(props);
    this.state = { books :  [] }
  }

  componentWillMount() {
    // fetch allbooks data in some database...
    this.setState({ books : allbooks});
  }

  ...
}

// dans /src/main.js
var bootstrapReact =
  () => ReactDOM.render(
      <BookList />,                  // pas de 'props'
      ...);
```

référence vers l'élément DOM

- il peut être nécessaire de disposer d'une référence dircet vers l'élément DOM
- utilisation de l'attribut `ref` qui définit un callback
  - appelé juste avant `componentDidMount()` ou `componentDidUpdate()`
  - qui prend en paramètre l'élément DOM créé
  - un *pattern* classique est de l'utiliser pour initialiser un attribut du composant qui mémorise la référence :
    `ref = { element => this.myElement = element }`

exemple

react v2.7

```
// dans /components/book.js
highlightTitle() {
  this.titleSpan.style.backgroundColor = "yellow";
  ...
}

render() {
  return(
    <div className="book">
      <span ref={ span => this.titleSpan = span }>
         {this.props.title}
      </span>
      ...
      <button onClick={() => this.highlightTitle.()}>highlight</button>
    </div>
  );
}
```

aller plus loin

- Flux ou Redux pour faciliter la gestion de l'état des composants
  Store et Actions
- React côté serveur
  applications isomorphes
- React Native pour construire des applications mobiles