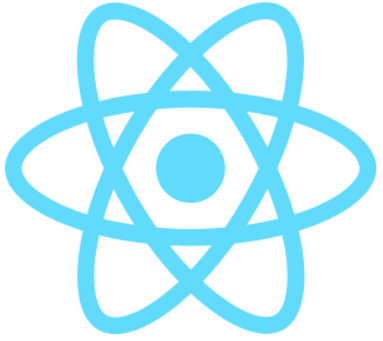
React



Première partie

Quelques jalons dans l'histoire des technologies du Web

- **1993 HTML** : statique ou généré dynamiquement côté serveur
- **1995 JavaScript** : ajout d'interactions côté client
- **1996 CSS** : séparation structure / présentation
- **2004 AJAX** (requêtes HTTP depuis JS) ➟ pages "dynamiques"
- **2006 jQuery** : bibliothèque surcouche au DOM, manipulation des pages facilitée, plugins
- **2007 iPhone & Android** : émergence d'un vrai web mobile
- **2010 Node.js + npm** : JavaScript côté serveur, dépendances, explosion de l'écosystème
  ➟ maturité (ou du moins pertinence) de JS comme environnement de développement
- **2014 HTML5** : version modernisée, nouveaux éléments sémantiques
- **2015 ECMAScript 6 (ES6)** : `let/const`, `modules`, classes, promesses, etc.

Et tout du long, évolution des navigateurs et des standards.

SPA : *Simple Page Application*

Modèle de développement apparu avec AJAX
UX similaire aux applications natives : pas de rechargement de page

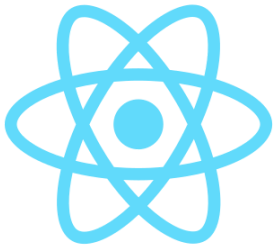
- Page HTML squelette unique + code applicatif entier en JavaScript côté client
- Interrogation d'une API
- Modification dynamique du DOM en JavaScript
- Interception des navigations d'une url à l'autre (routing client)

➠ Développement se complefixiant très vite, notamment en terme de cohérence de l'interface

Frameworks JavaScript MVC (Backbone.js, Ember, AngularJS, etc.)

- Architecture **M**odel-**V**iew-**C**ontroller
- Certains perçus comme trop dirigistes, peu performants, peu modulaires, etc.

React



Introduit par Facebook en 2014

Dédié au développement de SPA

Présenté comme une **bibliothèque**, pas un framework

Se focalise sur la partie View de l'architecture MVC

Principes :

- faciliter l'écriture de **composants** réutilisables
- garantir la **cohérence** de l'interface à chaque instant
- **modifier le DOM** de manière **performante et transparente**

Documentation officielle (bien faite et entièrement traduite)

Paradigmes de programmation

Programmation impérative

Description des étapes à suivre

Ex : "Quand le nom d'utilisateur change, le mettre à jour dans le header et dans la liste des messages"

V.S.

Programmation déclarative

Description du résultat voulu

Ex : "Dans le header et dans la liste des messages, afficher le nom de l'utilisateur courant"

en React, réalisé par ↓

Programmation réactive

Modification d'une source de données (état) ➠ propagation aux éléments qui en dépendent et mise-à-jour

Exemple élémentaire

Document HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <script src="example.js"></script>
  </head>
  <body>
    <h1>Exemple React</h1>
    <p>Éléments statiques</p>

    <!-- élément prévu pour React -->
    <div id="root"></div>
  </body>
</html>
```

Code Javascript

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

// composant
function Hello(props) {
  return <h1>Hello {props.name} !</h1>
}

// "accrochage" d'un composant React
// au DOM de la page
ReactDOM.render(
  <Hello name="Alice" />,
  document.getElementById('root')
);
```

Une fois l'appel à `ReactDOM.render(…)` effectué, tout ce qui est à l'intérieur va être géré par React.

Syntaxe JSX

Syntaxe proche du HTML, directement dans le code JavaScript :

- les composants sont assimilés à des éléments HTML
- les *props* sont assimilés à des attributs

Intérêts : rapidité d'écriture, encapsulation, rapprochement des concepts…

> ⚠️ Le JSX n'est pas du JavaScript valide ➟ repose sur une **transpilation**.
> En développement JS moderne, on travaille souvent dans un environnement transpilé (Babel).

Ce qu'on écrit

```jsx
const message = 'Hello, world!';
const element = <h1 className="greeting">
  {message}
</h1>;
```

Transpilé en

```jsx
const message = 'Hello, world!';
const element = React.createElement(
  "h1",
  { className: "greeting" },
  message
);
```

Syntaxe JSX

On mélange la notation balises/attributs avec des **expressions JavaScript entre accolades** `{ … }`

- Variables, valeurs calculées, appels de fonctions
- Structures de contrôles élémentaires (expressions booléennes et boucles `map`)

```
return (
  <div className={isOpened ? 'opened' : 'closed'}>
    <h3>Liste des utilisateurs</h3>
    <ul>
      {/* boucle */}
      {props.users.map(user =>
        <li key={user.id}>{formatUserName(user)}</li>
      )}
    </ul>
    {/* affichage conditionnel */}
    {errorMsg && <Error msg={errorMsg} />}
  </div>
)
```

Syntaxe JSX : quelques règles

- `false`, `null` et `undefined` ne sont pas rendus dans le DOM
- Toute balise doit être explicitement fermée (`<tag></tag>` ou `<tag />`)
- La casse de la première lettre des balises est importante :
  - **minuscule = balise HTML** (ex : `<h2>Introduction</h2>`)
  - **majuscule = composant React** (ex : `<UserName />`)
- Certains noms d'attributs HTML sont renommés :
  - `camelCase` : `onclick` → `onClick`, `tabindex` → `tabIndex`, etc.
  - attributs en conflit avec des mots-clés JS : `class` → `className`, `for` → `htmlFor`, etc.
- Les commentaires s'écrivent par blocs, aussi entre accolades `{/* … */}`

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

Composants et *props*

Une application React est constituée d'un arbre d'instances de composants

- Chaque composant peut **rendre** du HTML et/ou d'autres composants
- Les données se propagent dans l'arbre via les ***props*** (pour "*properties*"), **de haut en bas**
- Un composant est **re-rendu** automatiquement quand il reçoit de **nouvelles valeurs de *props***

https://fr.reactjs.org/docs/components-and-props.html

```
// Un composant prend en argument un objet "props"
function Hello(props) {
  // ce composant reçoit la prop 'name'
  return <h1>Hello, {props.name}</h1>;
}

// La syntaxe de destructuration d'objet
// fait mieux apparaître les props en entrée
function App({ users }) {
  return <div>
    {users.map(user =>
      <Hello name={user.name} key={user.id} />)}
  </div>;
}

const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Charlie' }
];

ReactDOM.render(
  // passage de la prop 'users' en JSX
  <App users={users}/>,
  document.getElementById('root')
);
```

Props

On peut passer ce qu'on veut en *props* :

- nombres, chaînes, booléens, objets, tableaux, etc.
- fonctions *callbacks* (gestionnaires d'événements notamment, abordés plus loin)
- du JSX, une instance de composant, un autre composant React

Une *prop* spéciale : `children` ➟ permet de passer sa valeur en contenu de la balise JSX.

```
const Details = ({ title, children }) => {
  return <details>
    <summary>{title}</summary>
    {children}
  </details>;
}

const App = () => {
  return <Details title="Conditions d'utilisation">
    <p>Article 1. : blabla</p>
    <p>Article 2. : …</p>
  </Details>
}
```

Virtual DOM

Modifier le contenu d'une page dynamiquement nécessite de passer par l'API du **DOM**.

Par exemple :

```
const myMessage = document.createElement('div');
myMessage.className = 'message';
myMessage.innerText = 'Lorem Ipsum';

const messageList = document.getElementById('messageList');
messageList.appendChild(myMessage);
```

React introduit une notion de **"DOM virtuel"** pour :

- **abstraire ces appels** à travers la syntaxe JSX
- **optimiser les accès au DOM réel** en appliquant des mises à jour minimales

https://fr.reactjs.org/docs/reconciliation.html

Deux syntaxes de déclaration de composant

Classe

Classe ES6 héritant de `React.Component`

```
import { Component } from "react";

class Hello extends Component {
  // seule méthode obligatoire
  render() {
    return <h1>Hello {this.props.name} !</h1>;
  }
}
```

Approche classique, orientée objet

Fonction

Fonction qui retourne du JSX

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

```
const MyComponent = ({ name }) => {
  return <h1>Hello {name} !</h1>;
}
```

Approche moderne favorisée par React

- Plus proche du modèle conceptuel
- Syntaxe plus légère
- Contrôle plus fin des conditions de rendu

State

Matérialise l'**état interne** d'un composant

- Privé et encapsulé ➟ on passe par les *props* pour propager ce *state* aux enfants
- Persistant entre deux rendus
- Quand le *state* d'un composant change, le composant est automatiquement re-rendu

⚠️ Une des grandes questions d'une appli React est de savoir quel composant doit gérer quel *state*.

- Deux composants qui partagent un état = descendants du même composant *stateful*
- Un composant *stateless* sera plus réutilisable (pas de "surprise")

https://fr.reactjs.org/docs/state-and-lifecycle.html

Exemple de *state* : une horloge

Classe

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

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

  // méthode du cycle de vie
  // on verra ça en détails en 2e partie
  componentDidMount() {
    this.timerID = setInterval(
      () => { this.setState({ date: new Date() }); },
      1000
    );
  }

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

  render() {
    return <div>Il est {this.state.date.toLocaleTimeString()}.</div>
  }
}
```

Fonction

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

const Clock = () => {
  // useState() prend la valeur initiale et retourne
  // [valeurCourante, fonctionDeModification]
  const [date, setDate] = useState(new Date());

  // on verra ça en détails en 2e partie
  useEffect(() => {
    const timerID = setInterval(
      () => { setDate(new Date()); },
      1000
    );

    return () => {
      clearInterval(timerID);
    }
  }, []);

  return <div>Il est {date.toLocaleTimeString()}.</div>
}
```

Interactions utilisateurs

On attache des *callbacks* aux différents types d'événements via les attributs JSX dédiés : `onClick`, `onKeyPress`, `onFocus`, `onSubmit`, … (version `camelCase` de ceux du HTML)

```
function EventExample() {
  function handleClickButton() {
    console.log('Bouton cliqué !')
  }

  function handleInputChange(event) {
    console.log("Valeur de l'input : ", event.target.value);
  }

  return <>
    <button type="button" onClick={handleClickButton}>Click!</button>
    <input type="text" onChange={handleInputChange} />
  </>;
}
```

Interactivité = *state* + *events*

On crée des composants interactifs en modifiant leur *state* via des événements utilisateurs.

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

function Clicker() {
  const [nbClicks, setNbClicks] = useState(0);

  const incrementNbClicks = () => {
    setNbClicks(nbClicks + 1);
  }

  return <div>
     <p>Nombre de clics : {nbClicks}</p>
    <button type="button" onClick={incrementNbClicks}>Click!</button>
  </div>;
}
```

Passage de callbacks en *props*

```
// Composant stateless, prend le callback onChangeTheme en prop
function ThemeSwitch({ currentTheme, onChangeTheme }) {
  return <div>
    <button
      disabled={currentTheme === 'dark'}
      onClick={() => { onChangeTheme('dark'); }}>Dark</button>
    <button
      disabled={currentTheme === 'light'}
      onClick={() => { onChangeTheme('light'); }}>Light</button>
  </div>;
}

function App() {
  const [theme, setTheme] = useState('light'); // L'état est maintenu ici

  return <div>
    <ThemeSwitch currentTheme={theme} onChangeTheme={setTheme} />
    <Content theme={theme} />
  </div>;
}
```

→ Permet la remontée d'informations aux parents en respectant le flux de propagation descendant.

Résumé

- Une application web React est constituée d'un **arbre de composants**
- Les composants **rendent des fragments de HTML**, gérés dans le DOM par React
- L'**état de l'application est maintenu dans le *state*** de certains composants
- Les **interactions utilisateur** modifient ces *states*
- Les changements de *state* sont **répercutés via les *props* de haut en bas** dans l'arbre
- Chaque **changement de *state*/*props* entraîne le re-rendu** des composants qui en dépendent

Liens utiles

- Site officiel React
- React Developer Tools
  - Firefox
  - Chrome, Chromium, Edge
- Create React App
  Pour monter un environnement de développement avec un outillage pré-installé

Suite dans la deuxième partie…