Les bases de React

Présentation

React :

- Est une librairie et non un framework
- Permet de créer des interfaces graphiques utilisateurs
- Utilise un Dom Virtuel pour diminuer le temps de réponse du DOM
- Peut être utilisé pour générer des pages dynamiques côté client et/ou côté serveur
- Peut s'intégrer avec différents frameworks
- Ne s'occupe de l'affichage et de rien d'autre !

Le DOM

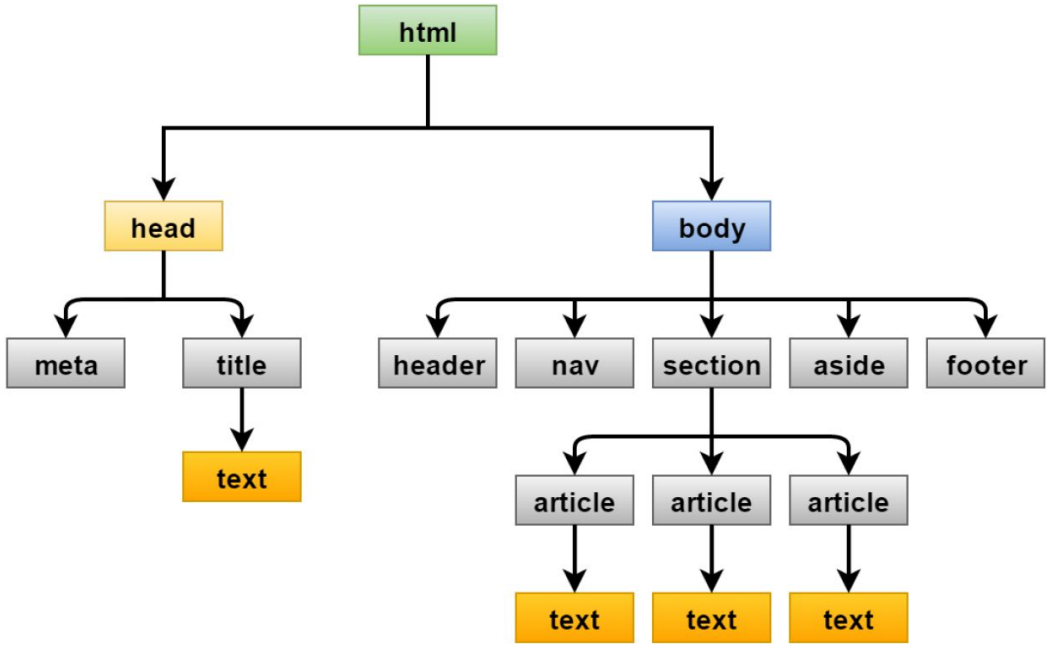
Le DOM (Document Object Model) est une représentation d'un document sous forme structurée, généralement sous forme d’arbre.

- Pour le navigateur, chaque balise est un objet avec des attributs et des méthodes
- Les balises et valeurs sont appelés **"noeuds"** dans le DOM
- Un noeud est appelé "parent" s'il a des noeuds descendants, ces derniers sont appelés "noeuds enfants"
- Les noeuds du même parent sont "frères"

Le DOM

```
<html>
  <head>
    <meta charset ="UTF 8">
    <title>Titre de ma page<title/>
  </head>
  <body>
    <header></header>
    <nav></nav>
    <section>
      <article>Texte de l'article 1</article>
      <article>Texte de l'article 2</article>
      <article>Texte de l'article 3</article>
    </section>
    <aside></aside>
    <footer></footer>
  </body>
</html>
```

DOM Simplifié :



DOM virtuel

React utilise un **DOM virtuel** :

- Une représentation en JavaScript du DOM
- Est transformé en DOM réel au premier affichage
- Le DOM réel est ensuite modifié à chaque modification du DOM virtuel

Création d’une application React

create-react-app

`create-react-app` est un outil permettant de générer un squelette d'application React.

- Utilisable avec npx
  - Node Package eXecute
- Permet d'intégrer directement TypeScript dans un projet React

Créer un projet React avec TypeScript :

```
$ npx create-react-app my-app --template typescript
```

Démarrer l'application React

Un script est automatiquement créé dans le fichier **`package.json`**, permettant de démarrer le projet en utilisant **`npm`** et **`react-scripts`** :

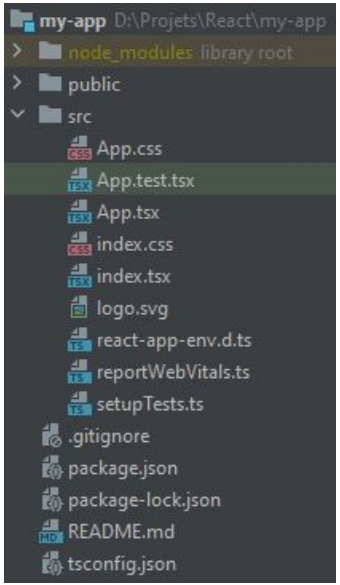
```
$ npm run start
```

Un fichier **`.env`** peut être créé afin d'y ajouter des variables d'environnement permettant de configurer l'application React :

```
// .env
PORT=4000 // Change le port de l'application
BROWSER=none // Empêche l'ouverture d'un nouvel onglet à chaque démarrage
```

Structure d'un projet React

Différents dossiers et fichiers sont ainsi créés :



`node_modules`	Contient les dépendances du projets définies dans le fichier `package.json`
`public`	Contient les ressources publiques (médias, polices, …)
`index.tsx`	Point d'entrée du projet
`package.json`	Fichier de configuration du projet
`tsconfig.json`	Fichier de configuration de TypeScript

package.json

Ce fichier hérité de Node.js permet de configurer le projet via différentes propriétés :

`name`	Nom du projet
`version`	Version du projet
`dependencies`	Liste les dépendances externes du projet qui seront téléchargées dans le dossier `node_modules`
`devDependencies`	Liste les dépendances externes nécessaires uniquement durant le développement et absentes à l'exécution
`scripts`	Liste des scripts pouvant être exécutées pour ce projet à l'aide de la commande `npm run [SCRIPT]`

index.tsx

Ce fichier est le point d'entrée du projet et est exécuté par react-scripts :

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

const root = ReactDOM.createRoot(                  // Permet d'initialiser l'application React
  document.getElementById('root') as HTMLElement   // dans l'élément ayant pour id 'root'
);                                                 // Voir fichier public/index.html

root.render(          // Dans l'élément root, afficher les composants suivants
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Exercice

Exercice

Créez une application React :

- à l'aide de create-react-app,
- nommée **game-of-thrones-characters**

JSX

Présentation

JSX :

- Est une extension syntaxique à JavaScript
- Conçue par et pour React
- Permet de se rapprocher de la syntaxe HTML pour l'écriture de composants React
- Est compilé en JavaScript compréhensible par le navigateur

```
root.render(
  <div>
    <App />
  </div>
);
```

```
root.render(
  React.createElement('div', null,
    React.createElement(App, null, null)
  )
);
```

Utiliser des expressions en JSX

Afficher une variable en JSX :

```
const name = 'Toto';

<div>
  <b>{name}</b>
</div>
```

Afficher le retour d'une fonction :

```
<div>
  <b>{getName(user)}</b>
</div>
```

Fragment 1/2

Une expression JSX ne doit contenir qu'une seule balise (qui elle peut avoir des enfants) :

```
const ok =
  <div>
    <a href="...">Link 1</a>
    <a href="...">Link 2</a>
  </div>;

const notOk =   // Erreur de compilation !
  <a href="...">Link 1</a>
  <a href="...">Link 2</a>;
```

Fragment 2/2

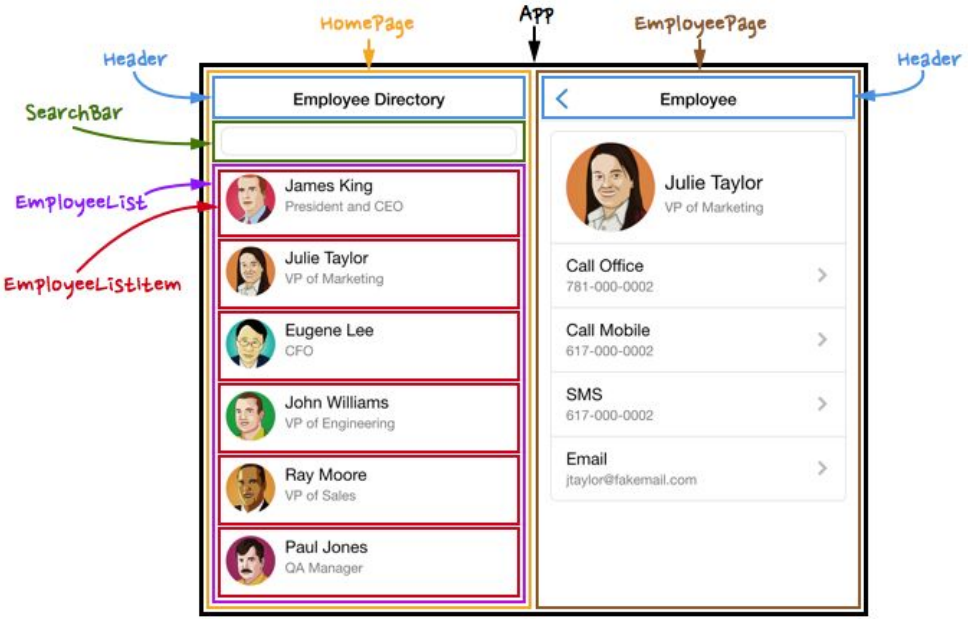
Lorsqu'une expression JSX doit comporter plusieurs balises mais qu'une balise "wrapper" ne devrait pas être présente (car nuirait à l'affichage du tout), l'expression JSX peut alors être contenue dans une balise **`Fragment`** :

```
import React from 'react';

const jsx =
  <React.Fragment>
    <a href="...">Link 1</a>
    <a href="...">Link 2</a>
  </React.Fragment>;

// Raccourci :
const jsx =
  <>
    <a href="...">Link 1</a>
    <a href="...">Link 2</a>
  </>;
```

Composants React

Composants

Une application React est divisée en composants ayant chacun une fonctionnalité précise.



Class components 1/2

Les composants React peuvent être créé de deux manières, dont via l'utilisation de class components :

- En créant des classes héritant de `React.Component`
- La fonction `render` doit retourner le contenu à être affiché

Class components 2/2

Exemple :

```
class HelloWorld extends React.Component {

  render() {
    return (
      <h1>Hello world !</h1>
    );
  }

}
```

Stateless functional components

Les composants peuvent également être créés sous la forme de fonctions sans état, plus simples à créer :

```
const HelloWorld: React.FunctionComponent = () => {
  return (
    <h1>Hello world !</h1>
  );
};

// Ou :

const HelloWorld: React.FC = () =>
  <h1>Hello world !</h1>;
```

Props 1/3

Tout composant peut accepter des valeurs, dites "propriétés" :

- Passées à l'utilisation du composant
- Dont le nombre et les types peuvent être spécifiés à la création du composant
- Provoquent le recalcul (rerender) du composant à chaque modification d'une propriété
- Sont récupérées via le constructeur des *class components*
    - Utilisables via `this.props`
- Ou sont passés directement aux *functional components*

Props 2/3

Exemple *class component* :

```
interface Props {
  title: string,
  subTitle?: string // Propriété optionnelle
}
```

```
class HelloWorld extends React.Component<Props> {
  constructor(props: Props) {
    super(props);
  }

  render() {
    const { title, subTitle } = this.props;

    return (
      <>
        <h1>{title}</h1>
        {subTitle && // Si subTitle est défini, alors ...
          <h2>{subTitle}</h2>
        }
      </>
    );
  }
}
```

Props 3/3

Exemple functional component :

```
const HelloWorld: React.FC<Props> = ({ title, subTitle }) =>
  <>
    <h1>{title}</h1>
    {subTitle &&
      <h2>{subTitle}</h2>
    }
  </>;

// Utilisation :
<HelloWorld
  title="Hello world !"
  subTitle="This is a subTitle"
/>
```

Children

La propriété **`children`** peut être utilisée pour passer des enfants à un composant :

```tsx
interface Props {
  children: React.ReactNode // Représente n'importe quel valeur JSX
}

const Title: React.FC<Props> = ({ children }) =>
  <h1>{children}</h1>;

<Title>
  Hello <b>World</b> !
</Title>
```

Exercice

Exercice

Créez la base de l'application Game of Thrones Characters :

- Créez un composant `CharactersList` affichant :
  - Un titre
  - Une liste statique de personnages de Game of Thrones avec pour chacun :
    - Un ID
    - Un nom
    - Une image
- Les personnages doivent pour l'instant être créés dans un tableau "global"
  - Créez une structure de donnée représentant ces personnages

Hooks

Présentation

Les hooks :

- Existent depuis React 16.8
- Visent à apporter des fonctionnalités des class components aux functional components
- Rendent les class components inutiles
- Ne peuvent être utilisé qu'à la **racine de functional components** (pas dans des boucles par exemple)

Gérer l'état d'un composant avec useState 1/3

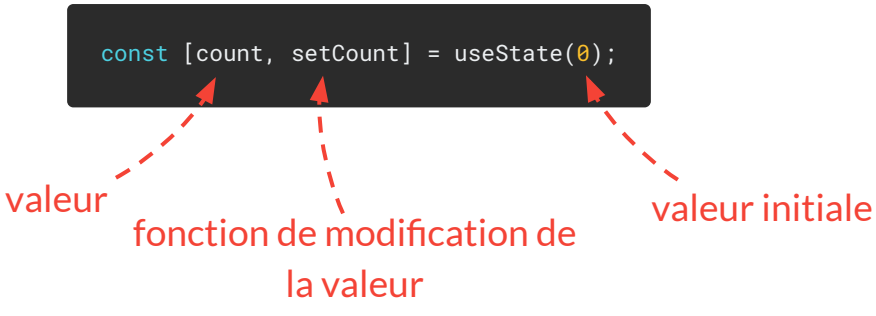
Les composants peuvent avoir un état représenté par une ou plusieurs variables :

- Une liste de valeurs à afficher
- Le retour d'une API REST
- L'état d'ouverture d'un popup
- …

React fourni un hook permettant de gérer les variables d'état d'un composant : `useState`

Gérer l'état d'un composant avec useState 2/3

```tsx
import React, { useState } from 'react';

const Counter: React.FC = () => {
  const [count, setCount] = useState(0);

  return (
    <span>Counter : {count}</span>
  );
};
```



Gérer l'état d'un composant avec useState 3/3

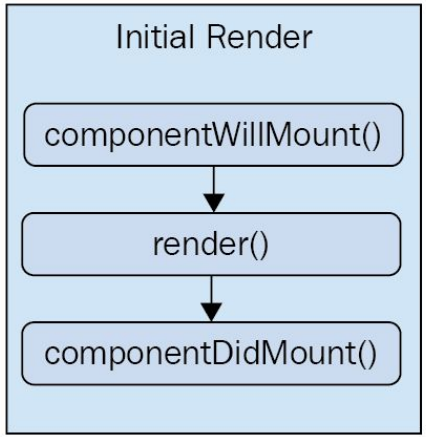
Attention : une modification d'une variable d'état entraîne le rerender du composant !

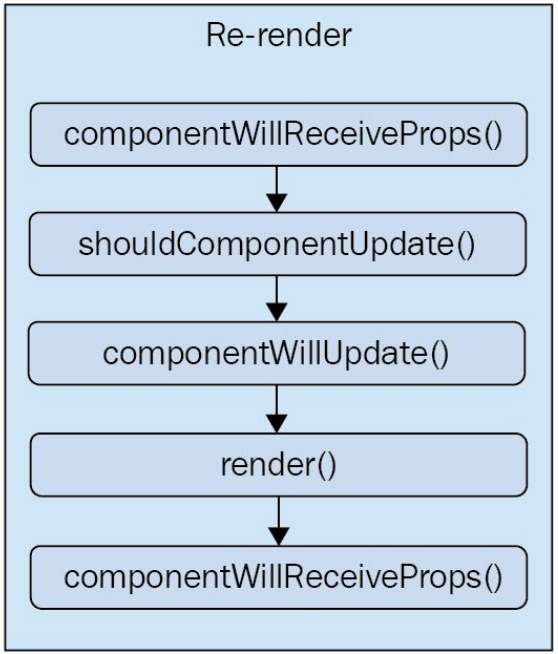
```
const Counter: React.FC = () => {
  const [count, setCount] = useState(0);

  setCount(count + 1); // Entraîne une boucle infinie !

  return (
    <span>Counter : {count}</span>
  );
};
```

Cycle de vie d'un composant avec useEffect 1/3

Les composants React possèdent un cycle de vie auquel le développeur peut réagir via différentes fonctions dans les **class components**.

Ces fonctions sont remplacées dans les **functional components** par le hook `useEffect`.





Cycle de vie d'un composant avec useEffect 2/3

Le hook `useEffect` prend en paramètre :

- une fonction de **callback** qui sera appelée lors des étapes `componentDidMount` et `componentDidUpdate`
- un **tableau de dépendances** permettant de limiter les appels au callback : ce dernier ne sera appelé que lorsqu'une des valeurs des dépendances sera modifiée

Cycle de vie d'un composant avec useEffect 3/3

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

// Sera appelée au chargement du composant et à chaque rerender
useEffect(() => {
  console.log('useEffect !');
});

// Sera appelée au chargement et à chaque fois que la variable count changera
useEffect(() => {
  console.log('useEffect !');
}, [count]); // Peut être une variable de props ou de state

// Ne sera appelé qu'une seule fois au chargement
useEffect(() => {
  console.log('useEffect !');
}, []);
```

Réagir aux actions de l'utilisateur

Fonctions de callback

Il est possible de récupérer les actions de l'utilisateur grâce à des fonctions de callback à placer sur les différents éléments de la page (comme en JavaScript "classique") :

- sur des champs de texte (`onChange`, `onKeyPress`, ...)
- sur des boutons (`onClick`, `onTouch`, ...)
- sur des formulaires (`onSubmit`)
- etc etc ...

Récupérer la valeur d'un champ de texte 1/2

Les éléments input possèdent une propriété `onChange` permettant de réagir à chaque nouveau caractère entré par l'utilisateur.

Il prend en paramètre un objet event contenant les informations du champ de texte.

```
<input onChange={(e) => console.log(e.target.value)} />
```

Récupérer la valeur d'un champ de texte 2/2

Il est possible de récupérer la valeur du champ et de la stocker, par exemple, dans une variable d'état du composant.

```
const SearchBar: React.FC = () => {
  const [searchTerm, setSearchTerm] = useState('');

  return (
    <input
      name="search"
      value={searchTerm}
      onChange={(e) => setSearchTerm(e.target.value)}
    />
  );
};
```

Réagir au clic sur un bouton (ou autre)

L'action de clic d'un utilisateur sur un élément peut être récupérée via la propriété **`onClick`** :

```
const Counter: React.FC = () => {
  const [count, setCount] = useState(0);

  return (
    <>
      <span>Counter : {count}</span>
      <button onClick={() => setCount(count + 1)}>Count !</button>
    </>
  );
};
```

Propagation des callbacks

Un composant peut passer une fonction callback à un de ces composants enfants via ses props :

```
interface Props {
  onCountIncrements: (count: number) => void
}

const Counter: React.FC<Props> = ({ onCountIncrements }) => {
  const [count, setCount] = useState(0);

  const onClick = () => {
    setCount(count + 1);
    onCountIncrements(count + 1);
  };

  return (<>
    <span>Counter : {count}</span>
    <button onClick={onClick}>Count !</button>
  </>);
};

<Counter onCountIncrements={(count) => console.log('new count :', count)} />
```

useRef

Le hook **`useRef`** peut être utilisé sur n'importe quel élément afin d'en récupérer une référence et la manipuler :

```
const SearchBar: React.FC = () => {
  const [searchTerm, setSearchTerm] = useState('');
  const inputRef = useRef<HTMLInputElement>(null);

  useEffect(() => {
    inputRef?.current?.focus();
  }, []);

  return (
    <input
      name="search"
      ref={inputRef}
      value={searchTerm}
      onChange={(e) => setSearchTerm(e.target.value)}
    />
  );
};
```

Exercice

Exercice

Créez un composant Character qui va afficher les détails d'un personnage avec :

- Son ID
- Son nom
- Son portrait
- Son titre
- Sa famille

Modifiez le modèle de données en conséquence.

Exercice

Lors d'un clic sur un personnage de la liste, afficher, au-dessus de celle-ci, les détails du personnage sélectionné.

Un clic sur un autre personnage changera le personnage sélectionné.

Exercice

Encore au-dessus, affichez un formulaire vous permettant d'ajouter un nouveau personnage avec :

- Son nom,
- Son titre,
- Sa famille,
- L'URL de son portrait

Un clic sur le bouton de validation ajoutera le personnage à la liste.

L'appuie sur la touche entrée lorsqu'un champ est focus passera le focus au prochain champ.
L'appuie sur la touche entrée dans le dernier champ valide également le formulaire.

Au chargement de la page, le focus doit être mis sur le champ "nom".