REACTJS

- Développement Web 420-427-RK
- Par Emeric Depierraz

ReactJS, c'est quoi ?

- ReactJS est une librairie plus qu'un framework.
- Il fonctionne avec Javascript
- Il est la partie ***vue*** du modèle ***MVC***
- Il utilise un DOM virtuel

Votre site de reference : https://reactjs.org/,
et en version française https://fr.reactjs.org/

En général et dans notre projet :

Nous allons découper et déterminer les composants utiles

Nous allons les créer en mode « classe »

Les composants vont déterminer les informations à afficher

À lire (et comprendre) : https://fr.reactjs.org/docs/components-and-props.html

ReactJS est initialement pensé pour des applications monopages. Cependant, rien n'empêche de l'utiliser dans un site multipages :

ce choix vous revient !

Application Web multi-pages	Application Web mono-page
Temps d'attente entre les pages, interactions saccadées, lenteur	Pas d'attente lors du chargement, impression de fluidité, rapidité
En naviguant, on change de page	En naviguant, on change le contenu de l'unique page
Requiert peu d'espace sur l'appareil client	Nécessite de stocker beaucoup d'informations
Facile à mettre à comprendre et à mettre en place, traditionnel	Requiert de comprendre l'architecture React en plus du HTML, CSS, JavaScript
La sécurité est plus dure à gérer (XSS)	Plus facile de se protéger des injections
Requiert beaucoup de mémoire (client)	Requiert peu de mémoire sur le client
Idéal pour les applications avec peu de fonctionnalités, dont la qualité du lien Internet n'est pas indispensable	Idéal pour les applications vastes avec un « plan du site » et nombreuses options dans les menus

JSX ?

Ni du JS ni du HTML, c'est le langage qui va être ***transpilé*** en javascript.

Utilisé par ReactJS, ce langage va permettre d'opérer la jonction entre votre script et votre HTML

Pour comprendre et voir son application :

https://fr.reactjs.org/docs/introducing-jsx.html

DOM virtuel

ReactJS gère un DOM (Document Object Model) virtuel d'une façon bien particulière pour qu'il soit le plus petit possible en mémoire (contrairement au DOM traditionnel) et que les mise-à-jour soient le plus efficaces possible.

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

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Récapitulons ce qui se passe dans cet exemple :

1. On appelle `ReactDOM.render()` avec l'élément `<Welcome name="Sara" />`.
2. React appelle le composant `Welcome` avec comme props `{name: 'Sara'}`.
3. Notre composant `Welcome` retourne un élément `<h1>Bonjour, Sara</h1>` pour résultat.
4. React DOM met à jour efficacement le DOM pour correspondre à `<h1>Bonjour, Sara</h1>`.

Assemblage de composants

Pour qu'un composant ReactJS soit facile à maintenir, il est important de le garder le plus simple possible. Ainsi, une interface sera un assemblage de multiples composants. Les ***props*** sont des propriétés dynamisant un composant.

Version originale

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Première simplification

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Deuxième simplification

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

État local d'un composant

Les composants de la diapositive précédente sont sans état. Si on veut créer un composant « *Horloge* », il faudra un état local.

Ici, l'état initial de l'horloge est basé sur la date et l'heure actuelle. Lorsque la fonction ***render()*** est appelée, c'est l'heure conservée dans l'état qui est affichée.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Cycle de vie d'un composant

L'exemple précédent est incomplet car l'état local ne change pas ! Ajoutons-donc quelques méthodes à la classe.

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}

componentWillUnmount() {
  clearInterval(this.timerID);
}

tick() {
  this.setState({
    date: new Date()
  });
}
```

componentDidMount() est appelée au moment où le composant est ajouté au DOM. Elle appelle ***tick()*** à toutes les 1000 millisecondes (à chaque seconde). ***componentDidMount()*** est un bon endroit pour charger des données d'une BD ou d'un service Web.

tick() change l'état local du composant, ici en modifiant la date et l'heure actuelle. ***render()*** sera donc réappelée par ReactJS.

componentWillUnmount() est appelée lors que le composant n'est plus utilisé dans le DOM.

Gérer les événements

Par défaut, une fonction JavaScript créer une nouvelle valeur pour la variable ***this***.
Pour employer le ***this*** de la classe du composant, il faut explicitement l'attacher avec la fonction ***bind()*.**

Le composant montre un bouton ON / OFF dont l'état change lorsqu'on le clique dessus avec la souris.

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // Cette liaison est nécéssaire afin de permettre
    // l'utilisation de `this` dans la fonction de rappel.
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

Fonction fléchée

En JavaScript, les fonctions fléchées ont la particularité de ne pas affecter de valeur à la variable ***this***.

Ainsi, lorsqu'on l'emploie dans un composant, il n'est pas nécessaire de lui attacher ***this*** avec la méthode ***bind()***.

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

        this.state = { isToggleOn: true };
    }

    handleClick = () => {
        this.setState(state => ({
            isToggleOn : !state.isToggleOn
        }));
    }

    render(){
        return (
            <button onClick={this.handleClick}>
                {this.state.isToggleOn ? 'ON' : 'OFF'}
            </button>
        );
    }
}
```

Par essence, la fonction fléchée est anonyme (sans nom). C'est pourquoi on l'affecte à une variable pour l'employer.

https://www.w3schools.com/js/js_arrow_function.asp