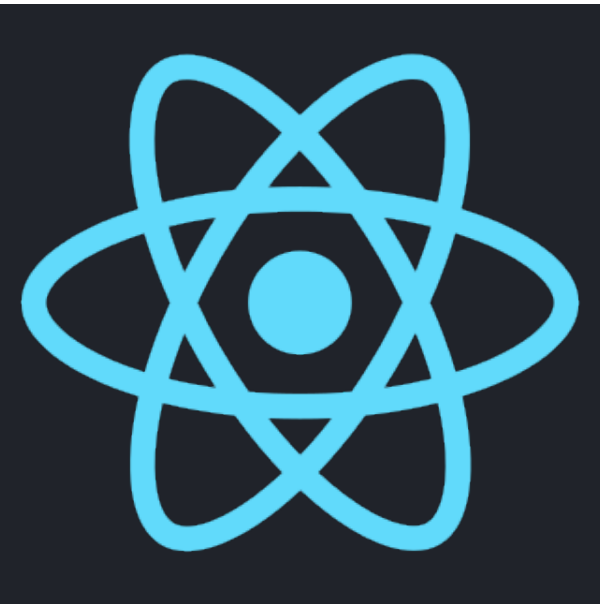
Technologies Front-End

Framework JS



Module « React »

Made by Valentin AMATHIEU

Prérequis

Compétences

» *HTML*

» *CSS*

» *JavaScript (ES2015+)*

» Node / NPM

Logiciels

- Node / NPM
  - Installation Mac OS X / Linux : https://nodejs.org/en/
  - Installation Windows : https://docs.microsoft.com/fr-fr/windows/nodejs/setup-on-windows
- Même si l'installation classique de Node fonctionne bien sur tous les OS mainstream, il est recommandé d'installer NVM afin de pouvoir **gérer plusieurs versions de Node** sur la même session.
- NVM permet au développeur de **switcher à tout moment la version de Node** par défaut, lui épargnant ainsi toutes les **erreurs de compatibilité de version**, lors des phases de **compilation** !

Question Time

?

Présentation

React

» Librairie *JavaScript*
» Construction d'interface utilisation (composants UI)
» Virtual DOM
» Open source
» Créé par Facebook
» Maintenu par Facebook et la communauté
  » Applications mobiles (avec React Native)
» Ecosystème
  » Redux (state management system)
  » React Router (client-side routing)
  » React Native (native mobile app)

Historique

» 2011 : Création de React (anciennement appelé « FaxJS ») par Jordan Walke, pour le fil d'actualité Facebook

» 2012 : Utilisation de React dans Instagram

» 2013 (Mai): 1ère release publique de React (v0.3)

» 2015 (Mars) : 1ère release publique de React Native

» 2016 (Avril) : Sortie de React 15.0 (nouveau système de versionning)

» 2017 (Avril) : Annonce de React Fiber, le nouvel algorithme du cœur de React

» 2017 (Septembre) : Sortie de React 16.0

» 2019 (Février) : Sortie de React 16.8 et introduction des Hooks

» 2020 (Août) : Annonce de React 17 !

Philosophie

» Programmation déclarative :

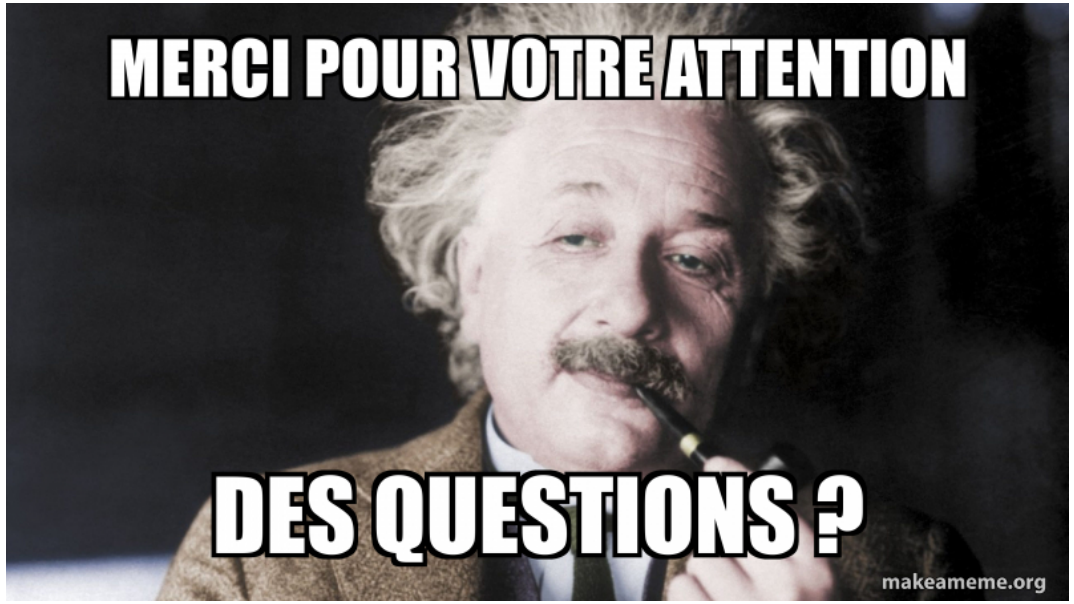
» La programmation déclarative est un paradigme de programmation qui consiste à créer des applications sur la **base de composants logiciels indépendants du contexte** et ne comportant aucun état interne…
Lien vers la page Wikipédia

» Programmation orienté composant :

» La programmation orientée composant (POC) consiste à utiliser une **approche modulaire de l'architecture d'un projet informatique**, ce qui permet d'assurer au logiciel une **meilleure lisibilité et une meilleure maintenance**. Les développeurs, au lieu de créer un exécutable monolithique, se servent de **briques réutilisables**.
Lien vers la page Wikipédia

Question Time

?



Utiliser React

Une librairie avant tout

» React est une **librairie** *JavaScript*.

» Il n'est donc **pas obligatoire** de construire une application entièrement en React !

» On peut l'utiliser pour des **parties d'un projet**.

» … Ou bien construire un **projet entièrement basé sur React** !

» Pour cette dernière solution, vous pouvez :

  » Soit **définir et créer votre propre architecture**, mais il sera à votre charge de choisir les **outils de build**, ainsi que de faire les **configurations**.

  » Soit utiliser des **outils qui construiront la base de vos projets**, vous permettant ainsi de vous concentrer tout de suite sur la **partie métier** !

» Voyons déjà comment inclure React dans un **projet déjà existant**.

Dans un projet existant

» Utiliser React dans un projet existant, consiste en 3 étapes.

» Voir démo « use-react/existing-project »

» 1ère étape : Ajouter les scripts contenant la librairie :

  » Une manière simple de le faire consisterait à ajouter des balises scripts directement dans le HTML.

  » react.js : Ce script contient le **cœur de la librairie**, avec tous le code nécessaire à la **création des composants de votre application** !

  » react-dom.js : Ce script contient entre autre le module qui va vous permettre de **lier les composants au DOM de votre page HTML** !

```
<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>
<script></script> <!-- Script utilisant React... -->
```

Dans un projet existant

» 2ème étape : Ajouter un élément dans le DOM qui contiendra les éléments produits par React

```
<h1>Bienvenue dans mon site !</h1>
<hr />
<div>Je suis fait en HTML classique</div>
<hr />
<div id="react-app"></div>  <!-- Cette div contiendra mon application React -->
<hr />
<div>Je suis fait en HTML classique</div>
```

Dans un projet existant

» 3ème étape : Créer un composant React et le lier au DOM

```
"use strict";

const e = React.createElement;

function MyFirstComponent() {
    const message = "Je suis fait en React ! Cliquer moi ! :)";
    return e(
        "div",
        { onClick: () => alert("Vous avez cliqué sur le texte !") },
        message
    );
}

const domContainer = document.querySelector("#react-app");
ReactDOM.render(e(MyFirstComponent), domContainer);
```

… ou dans un nouveau projet !

» Il existe 2 manières de créer un projet entièrement basé sur React

» La manière « **from scratch** » où il est à votre charge de :

  » Définir l'arborescence de votre projet

  » Choisir les outils de builds (module bundlers, task runners, linters, …)

  » Configurer ces outils

  » Coder votre application …!

» Cette méthode a le mérite de vous permettre de **contrôler entièrement le workflow** de votre projet, ainsi que la granularité du paramétrage.

» … Mais elle peut être couteuse en **temps ET en ressource** car cela revient à devoir maîtriser tous ces outils pour garantir un workflow de développement efficace.

» L'autre méthode consiste à utiliser des **outils pré-faits** s'occupant de tous ces aspects pour vous ! Il ne vous reste plus qu'à coder la partie métier ! :)

… ou dans un nouveau projet !

» Concernant la 1ère méthode, vous pouvez vous baser sur ce tutoriel from scratch, vous permettant de réaliser votre propre workflow de développement.

» Voir démo « use-react/new-project/custom-toolchain »

» Nous allons voir ici la 2ème méthode, celle de l'utilisation d'une **Toolchain** !

» Il existe différentes Toolchains, chacune proposant de créer des applications React avec des **architectures spécifiques**.

» Pour ce cours, nous allons utiliser la Toolchain la plus connue et sûrement la plus utilisé de l'écosystème React : Create React App

» Et bonne nouvelle, npm étant maintenant installé sur votre machine, pas besoin d'installer cette Toolchain ! En effet, la commande npx reconnaît automatiquement ce package !

… ou dans un nouveau projet !

» Voici la commande pour créer un nouveau projet React avec cet outil.

```
> npx create-react-app new-project
```

» Pour tester votre application, il suffit de se rendre dans le dossier et de lancer le script « start » avec npm

```
> cd new-project
> npm start
```

Question Time

?

La philosophie React

Dynamiser une page web

- Le *JavaScript* permet au développeur de **dynamiser le cycle de vie d'une page web**, via l'API DOM.
- Cette API fournit un ensemble d'opérations, visant à manipuler la structure du *HTML* et le **comportement** des différents éléments de la page :
  - Créer un nouvel élément
  - Modifier les attributs d'un élément
  - Associer des gestionnaires d'événement à un élément
  - Supprimer un élément
  - …
- Il existe aujourd'hui 2 principaux **paradigmes de programmation**, permettant de dynamiser une page web :
  - La **programmation impérative**, où le développeur doit utiliser l'API DOM de manière explicite
  - La **programmation déclarative**, où les notions de DOM sont cachées au développeur et gérées automatiquement par des outils (APIs UI / frameworks) faits spécialement pour.

La programmation impérative

» Quand on utilise l'API DOM directement en JavaScript standard (VanillaJS) ou bien indirectement via une API UI (jQuery), on est en **programmation impérative**.

» Ce paradigme consiste à exécuter, de manière **linéaire** et **séquentielle**, des instructions en JavaScript, décrivant précisément quelles actions on souhaite effectuer sur le DOM.

» Prenons un exemple avec une version VanillaJS et une jQuery :

```
<p>Je suis un paragraphe</p>
<script>
    const par = document.querySelector("p");
    par.style.color = "red";
    par.onclick = function () {
        par.style.color = "green";
    };
</script>
```

```
<p>Je suis un paragraphe</p>
<script>
    const par = $("p");
    par.css("color", "red");
    par.on("click", function () {
        par.css("color", "green");
    });
</script>
```

La programmation déclarative

» L'autre approche, dites « **déclarative** » consiste, au moyen de librairies spécialisées, à **définir la structure** d'une page web, à partir d'une **arborescence d'élément d'interface**.

» Le *HTML* est lui-même un langage basé sur ce paradigme.

» Quand on crée une page web en *HTML*, on crée une arborescence de balises, chacune représentant un élément de l'interface.

» L'idée sera la même du côté d'une librairie *JavaScript* spécialisée !

» Vous construirez une page web en définissant une hiérarchie (ou arborescence) **d'objets *JavaScript***, chacun étant chargé de représenter un élément de l'interface, ainsi que ses comportements.

Développer en React

» React est l'une de ces librairies !

» On appelle « **Éléments React** » les différents éléments de l'interface.

» On appelle « **Composant React** », l'ensemble constitué d'un élément React et des gestionnaires d'événements qui lui sont associés.

» La raison d'être d'un composant est de **fournir** (on parle aussi de **rendu**) des éléments React, entièrement dynamisés.

» En effet, React est aussi basé sur le **paradigme orienté composant**.

» Cela signifie que vous pouvez structurer votre application web en créant une hiérarchie de composant ! On obtient donc un **arbre de composants**.

» La librairie propose enfin un moyen de lier cet arbre de composants au DOM et se charge alors de **synchroniser** le tout.

Le DOM virtuel

» Afin de construire des interfaces utilisateurs (UIs), React crée une représentation objet, similaire au DOM classique (appelé DOM réel ou Browser DOM), mais bien plus rapide : le DOM virtuel (Virtual DOM).

» Le DOM virtuel est construit automatiquement par React, à partir des éléments connectés au DOM réel et est synchronisé avec ce dernier.

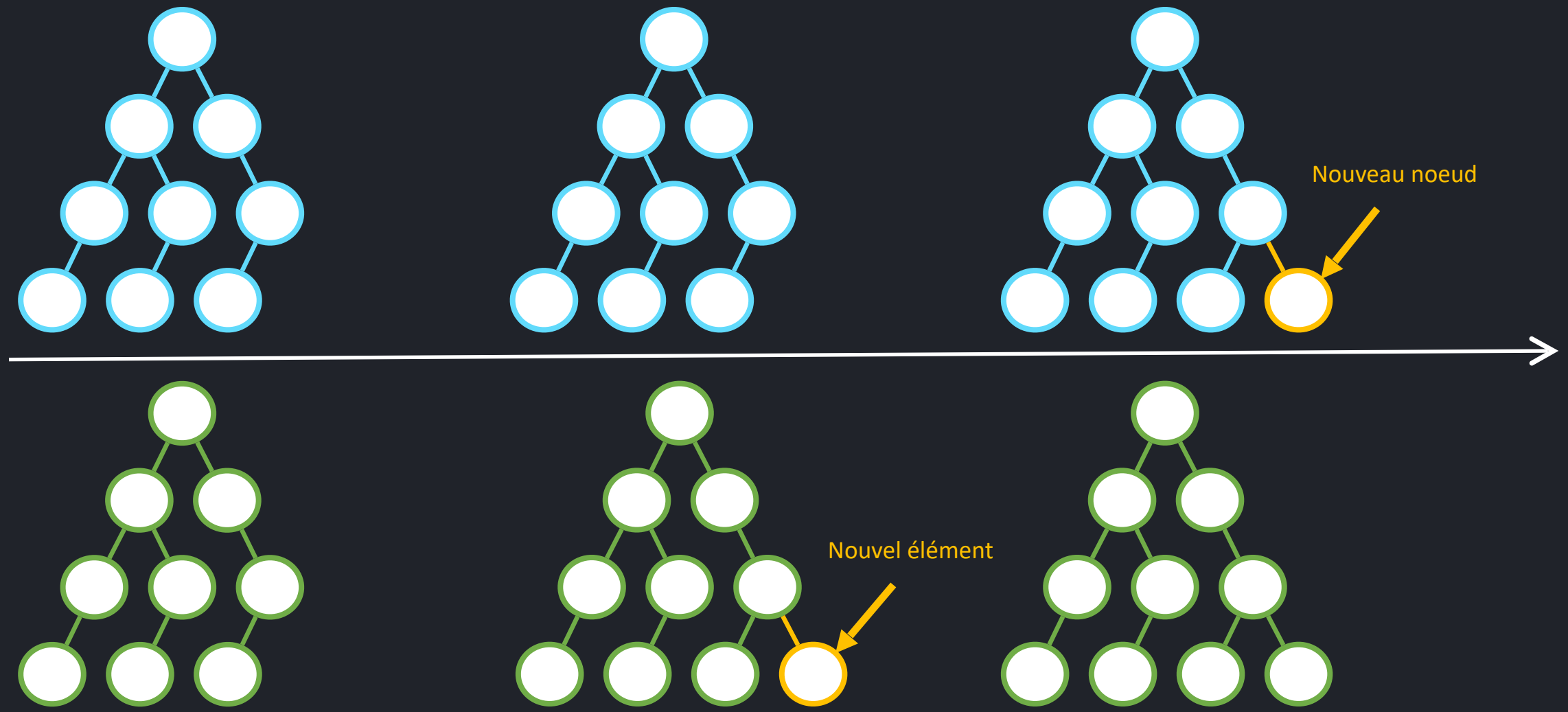
» Ainsi, tout changement d'état au niveau d'un des éléments du DOM virtuel, sera **automatiquement mis à jour** dans le DOM réel. Il n'est ainsi plus à la charge du développeur de procéder manuellement à ces étapes !

» On appelle ce processus la « **réconciliation** ».

La réconciliation

- La réconciliation consiste en l'exécution (par React) d'un algorithme, dont l'objectif est de considérer les 2 DOMs (virtuel et réel) et, à partir de leur racine respective, de comparer chaque nœud un à un.
- Les principales règles sont les suivantes :
  - Si 2 nœuds sont de type différent (balises différentes, composants différents), alors le nœud du DOM réel (ainsi que ses descendants) est détruit, puis entièrement reconstruit sur la base du nœud correspondant dans le DOM virtuel.
  - Si 2 nœuds sont de même type, l'algorithme considère alors leurs attributs respectifs. Le nœud du DOM réel sera conservé, mais toute différence d'attribut du nœud du DOM virtuel, lui sera reportée (mise à jour).
- Au fur et à mesure de votre découverte de React, quelques nouvelles règles vous seront expliquées, notamment concernant les spécificités liées aux Composants, ainsi qu'aux listes d'élément…

La réconciliation



Question Time

?

Les éléments React

Créer un élément React

» Voyons tout d'abord comment créer un élément React.

» Un élément React est, à l'instar d'un objet du DOM, la représentation en objet *JavaScript* d'une balise HTML, avec ses éventuels attributs et enfants.

» Pour créer un élément, on utilise la méthode « createElement() », qui nous est fournie par le module « react ».

```
import React from "react";

const el = React.createElement(
    "div", // Type de la balise
    { title: "Je suis une div" }, // Liste des propriétés
    "Contenu texte de la div" // Élément enfant ou null
);
```

Liaison avec le DOM

» Maintenant que vous avez un élément React, il faut le lier au DOM du navigateur.

» Il faut pour cela utiliser la méthode « render() » du module « react-dom ».

» Cette méthode prend en arguments :

- » Un élément React ou un composant renvoyant un élément
- » Un nœud du DOM dans lequel injecter cet élément

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

const el = React.createElement("div", { title: "Je suis une div" }, "Contenu texte de la div");
const appContainer = document.getElementById("root");

ReactDOM.render(el, appContainer);
```

Une nouvelle syntaxe !

» Nous avons vu comment créer un élément React.

» Cependant, une application web est (dans la majorité des cas) **composée de plus d'une balise**.

» Or, comme vous pouvez l'anticiper, créer des éléments React via la méthode createElement() et les composer hiérarchiquement peut devenir très vite **compliqué**, voir **illisible** !

```
React.createElement("div", { title: "Je suis une div" },
  React.createElement("p", { class: "red" },
    React.createElement("span", null, "...")));
```

» Mais rassurez vous ! Il existe aujourd'hui une nouvelle manière de faire : Le ***JSX*** !

Le JSX : syntaxe déclarative

» Une nouvelle syntaxe a été proposé afin de construire des éléments React, avec un concept que vous connaissez déjà…. Celui des **langages de balisage** !

» Un peu à la manière du *HTML* (ou du *XML* par extension), vous allez pouvoir composer vos hiérarchies d'éléments, en mettant directement des balises dans votre JavaScript « <balise>…</balise> »

```
const el = (
  <h1>Bienvenue sur mon <strong>super</strong> site</h1>
);
```

» On appelle cette syntaxe le *JSX* et l'élément créé sera ainsi appelé « **élément *JSX*** ».

» Un **élément *JSX* est une expression**. Il peut donc être manipulé en tant que tel !

Le JSX : règle de la balise parente

» Attention ! Si vous souhaitez créer un élément React, représentant plusieurs balises voisines, il est OBLIGATOIRE de définir une balise parente commune !

```
// Code non valide !
const el = (
    <h1>Bienvenue sur mon <strong>super</strong> site</h1>
    <p>J'espère que votre navigation sera agréable !</p>
)

// Code valide !
const el = (
    <div>
        <h1>Bienvenue sur mon <strong>super</strong> site</h1>
        <p>J'espère que votre navigation sera agréable !</p>
    </div>
)
```

Babel et la transpilation

» Bien évidemment, cette syntaxe n'est pas connue des navigateurs !

» Néanmoins, comme on le sait, il existe des **outils de transpilation** (comme Babel), permettant, à l'aide de plugins, de **transformer des syntaxes** !

» L'exemple le plus connu est certainement la traduction du *TypeScript* en *JavaScript*.

» Et bien il existe un plugin, notamment présent dans Babel, permettant de transformer la syntaxe du *JSX* en des éléments React !

» Voici un exemple de cette traduction :

```
const el = (
  <h1 title="Titre">
    Bienvenue sur mon site
  </h1>
);
```

```
const el = React.createElement(
  'h1',
  { title: 'Titre' },
  'Bienvenue sur mon site'
);
```

Babel et la transpilation

» Attention cependant ! L'utilisation du *JSX* ne vous congédie pas de l'import du module « react » dans votre script !

» En effet, Babel se charge de transpiler le *JSX* en une fonction issue de ce module !

» Il faut donc que ce dernier soit présent (importé), même si on ne s'en sert pas en apparence !

```
import React from "react"

const el = (
  <h1 title="Titre">
    Bienvenue sur mon site
  </h1>
);
```

```
import React from "react"

const el = React.createElement(
  'h1',
  { title: 'Titre' },
  'Bienvenue sur mon site'
);
```

Expression JavaScript

» Si vous désirez introduire dans votre *JSX* des **expressions JavaScript** (calculs, variables, appels de fonction, ...), il vous suffit de les placer entre **accolades** « {} ».

» **L'expression sera évaluée** et la valeur résultante sera **contrôlée** (des caractères sont échappés pour éviter les attaques XSS), puis **insérée** dans l'Élément *JSX*.

» Attention cependant ! **Seules les expressions JavaScript sont autorisées** ! Utiliser tout autre type de structure (déclaration de variable, if, for, ...) entraînera une **erreur** !

```
// Code valide
const age = 18;
const el = <div>J'ai { age } ans</div>

// Code non valide
const el = <div>J'ai { const age = 18; age } ans</div>
```

Les attributs JSX : camelCase

» A l'instar du *HTML*, on peut préciser des **attributs** aux éléments *JSX*.

» On peut aussi préciser ou non des **valeurs** à ces attributs (si non, la valeur par défaut est « true »).

» Attention cependant ! Le *JSX* étant une syntaxe utilisable au niveau du *JavaScript*, les **noms des attributs** doivent en fait correspondre aux **noms des propriétés DOM** associées !

» On utilisera ainsi la convention « **camelCase** » pour les noms !

```
<p tabIndex="0">Ce paragraphe est accessible via tab</p> // et pas tab-index="0"
```

Les attributs JSX : mots-clés proscrits

» De plus, tous les **mots-clés réservés au langage *JavaScript* sont proscrits** en tant que nom de propriété !

» Ainsi, « class » deviendra « className », « for » deviendra « htmlFor », …

```
<p className="red">Ce paragraphe à la classe "red"</p> // et pas class="red"
```

Valeurs des attributs JSX

» Pour préciser une valeur à un attribut *JSX*, vous pouvez utiliser une **chaîne de caractères** avec une valeur contenue entre **guillemets**.

```
<p className="red">Ce paragraphe à la classe "red"</p>
```

» Mais vous pouvez aussi préciser cette valeur en utilisant la notation des **accolades**, qui accepte n'importe quelle expression *JavaScript* évaluable en une

```
<p className={"red"}>Ce paragraphe à la classe "red"</p>
```

Spread des attributs JSX

» Il est possible d'associer toutes les **paires de clé-valeur** d'un objet *JavaScript*, en tant **qu'attributs** *JSX* à un élément !

» Pour ce faire, il faut utiliser la syntaxe *ES2015* du « **spread** ».

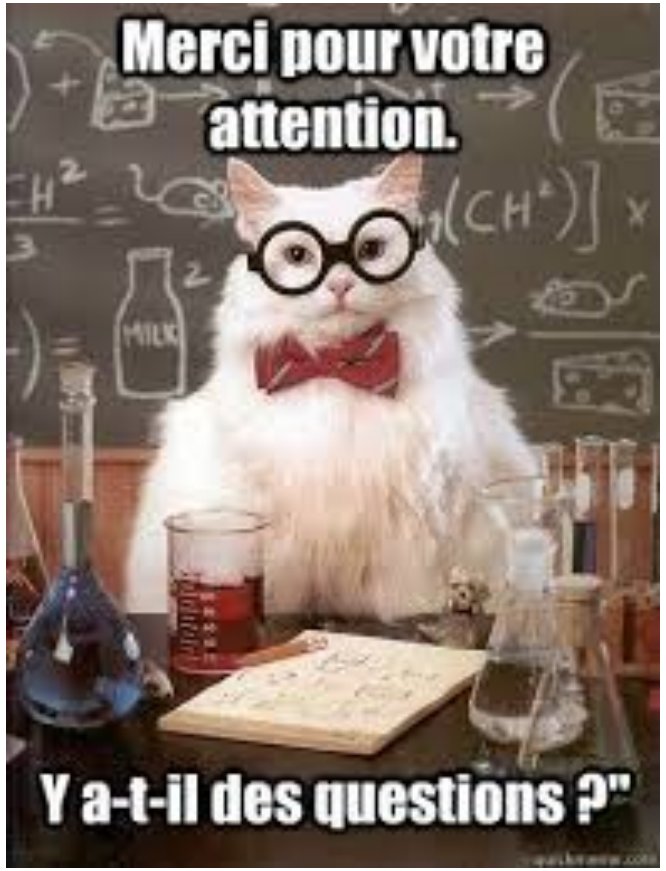
» Et rien n'empêche de **combiner** cette syntaxe avec celle attributs classiques.

```
// Ceci...
const attributes = {
    title: "Je suis un paragraphe",
    className: "red"
}
const el = <p { ...attributes } tabIndex="1">Bienvenue sur mon site</p>

// est la même chose que ceci...
const el = <p title="Je suis un paragraphe" className="red" tabIndex="1">Bienvenue sur mon site</p>
```

Question Time

?



Les composants

Problèmes des éléments seuls

» Nous avons vu comment créer des éléments React.

» Cependant, ces éléments sont actuellement dénués des notions « **d'état** » et de « **comportements** » qui permettent de **dynamiser** une application.

» De plus, ces éléments ne sont **pas réutilisables**, à plusieurs endroits de notre code, ce qui nous fait perdre en modularité.

» Pour palier à ces problèmes, il va nous falloir **encapsuler** nos éléments, de manière à pouvoir les rendre **dynamiques** et **réutilisables**.

» Nous allons donc voir maintenant comment faire des « **Composants** » !

Qu’est-ce qu’un composant ?

- Un composant est simplement une **structure**, encapsulant un **élément React**, ainsi que ses **comportements** et capable de **fournir** cet élément.
- Il existe 2 manières de créer un composant :
  - En créant une **fonction**, qui retourne un élément *JSX*.
  - En créant une **classe *ES2015***, qui implémente une méthode « render() », chargée de retourner un élément *JSX*.
- La **1ère lettre** du nom d’un composant (nom de la fonction ou nom de la classe) doit **OBLIGATOIREMENT** être en **majuscule** !
- En effet, les éléments *JSX* commençant par une lettre minuscule sont considérés par React comme étant des éléments *HTML* standards.

Intérêts des composants ?

» L'approche composant nous permet de **décomposer nos interfaces utilisateurs** en différentes unités **visuelles** et **logiques**.

» Ce découpage apporte plusieurs avantages comme une **meilleur lisibilité du code**, ainsi que la possibilité de **réutiliser ces composants** dans différentes parties de notre application !

» Au niveau du JSX, on pourra en effet utiliser un composant, en créant une **balise portant le nom de ce composant** !

» Ces balises pourront être **simples** <MonComposant/> ou **doubles** <MonComposant></MonComponsant>, au choix.

» Nous verrons plus tard l'intérêt d'utiliser des balises doubles pour un composant.

1 composant = 1 module JS

» Vous allez apprendre comment créer un composant.

» Comme une application React peut être constituée de plusieurs composants, il est vivement recommandé de créer un **module *JavaScript*** pour chacun de vos composants.

» N'oubliez pas **d'exporter** les composants (la fonction ou la classe) dans leur module, afin de les rendre accessibles ailleurs dans votre projet.

» La méthode d'export recommandée est celle de **l'export par défaut**.

» En effet, la plupart du temps, on ne souhaitera exporter qu'un composant, depuis son fichier.

Composant en mode fonction

» Pour créer un composant en « mode fonction », il suffit de créer une fonction retournant un élément JSX.

```
import React from "react"

export default function Navigation() {


    return (<ul>
        <li>Accueil</li>
        <li>CV</li>
        <li>Portfolio</li>
    </ul>);
}
```

Composant en mode classe

» Pour créer un composant en « mode classe », il suffit de créer une classe, héritant de la classe « Component » du module « react » et implémentant la méthode « render() ».

```
import React from "react"

export default class WelcomeMessage extends React.Component {


    render() {

        return (
            <p>Bienvenue sur mon site !</p>
        );
    }
}
```

Utiliser en tant qu'élément JSX

» Vous souhaitez faire **référence** à votre composant dans un élément JSX ?

» Rien de plus simple, il suffit de **créer une balise** dont le nom est celui du composant !

```
import React from 'react';
import ReactDOM from 'react-dom';
import Navigation from './Navigation.js';
import WelcomeMessage from './WelcomeMessage.js';

function App() {
    return (
        <div>
            <Navigation />
            <hr/>
            <WelcomeMessage />
        </div>
    );
}

ReactDOM.render(<App/>, document.getElementById("#root"));
```

Question Time

?

Atelier Time

Exercice

« Layout »

Les props

Les props

» Les intérêts de diviser une application en plusieurs briques fondamentales (composants) sont avant tout la **maintenabilité** et la **réutilisabilité**.

» Mais le choix de réutiliser un composant est souvent lié à la volonté de **personnaliser la structure, l'apparence et/ou le contenu** de ce dernier.

» Il est donc nécessaire de pouvoir fournir des **données en entrée** à un composant, afin de **manipuler dynamiquement ces aspects**.

» Pour ce faire, React vous propose le système des **propriétés**, appelées « **props** », à savoir la possibilité de fournir des données en entrée à un composant.

» Comme pour les attributs JSX, si non précisée, la **valeur par défaut** d'une propriété envoyée à un composant est « true ».

» La valeur d'une propriété est en **lecture seule** ! Elle ne peut pas être modifiée !

Envoyer des props

» Pour envoyer des propriétés à un composant, il suffit de fournir des **attributs** (1 attribut = 1 propriété), au niveau du *JSX*, dans la balise de ce composant.

```
<Profile name="John Doe" age={35} />
```

» De manière générale, bien que la notation en chaîne littérale soit supportée, on préférera utiliser la **notation accolade** « {} », qui nous permettra de transmettre des données avec les **bons types** !

» Toute expression *JavaScript* valide peut être envoyée de cette manière (valeur littérale, variables, tableaux, objets, fonctions, ...) .

```
const profileInfos = { name: "John Doe", age: 35 };

<Profile infos={profileInfos} />
```

Accéder aux props

» Pour les composants en **mode fonction**, les props seront disponibles en tant que **1er argument de la fonction**.

```
export default function Profile(props) {
    const { name } = props;
    return (<h1>Bienvenue sur votre espace personnel { name } !</h1>)
}
```

» Pour les composants en **mode classe**, les props seront accessibles via la **propriété d'instance « props »**.

```
export default class Profile extends Component {
    render() {
        const { name } = this.props;
        return (<h1>Bienvenue sur votre espace personnel { name } !</h1>)
    }
}
```

La propriété children

» Il a été expliqué qu'il est possible de représenter un composant en JSX, sous la forme d'une balise double.

» Cette syntaxe permet de fournir à ce composant une **expression JavaScript**.

» **Le plus souvent**, cette expression sera une **arborescence d'autres éléments JSX** (élément HTML ou autres composants).

```
<FenetreModal>
    <Profile infos={profileInfos} />
    <BarreAction/>
</FenetreModal>
```

» Mais comment exploiter cette arborescence afin de l'utiliser au sein de son composant hôte ?

» Il existe une **propriété spéciale**, accessible au sein de vos composants, appelée « **children** », initialisée avec cette fameuse expression !

Utilisation de children

» S'il s'agit d'une propriété à laquelle nous pouvons accéder au sein de nos composants, alors rien ne différentie l'utilisation de « children » par rapport à une autre propriété !

» Vous pouvez l'utiliser comme n'importe quelle autre !

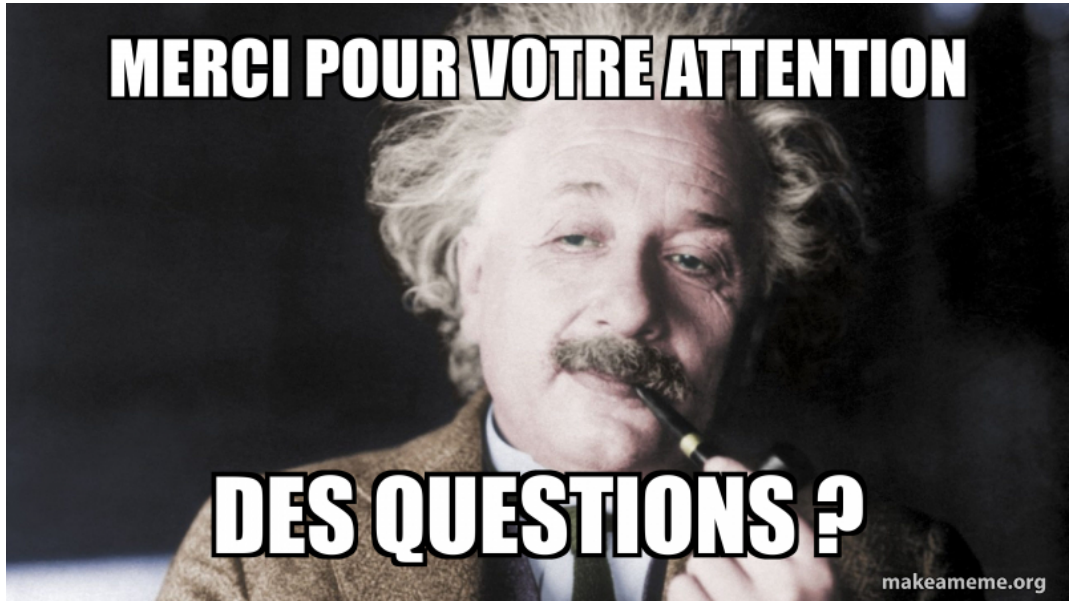
```
export default function FenetreModal(props) {
    return (
        <div className="modal">
            { props.children }
        </div>
    )
}
```

» Il s'agit juste d'une syntaxe de transmission de propriété plus pratique et plus adaptée pour le passage d'expression comme une arborescence de composants.

» Il est à votre appréciation de choisir une syntaxe ou l'autre.

Question Time

?



Atelier Time

Exercice

« Profile »

Atelier Time

Exercice

« Layout children »

États et cycles de vie

Des composants dynamiques

- Jusqu'à maintenant nous avons vu comment créer des composants, les hiérarchiser et leur transmettre des informations (props).
- Cependant, nous n'avons pas encore vu comment **dynamiser** véritablement notre application.
- En effet, bien qu'ils reçoivent des propriétés qui les personnalisent, nos composants ne peuvent actuellement pas voir leur état évoluer.
- En développement web, le concept permettant de faire évoluer l'interface d'une page est celui des **événements.**
- Il existe 2 principaux types d'événements :
  - Les événements liés aux **interactions** de l'utilisateur avec la page
  - Les événements liés au **cycle de vie** de la page et de ses éléments (chargement, …)
- Afin d'utiliser les événements, il nous faudra utiliser les **composants en mode classe.**

Les états

» Nous allons donc voir comment utiliser les événements afin de faire évoluer l'état de nos composants.

» Mais comment allons donc **stocker** l'état de nos composants ?

» Petit rappel : Les « props » sont en lecteur seule, elles ne sont donc pas modifiables.

» Le 1er reflexe serait donc de **créer des propriétés** (autre que « props ») dans les classes de nos composants, chargées de **stocker ces fameux états**.

» Il va cependant nous falloir un moyen d'indiquer à React, que la **mise à jour d'une des propriétés** du composant doit **produire un nouveau rendu** de ce dernier.

» Pour cela, nous avons la méthode « setState() » !

setState

» Tout appel à la méthode d'instance setState() (héritée de React.Component) d'un composant, entraînement un nouveau rendu de ce composant.

» La méthode setState() attend, en paramètre, un objet ou une callback renvoyant un objet (on préfèrera l'utilisation de la callback).

» Les propriétés de cet objet seront ainsi mergées dans une propriété du composant, appelée « state » et qui est chargé de contenir l'état du composant.

» Ainsi, toute donnée d'un composant, dont la mise à jour doit entraîner un nouveau rendu du composant, doit être présente dans cette propriété « state ».

```
this.setState(function(state) {
    return { nightMode: !state.nightMode }
});
```

Initialiser l'état

» Quand on utilise setState() au sein d'un composant, il est **indispensable d'initialiser une fois** la propriété « state » de ce composant.

» On utilisera donc le **constructeur** de la classe afin de faire cette initialisation.

» Attention cependant ! Le constructeur par défaut d'un composant reçoit les « props » en paramètre, qui sont ensuite **envoyées à la classe mère** « React.Component ».

» Si vous redéfinissez le constructeur, il faudra systématiquement appliquer le même principe, sous peine d'erreurs !

```
constructor(props) {
    super(props);
    this.state = { nightMode: false };
}
```

» Nous avons vu comment initialiser et mettre à jour l'état, voyons maintenant où faire appel à setState() !

Les cycles de vie

» Dans React, chaque composant possède un **cycle de vie**, composé de plusieurs **étapes**, elles-mêmes accessibles via des **méthodes** de classe :
  » constructor() : Appelé à la création de l'instance du composant.
  » componentDidMount() : Appelée dés la fin du 1er rendu du composant dans le DOM, après le 1er appel à render().
  » componentDidUpdate() : Appelée à chaque mise à jour de l'état du composant, après render().
  » componentWillUnmount() : Appelée juste avant le démontage du composant dans le DOM et avant la destruction de l'instance.

» Les 2 méthodes les plus utilisées sont componentDidMount() et componentWillUnmount().

» Ces fonctions sont les meilleures candidates pour tout ce qui va concerner l'utilisation des WebAPIs et plus particulièrement les **appels réseaux** (fetch) et les fonctions de **temporisation** (timeout, interval).

Exemple de l'intervalle

- Prenons l'exemple d'un intervalle fait avec setInterval() :
  - Cette méthode est une WebAPI permettant de **lancer une fonction à intervalle de temps régulier**.
  - A son appel, elle nous renvoie un id de l'intervalle que l'on peut utiliser par la suite pour arrêter l'intervalle, grâce à clearInterval(id).
  - Si aucun appel explicite à clearInterval(id) n'est fait, le navigateur **continuera d'exécuter régulièrement la fonction** et ce, **même si le composant, depuis lequel l'intervalle a été créé, est détruit** ! Ce qui risque d'occasionner des **fuites réseaux** et autres problèmes…
- Dans l'exemple précédent, nous aurions donc besoin de :
  - Lancer un intervalle à l'initialisation du composant
  - Stocker l'id de l'intervalle
  - Détruire cet intervalle à la destruction du composant

Exemple de l'intervalle

» Commençons par créer l'intervalle et stocker l'id.

```
componentDidMount() {
    this.id = setInterval(function() {
        // ...
    })
}
```

» Comme vous le voyez, l'id de l'intervalle a été ici stocké dans une propriété classique. En effet, il n'y a aucun intérêt à la stocker dans le state, car cette donnée n'a aucun lien avec le rendu de notre composant !

» Enfin, on n'a plus qu'à indiquer que l'intervalle doit être supprimé à la suppression du composant

```
componentWillUnmount() {
    clearInterval(this.id);
}
```

Question Time

?

Atelier Time

Exercice

« Clock »

Les événements

React et les événements

» Nous allons voir maintenant comment prendre en compte les événements, déclenchés par les **interactions de l'utilisateur** avec la page web.

» Le principe sera sensiblement le même qu'en *JavaScript* et DOM classique.

» Pour chaque événement d'un élément du DOM, il faudra **lier une méthode**, chargée de gérer l'événement.

» La principale différence résidera dans le **nom** qu'il faudra donner à l'attribut représentant l'événement.

» En effet, en HTML classique, le nom d'un événement est sous la forme « onevent », alors qu'en React, il sera de la forme « onEvent ».

```
<button onClick={this.handleNightMode}>Passer en mode nuit</button>
```

Handler et objet event

» La fonction que vous lier à l'événement d'un élément, doit être une **méthode de votre composant**.

» A son appel, React envoie en paramètre à cette méthode, un **objet représentant les informations de l'événement** (position de la souris, infos du click, touche du clavier, ...).

» Cette objet est une instance de la classe SyntheticEvent, qui est une surcouche de la classe Event standard du navigateur (compatibilité cross-browser).

```
handleNightMode(event) {
    this.setState(state => {
        nightMode: true
    });
    alert("Passage en mode nuit !");
}
```

Handler et binding

» Attention ! Rappelez vous qu'un événement est un **phénomène asynchrone** !

» Cela signifie qu'une référence à l'objet « this », au sein de la méthode de l'événement, peut **ne pas faire référence à l'instance de votre composant** !

» Or, dans la plupart des cas, le déclenchement d'un événement nous amènera à faire appel à setState() (exemple précédent), qui n'est accessible que depuis this.

» Il nous faut donc **lier la méthode à l'instance** pour anticiper la perte du this !

» Il est préconisé de faire ce lien au niveau du constructeur.

```
constructor(props) {
    super(props);
    this.handleNightMode = this.handleNightMode.bind(this);
}
```

Passer des paramètres

» Il peut être parfois utile de pouvoir passer des paramètres à la méthode directement depuis le JSX.

» Dans une liste d'élément, par exemple, on pourrait avoir, pour chaque ligne, un bouton permettant de faire des actions spécifiques sur cet élément. Il peut donc être nécessaire de passer des informations concernant cet élément à la méthode de l'événement.

» Vous pouvez donc faire comme suit pour gérer ce genre de cas.

```
handleDelete(event, id) {
    // ...
}

// ...

<button onClick={(event) => this.handleDelete(event, id)}>Supprimer</button>
```

Event handlers en tant que props

» Souvent, on souhaiterait pouvoir **modifier le state d'un composant**, à partir d'un événement **déclenché dans un composant enfant**.

» Pour ce faire, il suffit de définir dans le composant, contenant le state, une méthode permettant de modifier ce state (avec setState()) et de **fournir cette méthode en propriété au composant enfant**, afin que ce dernier puisse l'appeler.

» Il ne faut pas oublier bien sûr de lier la méthode à son instance d'origine (dans le constructeur, comme vu précédemment).

```
handleDelete(event, id) {
    // ...
}

// ...

<ChildComponent onDelete={this.handleDelete} />
```

Question Time

?

Atelier Time

Exercice

« Counter »

Rendu conditionnel & liste

Contenu conditionnel

» On ne peut pas utiliser de structure comme « if » en *JSX*.

» Si vous souhaitez déterminer la valeur d'une variable à insérer dans du *JSX*, vous pouvez en revanche faire un test avec « if » en amont !

```
const age = 18;

// Code non valide !
<div>Je suis { if (age >= 18) { "majeur" } else { "mineur" } }</div>

// Code valide !
let majority = "mineur";
if (age >= 18)
  majority = "majeur"
<div>Je suis { majority }</div>
```

Alternatives raccourcis

» Une alternative moins verbeuse pour représenter une simple condition « if » consiste à utiliser l'opérateur « && ».

```
const age = 18;
<div>{ age >= 18 && "Accès autorisé !" }</div>
```

» Une autre solution de contenu conditionnel, permettant de gérer l'équivalant de « if… else », consiste en l'utilisation des **ternaires** => condition ? expression_si_vrai : expression_si_faux

```
const age = 18;
<div>Je suis { age >= 18 ? "majeur" : "mineur" }</div>
```

Listes

» Pour représenter des listes d'élément, on ne peut pas utiliser de structure comme « for » en *JSX*.

» Si vous souhaitez générer plusieurs éléments, en mode liste, et les insérer dans du *JSX*, vous pouvez en revanche faire une boucle comme « for » en amont !

```
const listeCourse = ["Pain", "Café", "Sucre"];


// Code non valide
<ul>{ for (let item of listeCourse) { <li>{ item }</li> }</ul>

// Code valide
const items = [];
for (let item of listeCourse)
    items.push(<li>{ item }</li>);
<ul>{ items }</ul>
```

Fonctions d’ordre supérieur

» Une autre solution de contenu répétitif, beaucoup plus concise, consiste en l’utilisation des fonctions d’ordre supérieure => map(), filter(), …

```
const listeCourse = ["Pain", "Café", "Sucre"];
<ul>{ listeCourse.map(item => <li>{ item }</li>) }</ul>
```

» Comme vous l’aurez compris, le *JSX* comprend aussi bien l’insertion d’un **seul élément**, que d’un **tableau de plusieurs éléments** !

Les clés

» Vous rappelez-vous de l'algorithme de **réconciliation** de React ?

» Ce dernier analyse les éléments du DOM virtuel et s'occupe de mettre à jour ces éléments s'ils ont changé… et voir les **supprime** même si les nœuds ne correspondent pas !

» On sait que souvent dans une application, les éléments constituant une liste sont susceptibles de **changer de place** dans le DOM (suite à un tri par exemple).

» Or dans ce genre de cas, la **réconciliation entraînera une suppression** des nœuds pour, bien souvent, les recréer à l'identique mais dans un ordre différent !

» Cela n'est pas très efficace et peut entraîner de **lourds problèmes de performance**…

» Mais rassurez-vous, React nous fournit le moyen d'indiquer à l'algorithme un **identifiant pour nos éléments**, afin que ce dernier les retrouve facilement, afin de simplement les réordonner. C'est le système des **clés** (ou key) !

Renseigner une clé

» Pour fournir une clé à un élément, il suffit de lui donner l'attribut key, auquel on va lier une **valeur unique**, qui ne changera jamais (un id par exemple).

» En effet, fournir une clé pouvant varier peut entraîner des **problèmes d'ordonnancement** des éléments !

```
const listeCourse = [
    { id: 1, title: "Pain" },
    { id: 2, title: "Café" },
    { id: 3, title: "Sucre" }
];
<ul>{ listeCourse.map(item => <li key={ item.id }>{ item.title }</li>) }</ul>
```

» Maintenant vous pouvez réordonner la liste comme vous le souhaitez, React le fera sans erreur et efficacement !

Problème du parent commun

» Parfois, on souhaiterait faire un composant afin **d'encapsuler un élément destiné à être représenté dans une liste**, comme la liste des colonnes d'un tableau (<td>) par exemple qui serait ensuite ajoutée à une ligne de tableau (<tr>).

» Mais comme on le sait, un composant ne peut pas renvoyer **plusieurs éléments** sans que ces derniers aient un **parent commun**.

» On peut donc se retrouver dans ce genre de situation.

```
return (
    <div>
        <td>Colonne 1</td>
        <td>Colonne 2</td>
        {/* ... */}
    </div>
)
```

» Le rendu de cet élément dans une ligne peut créer des **problèmes d'affichage**…

Fragment

» C'est pour répondre à cette problématique qu'ont été introduit les Fragments, un composant invisible, servant uniquement à faire office de parent commun à plusieurs éléments JSX.

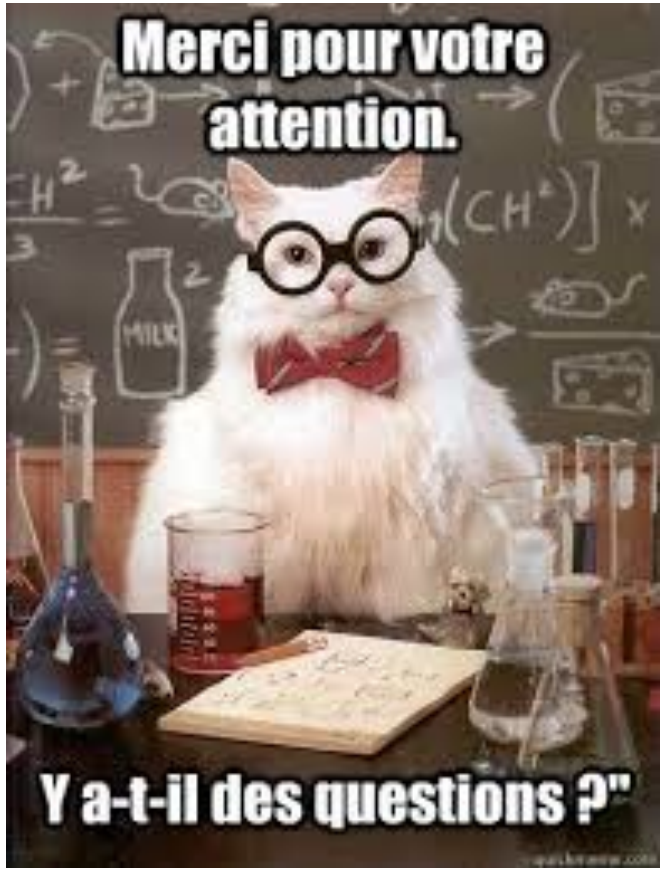
```
<React.Fragment>
    <td>Colonne 1</td>
    <td>Colonne 2</td>
    {/* ... */}
</React.Fragment>
```

» Cette syntaxe peut paraître un peu lourde, aussi il existe un raccourci !

```
<>
    <td>Colonne 1</td>
    <td>Colonne 2</td>
    {/* ... */}
</>
```

Question Time

?



Atelier Time

Exercice

« Apple basket »

Les Formulaires

La source de vérité

» En HTML classique, les éléments de formulaire **gèrent leur propre état interne**.

» Un champ texte (input) contient sa propre valeur, il est sa propre **source de vérité**.

» En React, on va procéder différemment. L'idée est de faire en sorte que le **composant** qui procède au rendu d'un élément de formulaire, devienne la **source de vérité de cet élément**.

» Pour cela, on fera en sorte de maintenir la valeur des éléments du formulaire, au niveau du state du composant.

```
this.state = {
    firstName: "",
    lastName: "",
    age: 0,
};
```

Composant contrôlé

» Il suffit ensuite de **fournir les valeurs du state** en tant que **valeur aux éléments du formulaire** et de **gérer les événements** liés interactions de l'utilisateur afin de **mettre à jour l'état** de l'élément !

» Un composant adoptant cette technique est appelé « **composant contrôlé** ».

```
handleFirstNameChange(event) {
    this.setState({ firstName: event.target.value });
}

render() {
    return (
        <input
            type="text"
            name="firstName"
            value={this.state.firstName}
            onChange={this.handleFirstNameChange}
        />
    )
}
```

Soumission du formulaire

» Concernant la soumission du formulaire, le comportement *HTML* par défaut consiste en un **changement de page** et un **envoi des données** du formulaire en GET/POST/...

» En React, on préfèrera **contrôler nous même ce processus**, en empêchant ce changement de page et décidant la manière dont vont être traitées ces données.

```
handleSubmit(event) {
    event.preventDefault();
    // Éventuel appel à fetch pour envoyer les données du state en post...
}

render() {
    return (
        <form onSubmit={this.handleSubmit}>
            {/* ... */}
            <button type="submit">Envoyer</button>
        </form>
    )
}
```

Valeur des éléments de formulaire

» La plupart des éléments de formulaire possède l'attribut value, permettant de déterminer leur valeur.

» Mais ça n'est pas le cas en *HTML* pour les balises <textarea> et <select>.

» React redéfinit le comportement par défaut de ces balises afin que cela soit le cas.

```
<textarea value={this.state.description} onChange={this.handleDescriptionChange} />

<select value={this.state.sex} onChange={this.handleSexChange}>
    <option value="m">Homme</option>
    <option value="f">Femme</option>
</select>
```

» Pour les <select multiple> acceptant plusieurs valeurs simultanées, il suffit de fournir un **tableau** contenant les values des options que l'on souhaite sélectionner.

Composant non contrôlé

» La plupart des éléments de formulaire peuvent être parfaitement contrôlés.

» Mais certains, dont les valeurs sont en **lecture seule**, **ne le peuvent pas** !

» C'est le cas notamment des champs de type fichier <input type="file"/>.

» Pour pouvoir **récupérer des informations** sur ces éléments, il va falloir passer par une technique appelée « **ref** ».

» Une ref permet de récupérer une **référence** à un élément du DOM, comme on pourrait le faire en *JavaScript* standard avec la méthode querySelector() ou avec le sélecteur $(selector) de jQuery.

Utiliser les refs

» Le principe consiste, en premier lieu, à **définir une ref** dans notre composant.

```
constructor(props) {
    super(props);
    this.fileInput = React.createRef();
}
```

» Ensuite, il faut **lier cette ref à l'élément** dont on souhaite avoir une référence.

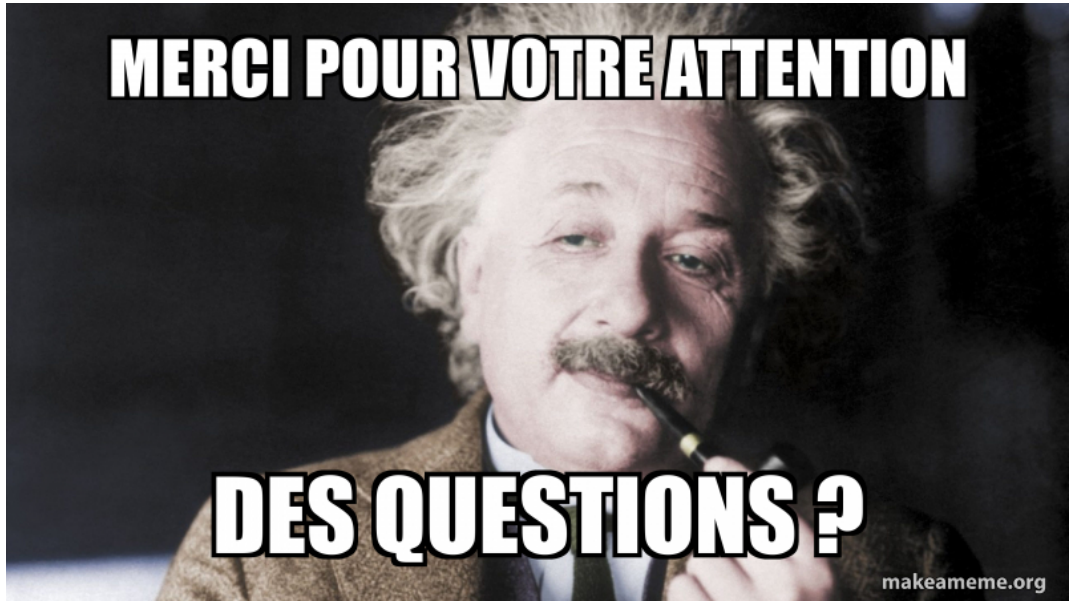
```
<input type="file" ref={this.fileInput} />
```

» Enfin, pour obtenir des infos sur l'élément, il suffit de passer par sa ref.

```
handleSubmit(event) {
    alert(`Nom du fichier : ${this.fileInput.current.files[0].name}`);
}
```

Question Time

?



Atelier Time

Exercice
« Contact form »

Atelier Time

Exercice

« Tasklist »

Optimisation de rendu

shouldComponentUpdate

» Le processus de rendu d'un composant intervient lorsqu'un changement survient au niveau de ses props ou de son état.

» Parfois ces changements n'entraînent aucune modification de l'UI.

» Ainsi les rendus produits consomment des ressources inutilement.

» Il est possible de redéfinir la politique de mise à jour (rendu) d'un composant grâce à la méthode shouldComponentUpdate.

» Cette méthode est appelé par React afin de déterminer s'il doit ou non produire un nouveau rendu du composant.

» Par défaut, elle renvoie true, mais en implémentant nous-même cette méthode, nous pouvons décider des conditions de rechargement d'un composant.

shouldComponentUpdate

» Par défaut

```
shouldComponentUpdate(nextProps, nextState) {
    return true
}
```

» Exemple de redéfinition

```
shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color ≠ nextProps.color) {
        return true;
    }
    return false;
}
```

» Un rendu de ce composant ne sera fait que si la couleur en props change

» Voir démo « optimization - SCU »

Composants purs

» Cette technique permet d'indiquer à React de ne produire un rendu que si les valeurs des props ou états changent, et non systématiquement (rechargement d'un composant parent par exemple, entraînant le rechargement de celui-ci).

» Généralement, l'idée est de comparer les anciennes et nouvelles valeurs de tous les props et états, de manière superficielle. Si une différence est relevée, alors true est renvoyé, sinon false.

» Pour ne pas avoir à faire cette implémentation (possiblement très longue) de shouldComponentUpdate, on peut créer un Composant pur, qui fera exactement la même chose.

```
class MonComposant extends PureComponent {
        // ...
}
```

» Voir démo « optimization - Pure »

Problème des données complexes

» Les composants purs sont pratiques, mais l'utilisation de structures plus complexes (objets, tableaux) en tant que props ou états peuvent entraîner des bugs, si on les utilise mal !

» En effet, une mutation direct sur un objet fourni en tant que props ou état, n'entraînera pas un rechargement du composant pur, étant donné que la comparaison de shouldComponentUpdate sera superficielle (comparaison des références et non des valeurs profondes !).

» La solution consistera en l'utilisation du principe de l'immutabilité ! (Avec l'opérateur spread de ES2015, cela est facilité)

» Voir démo « optimization - ComplexData »

Composants d’ordre supérieur

Une histoire de spécialisation

» L'une des utilisations possibles des closures *JavaScript*, consiste en la **spécialisation de fonctions** apparemment généralistes.

```
function add(a) {
    return function(b) {
        return a + b;
    }
}

const add2 = add(2);
const add5 = add(5);
console.log(add2(4)); // 6
console.log(add5(4)); // 9
```

» Sachez qu'il existe une technique un peu similaire en React, permettant de **spécialiser des composants** se voulant généralistes à la base.

» On appelle de tels composants, des « **Composants d'ordre supérieur** » (ou HOC).

Composition de composant

» Un HOC est une fonction qui prend en paramètre un composant et qui renvoie un autre composant.

» Le composant renvoyé est en fait une transformation du composant passé en paramètre.

» L'idée est de ne pas modifier le composant d'origine, mais plutôt de proposer une version améliorée de ce dernier.

» Voir démo « hoc ».

```
function hoc(WrappedComponent) {
    return class extends React.Component {
        render() {
            return <WrappedComponent />
        }
    };
}
```

Question Time

?

Architecture « flux »

Problèmes de la gestion d’état

» Voir démos « multiple-tasklists »

» Objectif : Reprendre l'exercice « Tasklist » et donner la possibilité d'ajouter plusieurs listes de tâches !

» Problèmes :

- » On doit revoir toute l'architecture de notre application pour lier les tâches aux listes au niveau de l'état ...!
- » On doit revoir la manière de créer les tâches dans les listes et de changer leur nom.
- » On doit remonter tous ces états et handler au niveau du composant <App> et les faire tous redescendre dans les composants enfants, comme pour les headers des listes
- » NIGHTMARE ! Et encore l'application est simple !

Recherche d’une solution

» Source de donnée :

- » On aimerait un méthode ou un système pour pouvoir passer l'état de l'application à nos composants (UI) sans ce passage de props infernal.
- » Il faudrait donc une source de données, globale, unique et indépendante de l'UI (nos composants), qui contiennent tout l'état de notre application.

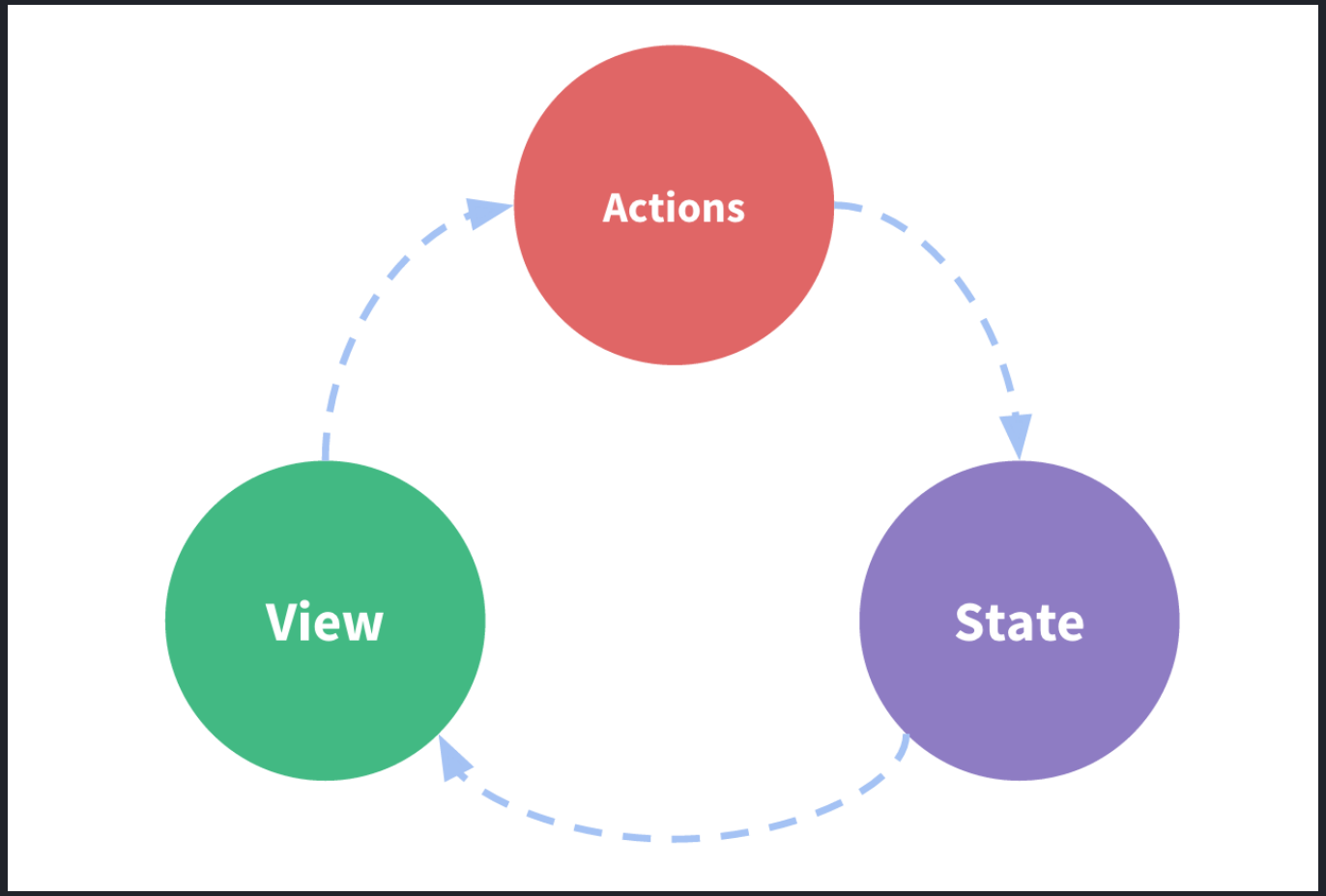
» Modification de l'état :

- » On aimerait aussi que nos composants soient toujours capable de modifier l'état sans passer les event handlers en props.
- » Il faudrait donc un système permettant à nos composants d'effectuer des modifications ou des actions sur l'état de l'application.

Système de gestion d’état

- Ces réflexions sont la base conceptuelle d'un design pattern que l'on appelle la **Gestion d'état** (State management).
- Et les outils qui permettent de mettre en place ce design pattern sont appelés **Systèmes de gestion d'état** (State Management Systems).
- L’architecture la plus connue est appelée « Flux » et a été créée par Facebook.
- De manière générale, les SMS fournissent des fonctionnalités supplémentaires à cette base. Nous allons en découvrir une ou deux ici.
- Mais avant cela, un peu de vocabulaire du monde des SMS :
  - La source de données, qui contient l'état, est appelé un "**store**"
  - Les actions, qui vont modifier l'état, sont appelés des "**actions**"
  - On dit d'une entité (un composant par exemple) qui fait appel à une action, qu'elle "**dispatch**" cette action au store

Schéma d’un SMS



Contrôler l’état

» Bien contrôler l'état d'une application ainsi que les changements qui lui sont apportés est une entreprise difficile. (VanillaJS, jQuery)

» Le debug est un casse-tête et bien souvent, des bugs ne sont pas systématiquement liés à une fonctionnalité en particulier, mais plutôt à l'enchaînement de plusieurs qui, au bout du compte peuvent **corrompre l'état, dans une complexité absolue**...

» Il convient donc de pouvoir parfaitement **maîtriser le comportement de chaque action**.

» Il nous faut donc **éliminer tout effet de bord** éventuel afin de garantir une dimension **déterministe à notre état**.

Notion de réducteur

» La programmation fonctionnelle et notamment le principe des **fonctions pures**, peuvent garantir cette dimension.

» L'idée principale est de considérer qu'un état spécifique (exemple : liste vide) qui subit une action spécifique (exemple : ajout d'un élément « toto ») renverra toujours le même état modifié. (exemple : liste avec un élément « toto »).

» Les SMS introduisent ce principe au travers de ce qu'on appelle des **fonctions de réduction**, ou plus généralement des "**réducteurs**" (reducers).

» Comme leur nom l'indique, on prend 2 éléments (état et action) que l'on réduit en 1 autre (l'état modifié).

Notion de middleware

- Le principe de réducteur introduit la notion **d'historisation** de l'ensemble des actions subies par le store. Cela permet de garder une trace de son évolution précise, ainsi que les données impliquées, tout au long du cycle de vie de notre application.
- Il est possible dans certain SMS de mettre à disposition cet historique à des logiciels tiers (appelés **middlewares**) afin que ces derniers puissent les exploiter.
- Il existe plusieurs cas d'utilisation différents de ces historiques :
  - Logger ces données dans des fichiers
  - Créer des graphiques d'évolution de l'état
  - Faire des systèmes permettant de remonter dans le temps afin de visualiser l'état de l'application à différentes périodes du cycle de vie. On verra comment utiliser l'un de ces systèmes, directement via les devtools de votre navigateur web !
- Il est recommandé, de manière générale, de n'utiliser qu'un seul store par application, à cause notamment des middlewares (stores multiples).

Petit point

» Nous avons vu comment rendre notre état (et les actions associées à ce dernier) indépendants de nos composants, au travers du store. Cette base est quasi la même, algorithmiquement parlant, d'un SMS à un autre.

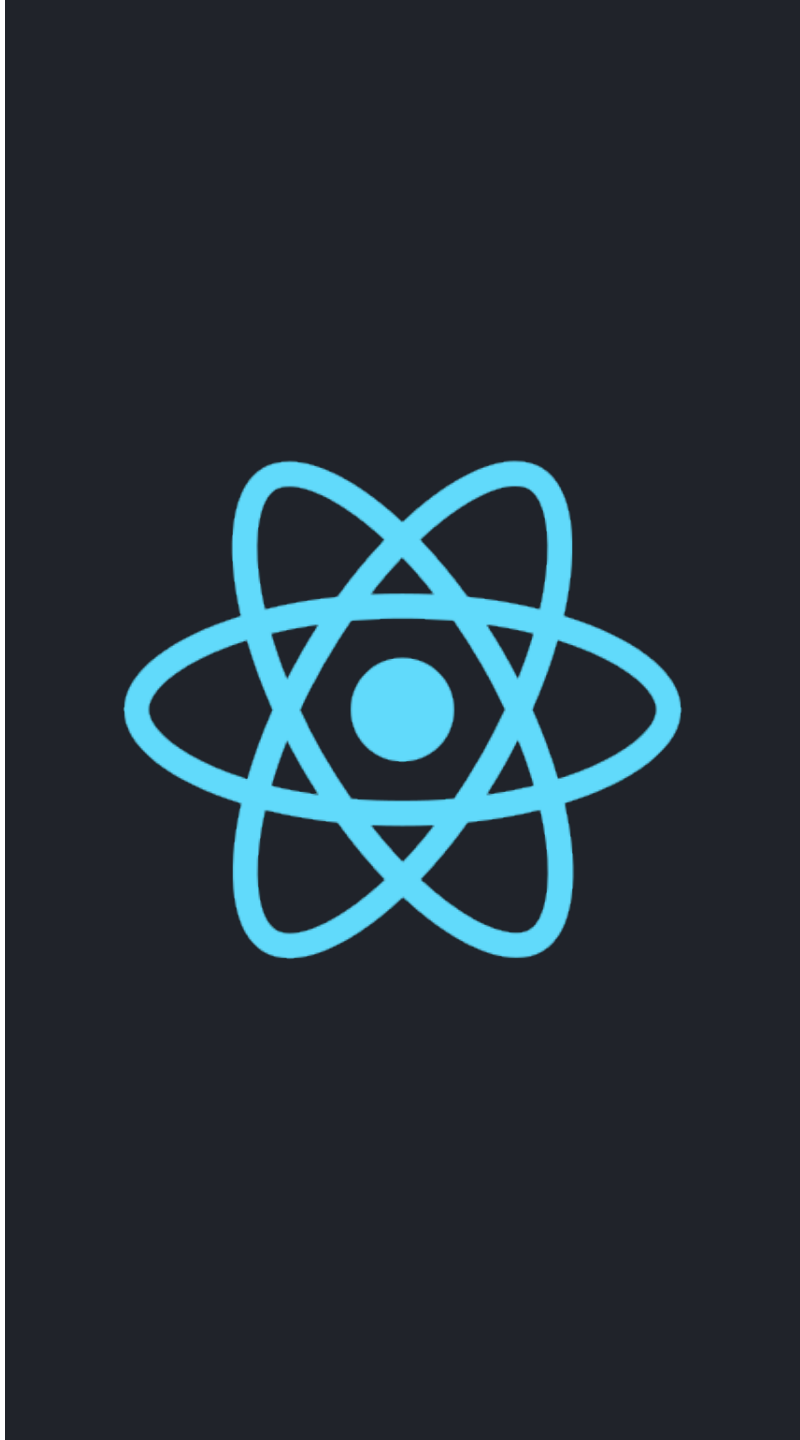
» Mais nous n'avons pas encore parlé de la manière dont nos composants vont pouvoir accéder à l'état et aux actions de ce store. La raison est que cela va avoir tendance à différer d'un SMS à un autre. C'est ce qui va faire, entre autre, leur spécificité.

» Avant de passer au code, nous allons finir pour ces explications en parlant d'une vision, d'un parti pris, au travers du SMS le plus utilisé dans l'écosystème React, celui que l'on appelle Redux.

Question Time

?

Redux



Redux

» Redux est un SMS agnostique. Il a été développé initialement pour s'utiliser avec React, mais son design est d'un niveau d'abstraction tel qu'il peut être utilisé avec n'importe quelle application *JavaScript* qui a besoin de gérer un état.

» Nous utilisons pour notre part React, nous allons donc voir quelles sont les spécificités de cette association React/Redux.

» Il va nous falloir en 1er lieux installer le package « redux ».

Création d'une action

» Créer une action est une tâche très simple !

» Les bonnes pratiques suggèrent de créer en premier lieux les types des actions (appelés « constants »), dans un fichier séparé.

```
// constants/movies.js
export const ADD_MOVIE = 'ADD_MOVIE'
export const DELETE_MOVIE = 'DELETE_MOVIE'
```

» Ensuite, il faut créer un fichier qui contiendra les actions.

```
// actions/movies.js
import { ADD_MOVIE, DELETE_MOVIE } from '../contants/movies'

function addMovie(title) {
    return { type: ADD_MOVIE, title }
}

function deleteMovie(title) {
    return { type: DELETE_MOVIE, title }
}
```

Création d'un reducer

» Il nous faut ensuite créer un reducer, qui sera chargé de renvoyer un nouvel état du store à chaque nouvelle action dispatché par l'application.

```
// reducers/movies.js
import { ADD_MOVIE, DELETE_MOVIE } from '../contants/movies'

const initialState = {
    movies: []
}

export default function movieApp(state = initialState, action) {

    switch (action.type) {
        case ADD_MOVIE:
            return { movies: [...state.movies, { title: action.title }] }
        case DELETE_MOVIE:
            return { movies: state.movies.filter(movie => movie.title != action.title) }
        default:
            return state
    }
}
```

Création d'un store

» Nous avons désormais tous les éléments nécessaires à la création de notre store.

» Comme il peut y avoir plusieurs reducers dans notre application, il conviendra aussi de les combiner pour que le store les utilise !

» On pourra aussi lier le store aux devtools Redux du navigateur !

```
// store.js
import { createStore, combineReducers } from 'redux'
import movieApp from './reducers/movies.js'

const reducers = combineReducers({
  movieApp // , otherReducer, ...
})

export const store = createStore(
    reducers,
    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

Dispatcher une action

» Le store est prêt, maintenant vous pouvez dispatcher des actions dans votre application, afin de mettre à jour l'état du store !

```
// some_script.js
import { store } from './store'
import { addMovie, deleteMovie } from './actions/movies'

store.dispatch(addMovie('Star wars : La revanche des Siths'))
store.dispatch(addMovie(`Harry Potter à l'école des sorciers`))
store.dispatch(addMovie("Dragon Ball Evolution"))
store.dispatch(addMovie('Avengers : Endgame'))

store.dispatch(deleteMovie("Dragon Ball Evolution")) // Ahem... c'était une erreur...
```

Utilisation avec React

» Pour utiliser Redux avec React, il va nous falloir fournir le store à l'arborescence des composants qui devront y accéder.

» Mais contrairement à la gestion classique de l'état sans SMS, le store ne sera fournis qu'à un et un seul nœud de notre application.

» Les composants React pourraient très bien accéder tel quel au store (en mode variable globale) et dispatcher des actions.

» Le problème est que nous perdrions ainsi le principe de "réactivité" !

» Le store ne serait pas vraiment "lié" aux composants. Si l'état du store venait à changer, les modifications ne seraient pas répercutées sur les composants qui en dépendent (comme le fait si bien setState()).

» Nous avons donc besoin que les composants soient à "l'écoute" des changements qui ont lieu dans le store.

Le composant <Provider>

» Pour ce faire, on utilisera le composant <Provider> fourni par le package « react-redux ».

» Il faut donc installer le package « react-redux ».

» Ce composant prend un attribut "store" qui devra contenir le store Redux.

» Les composants dont on souhaite qu'ils puissent accéder au store seront donc des composants enfants de <Provider>.

» Un autre intérêt d'utiliser <Provider> est que les états et actions seront présents directement au niveau des props des composants, plutôt que de devoir importer le store dans le fichier du composant à chaque fois pour y accéder.

Utilisation de <Provider>

» Pour utiliser <Provider>, il faut l'importer et en faire le composant parent de notre application.

```
// index.js
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'
import { store } from './store'
import MovieListContainer from './containers/MovieListContainer'

ReactDOM.render(
  <Provider store={store}>
    <MovieListContainer />
  </Provider>,
  document.getElementById('root')
)
```

» **ATTENTION ! Les choses se compliquent à partir de maintenant !**

Les containers

» Comme nous avons très souvent des **composants de présentations** (affichage en fonction des props), nous ne voulons **pas lier systématiquement** tous les composants au store.

» Cela est inutile et peut être **couteux en terme de performance** comme nous le verrons par la suite.

» Il faut donc considérer que certains composants seront liés au store (on appelle ces composants des **Conteneurs** (containers)) et que d'autres non (composants de présentation).

» Dans la philosophie de React-Redux, l'utilisation de <Provider> ne suffit donc pas.

» Ce composant est le fournisseur du store, il nous faut maintenant définir quels vont être les lieux de livraison !

Le HOC « connect »

» Nous allons pouvoir réaliser cela grâce à un HOC appelé connect().

» Ce HOC permet de lier le store, fourni par <Provider>, à un composant de destination.

» Comme on le sait, un HOC est une fonction qui permet de prendre un composant, et de retourner une version améliorée de ce composant.

» Les composants issus du HOC connect() sont communément appelés des "containers" et sont généralement écrits dans des fichiers spécifiques.

Utiliser connect()

» En réalité, connect() est une fonction qui renvoie un HOC.

» Pour l'utiliser donc, il faut l'appeler, puis utiliser le HOC retourné sur le composant que l'on souhaite transformer en container.

```
// MovieListContainer.js
import { connect } from "react-redux";
import MovieList from "./components/MovieList";

const MovieListContainer = connect()(MovieList);
export default MovieListContainer;
```

» Par défaut, si l'on ne précise pas d'arguments à connect(), ce dernier fournit l'entièreté du store et des actions au container.

» Mais **cela n'est pas recommandé** à plusieurs titres…

Problème de sémantique

» Cela n'a pas de sens de fournir l’entièreté du store et des actions à un container, étant donné qu'un container n'utilise en général qu'une partie seulement de l’état du store.

» Si un développeur, qui découvre le projet, regarde le fichier du container, il ne comprendra pas quel est l'objectif du container !

Problème de performance

» À chaque fois que l'état du store sera modifié (dispatch d'une action), les containers en seront notifiés.

» L'utilisation par défaut de connect() entraînera donc systématiquement la génération d'un nouveau rendu du container.

» On comprend donc que si l'état qui a été modifié n'a aucun lien avec un container (si le container ne l'utilise pas), alors ce dernier aura été rechargé pour rien.

» Ceci ralentira inutilement votre application ! :(

Problème de lisibilité

» Si vous laissez le comportement par défaut de connect(), les actions ne seront pas passées au niveau des props.

» À la place, il s'agira de la fonction dispatch() qui sera passée.

» Ainsi, il vous faudra importer les actions dans votre composant et appeler manuellement dispatch() sur ces actions.

```
this.props.dispatch({ type: "ADD_MOVIE", title: "Avengers : Endgame" })
```

» Il n'est jamais conseillé de faire ainsi !

» Votre composant n'est censé faire que 2 choses :

- » Afficher des données
- » Appeler des fonctions dont les noms ont du sens et qui s'occuperont de faire le dispatch de l'action pour vous.

Problème de lisibilité

» Introduire de la sémantique liée à Redux (dispatch) au sein de la logique de votre composant n'est pas une bonne pratique et **peut conduire à des erreurs**.

» Une meilleure version de l'appel à cette action serait :

```
this.props.addMovie("Avengers : Endgame")
```

Les arguments de connect

» Pour palier à ces inconvénients, connect() peut prendre plusieurs arguments.

» Ce sont les 2 premiers qui nous intéressent ici :

- » la fonction mapStateToProps
- » la fonction mapDispatchToProps

mapStateToProps()

» La fonction mapStateToProps prend 2 paramètres :
  » Le state issus du store de l'application
  » Les props éventuellement envoyées au container lors de son insertion dans le JSX.

» Comme son nom l'indique, cette fonction doit renvoyer un objet qui sera fusionné aux props envoyées au composant (celui passé à connect()).

```
const mapStateToProps = (state, props) => {
    return { movies: state.movies };
};
```

» Ainsi, le processus de rendu du composant sera lié à ces props.

» En effet, si les valeurs des props, envoyées au composant, ont changé depuis le dernier rendu, un nouveau rendu sera fait (comme shouldComponentUpdate). Sinon non.

mapDispatchToProps()

» La fonction mapDispatchToProps possède 1 paramètre :

  » La fonction dispatch

» Comme son nom l'indique, cette fonction doit renvoyer un objet qui sera aussi fusionné aux props du composant (celui passé à connect()).

» Les valeurs associées au clés de cet objet seront les fonctions de dispatchs des actions.

```
const mapDispatchToProps = (dispatch, props) => {
    return {
        handleAddMovie: (title) => { dispatch(addMovie(title)); },
        handleDeleteMovie: (title) => { dispatch(deleteMovie(title)); },
    };
};
```

Rendu final du HOC connect

```
import { connect } from 'react-redux'
import { addMovie, deleteMovie } from '../actions'
import MovieList from '../components/MovieList'

const mapStateToProps = (state, props) => {
  return { movies: state.movies }
}


const mapDispatchToProps = (dispatch, props) => {
  return {
    handleAddMovie: (title) => { dispatch(addMovie(title)) },
    handleDeleteMovie: (title) => { dispatch(deleteMovie(title)) }
  }
}


const MovieListContainer = connect(mapStateToProps, mapDispatchToProps)(MovieList)


export default MovieListContainer
```

Exemple du composant MovieList

```
// MovieList.js
import React, { Component } from "react";

export default class MovieList extends Component {
    render() {
        const { movies, handleAddMovie, handleDeleteMovie } = this.props;

        return (
            <>
                <button onClick={ handleAddMovie("Nouveau film") }>
                    Ajouter un nouveau film !
                </button>
                <ul>
                    {movies.map((movie) => (
                        <li onClick={ handleDeleteMovie(movie.title) }>
                            {movie.title}
                        </li>
                    ))}
                </ul>
            </>
        );
    }
}
```

Intérêts des containers

- La compréhension et l'écriture des containers peuvent paraître compliquées quand on découvre Redux.
- Dans les débuts de son utilisation, les développeurs avaient tendance à n'utiliser qu'un seul container à la racine de l'arborescence des composants et faisaient ainsi passer le contenu du store au travers de l'arbre des composants, via les props.
- Mais l'expérience a montré que cette technique présentait 3 problèmes majeurs :
  - D'une part, le problème évoqué plus tôt concernant le processus de rendu des composants présentationnels qui sera systématique.
  - D'autre part, il s'agit d'un anti-pattern, dans le sens où l'intérêt initial d'un SMS est d'éviter ce passage de props au travers des différents niveaux de l'arbre des composants.
  - Ensuite, les résultats de benchmarking ont montré que l'abus de cette technique dégradait les performances, tout comme le fait de vouloir connecter systématiquement le store à tous les composants.

Arborescence Redux conseillée

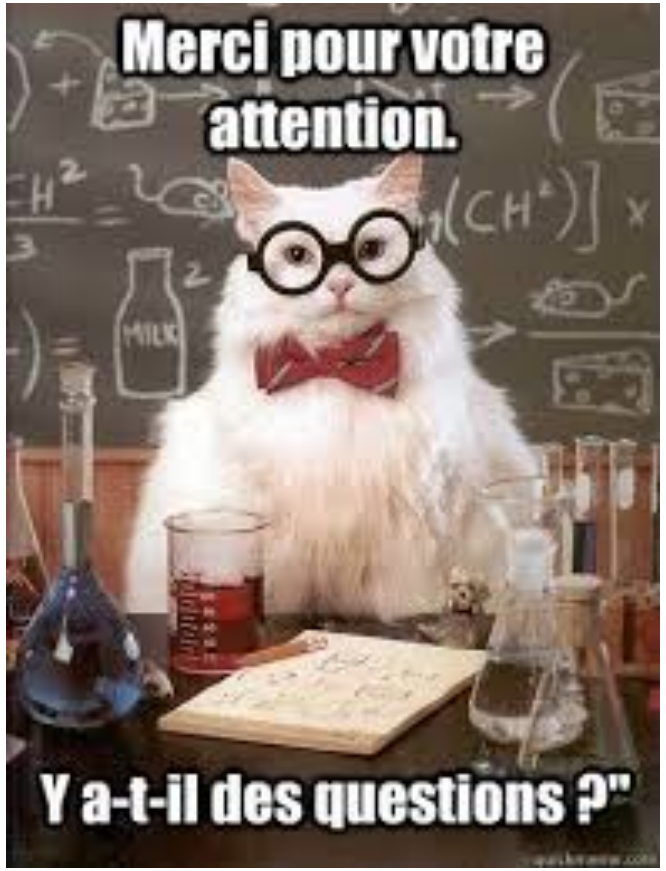
» Le secret de Redux réside donc dans la création, la configuration et la répartition intelligente des containers et des composants de présentation.

» Arborescence de l'application :

- » constants/
- » actions/
- » reducers/
- » store.js
- » containers/
- » components/
- » pages/ (si router)

Question Time

?



Atelier Time

Exercice

« Tasklist redux »

Fin du mon.. module !