REACT

Notions de rappels javascript

Les fonctions fléchées

```
const paragraph = document.body.querySelector("p");

function showText(domNode, string) {
  return domNode.innerHTML = string
}

showText(paragraph, "Hello World !");
```

Avec les fonctions fléchées :

```
const paragraph = document.body.querySelector("p");

const showText = (domNode, string) => {
  return domNode.innerHTML = string
}

showText(paragraph, "Je suis le résultat d'une fonction fléchée !");
```

Le nom de la fonction est déclaré en constante.
Entre parenthèse viennent les paramètres de la fonction.
Enfin la flèche amène la logique de la fonction.

> 👨‍💻 Vous tomberez certainement sur plusieurs variantes de fonctions fléchées, par exemple :

```
const showText = string => paragraph.innerHTML = string;

showText("Moins c'est long, plus c'est bon ! (en javascript)")
```

Ici pas de parenthèses nécessaires du fait qu'il n'y a qu'un paramètre.
N'ayant qu'un return dans cette fonction, pas besoin d'accolade non plus, le return est écrit sur la même ligne.

> Fonctions fléchées - JavaScript | MDN (mozilla.org)

Les instructions *let* et *const*

La portée des variables

let et *const* permettent de déclarer une variable dont la portée est celle du bloc où elle est déclarée.

> ℹ️ La portée d'une variable désigne l'espace dans laquelle cette variable sera disponible.

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  // c'est la même variable !
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // c'est une variable différente
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

⚠️ ***const* crée une référence !**

Exemple :

```
const constante = 43;
constante = "String" // génèrera une erreur
```

Cela ne signifie pas que la valeur référencée ne peut pas être modifiée ! Ainsi, si le contenu de la constante est un objet, l'objet lui-même pourra toujours être modifié.

> const - JavaScript | MDN (mozilla.org)
> let - JavaScript | MDN (mozilla.org)

Les méthodes map()

La fonction map() permet d'itérer un tableau, permettant d'effectuer une certaine logique à chaque itération.

```
const array = ["lion", "chat"];

array.map(animal => console.log(animal));
// affichera dans la console les uns après à les autres dans la console "lion", "chat"
```

Les méthodes filter() et find()

La méthode filter() crée et retourne un nouveau tableau en itérant sur le tableau de base et acceptant une fonction callback.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

La méthode find() retournera la première itération du tableau qui respecte la condition donnée.

La déstructuration

Permet d'extraire des données d'un tableau ou d'un object.

```
const {a, b} = {a: 10, b: 20};
console.log(a); // 10
console.log(b); // 20
```

Les spread operators

Dans l'exemple ci-dessous, les valeurs du tableau itérable `articulations` sont insérées les unes après les autres dans le tableau `corps`.

```
const articulations = ['épaules', 'genoux'];
const corps = ['têtes', ...articulations, 'bras', 'pieds'];
// ["têtes", "épaules", "genoux", "bras", "pieds"]
```

La logique React

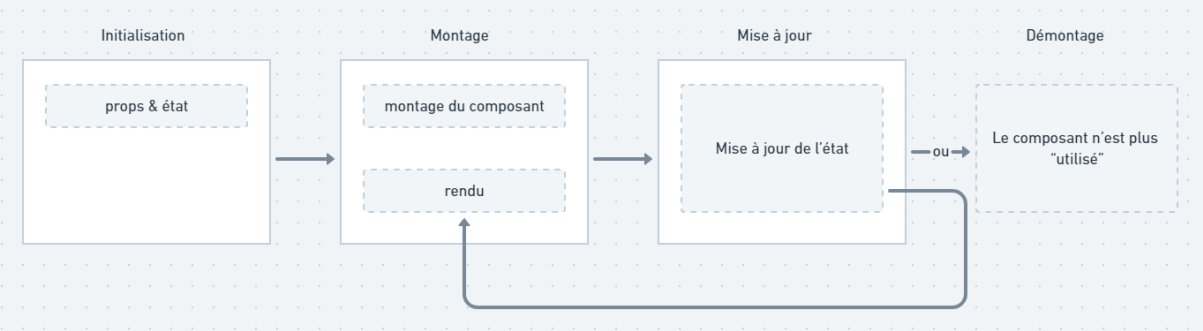
> *Penser en React revient à découper l'interface utilisateur en éléments indépendants et réutilisables.*

Ces éléments sont appelés des **composants**. Chaque composant enferme sa propre logique, son rendu (JSX) et si besoin son CSS.

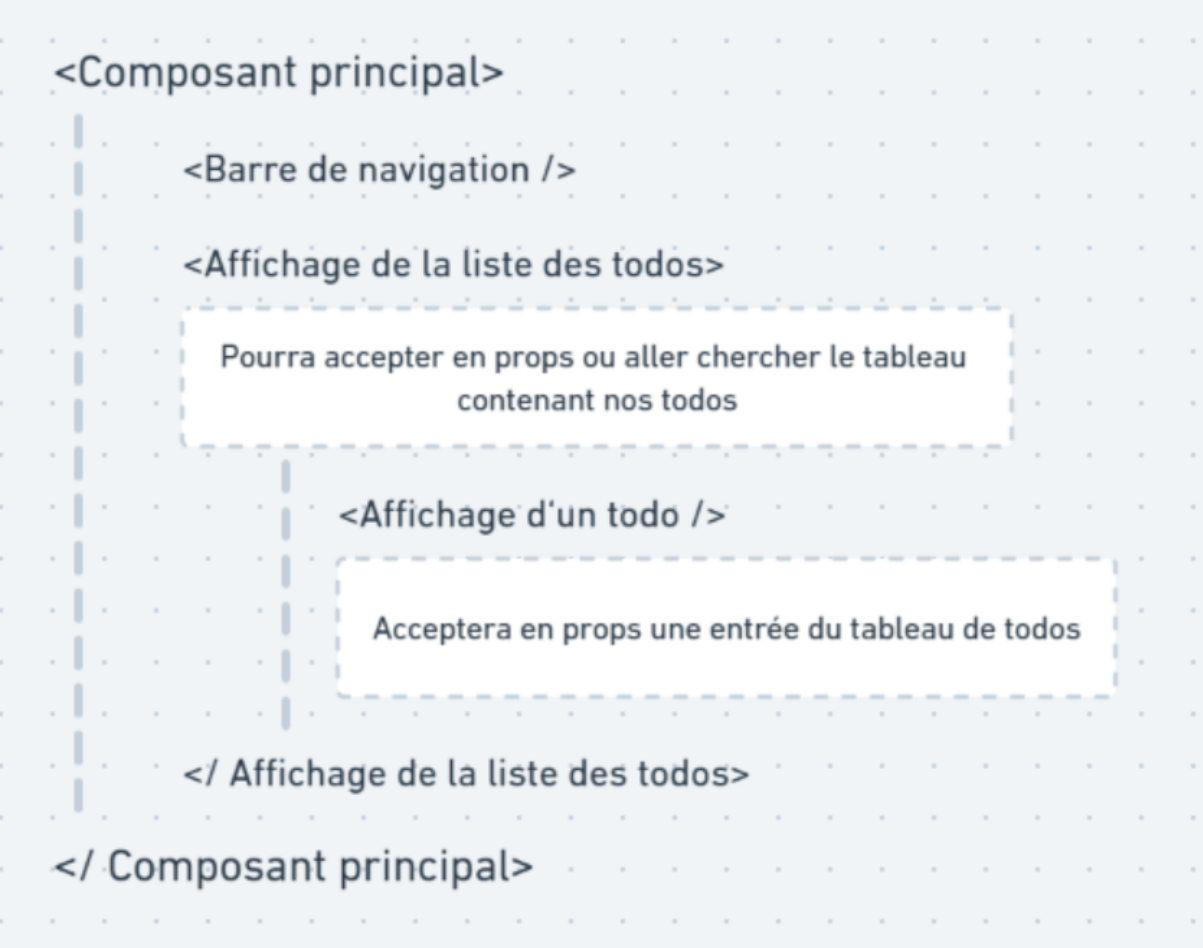
JSX est une manière d’écrire du HTML dans nos **composants**. Il sera retranscrit ensuite en HTML.

Nos **composants** peuvent accepter des **props**. Par exemple, un composant affichant une liste de to-do pourra accepter en **props** le tableau contenant ces to-dos.

Chaque composant a un **état dit local**. Cet état dépend entre autres des props passés et de variables déclarés au sein du composant. Chaque altération apportée à une de ces props entraînera un nouveau “rendu” du composant, ou également appelé “montage” (***mount*** en anglais). Tous ceci correspond au **cycle de vie** du composant.



Cycle de vie d’un composant



Exemple simplifié de d’une application React

Dans quelque contexte ?

- Développé sous React offre une souplesse et un certain confort. De sa structure moléculaire, le code est plus facile à entretenir. Il permet également à chaque membre d’une équipe de développer des parties individuelles.
- Une des librairies front les plus utilisées et donc une grande communauté.
- React peut être utilisé seul, au sein d’un Framework front tel que Next.js ou encore en complément de certains Framework back tel que Symfony.

Installation d’un simple projet React

> Create React App (create-react-app.dev)

Arborescence des fichiers

Arborescence présente lors de CRA

node_modules

Contient les dépendances dont votre application a besoin.

.gitignore

Fichier trackant les dossiers et fichiers à ne pas commit sur github.

package.json

Contient des metadatas importantes à votre projet telles que le nom de votre projet, sa version, ou certains scripts, ex: npm start.
Liste également les dépendances du projet qui permettront de les installer lors de l'installation du projet.

yarn.lock (ou package-lock.json)

Assure que les packages soient consistants peu importe la machine faisant tourner le projet en listant les versions des dépendances des packages listés.

Dossier /public

Contient les fichiers accessibles de l'application.

index.html
Template desservi au lancement de l'application par npm start. Les composants React sont injectés dans la div racine. Des librairies CSS ou autres peuvent y être appelés.

Dossier /src

Comprendra nos composants, test, CSS etc..

App.css

Contient les styles de notre application.

Comment structurer son projet React

Il n'y a pas de structure imposé/officielle !

> *React n’a pas d’opinion sur la manière dont vous ordonnez vos fichiers à l’intérieur de vos dossiers.*
> Structure de fichiers – React (reactjs.org)

Ce qui ne veut pas dire que le projet ne doit pas être organisé. Il existe cependant des standards, telle que grouper vos fichiers par fonctionnalité ou par route, ou encore de grouper par type de fichier (ex: un dossier component, un dossier utils pour d’autres fonctions, etc…).

Codons ensemble !

Nous allons coder une mini application avec des to-dos.

Notions vues au cours du code

- La mise en place de composants
- Hooks d’état (useState et useEffect)
  Il en existe d’autres !
  Introduction aux Hooks – React (reactjs.org)
- L’affichage conditionnel
- Liste et clés
- Etat et cycle de vie

Exercices

- [ ] Vous allez coder une simple onepage avec une liste de posts, dans un premier temps de façon statique, c’est-à-dire que les posts sont affichés directement dans le HTML.
- [ ] Si possible, découpé votre onepage en composants réutilisables, par exemple un composant <Post /> qui aura comme props un post, et qui s’occupera de l’afficher.

```
[
  {
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
```

```
  },
  {
  "userId": 1,
  "id": 2,
  "title": "quis ut nam facilis et officia qui",
  "completed": false
  },
  {
  "userId": 1,
  "id": 3,
  "title": "fugiat veniam minus",
  "completed": false
  },
  {
  "userId": 1,
  "id": 4,
  "title": "et porro tempora",
  "completed": true
  },
  {
  "userId": 1,
  "id": 5,
  "title": "laboriosam mollitia et enim quasi adipisci quia provident illum",
  "completed": false
  },
]
```

Annexe

Quick Start (reactjs.org)