

React JS Development

Pourquoi React est si populaire ?

React est une librairie Javascript dont le but est de simplifier le développement d'interfaces.

Développé en 2011 par Facebook et open-sourced en 2013, il est utilisé par de nombreuses applis.

Son objectif principal est de faciliter le raisonnement sur une interface et son état à tout moment, en divisant l'interface utilisateur en une collection de composants.

Moins complexe que les autres alternatives

Au moment où React a été annoncé, Ember.js et Angular 1.x étaient les choix prédominants en tant que framework. Ces deux éléments imposaient tellement de conventions au code que le portage d'une application existante n'était pas du tout pratique.

React a fait le choix d'être très facile à intégrer dans un projet existant, car c'est ainsi qu'ils ont dû procéder chez Facebook afin de l'introduire dans la base de code existante. De plus, ces 2 frameworks ont apporté trop de choses sur la table, tandis que React a uniquement choisi d'implémenter la couche View au lieu de la pile MVC complète.

Timing parfait

À l'époque, Angular 2.x a été annoncé par Google, avec l'incompatibilité descendante et les changements majeurs qu'il allait apporter.

Passer d'Angular 1 à 2 était comme passer à un framework différent, donc cela, ainsi que les améliorations de la vitesse d'exécution promises par React, en ont provoqué une rapide adoption de React.

React est-il simple à apprendre ?

Même si j'ai dit que React est plus simple que les frameworks alternatifs, plonger dans React est toujours compliqué, mais surtout à cause des technologies corollaires qui peuvent être intégrées à React, comme Redux et GraphQL.

React en lui-même a une très petite API, et vous devez essentiellement comprendre 4 concepts pour commencer :

- Components
- JSX
- States
- Props

Comment exploiter React ?

Le plus simple est d'ajouter directement le fichier JavaScript React dans la page. C'est mieux lorsque votre application React interagira avec les éléments présents sur une seule page et ne contrôlera pas réellement tout l'aspect de la navigation.

Dans ce cas, vous ajoutez 2 balises de script à la fin du body:

```
<html>
  ...
  <body>
    ...
    <script

src="https://cdnjs.cloudflare.com/ajax/libs/react/16.8.3/umd/react.development.js"
    crossorigin
  ></script>
  <script

src="https://cdnjs.cloudflare.com/ajax/libs/react-dom/16.8.3/umd/react-dom.production.m
in.js"
    crossorigin
  ></script>
</body>
</html>
```

Comment exploiter React ?

Ici, nous avons chargé à la fois React et React DOM. Pourquoi 2 bibliothèques ? Car React est 100 % indépendant du navigateur et peut être utilisé en dehors de celui-ci (par exemple sur les appareils mobiles avec React Native). D'où la nécessité de React DOM, pour ajouter les wrappers pour le navigateur.

Après ces balises, vous pouvez charger vos fichiers JavaScript qui utilisent React, ou même JavaScript en ligne dans une balise script:


```
<script src="app.js"></script>
```

```
<!-- or -->
```

```
<script>  
  //my app  
</script>
```

Comment exploiter React ?

Pour utiliser JSX vous avez besoin d'une étape supplémentaire : charger Babel

```
<script      src="https://unpkg.com/babel-standalone@6/babel.min.js">
</script>
```

et chargez vos scripts avec le MIME **text/babelType** :

```
<script src="app.js" type="text/babel"></script>
```

Comment exploiter React ?

Nous pouvons maintenant ajouter du JSX dans notre app.js :

```
const Button = () => {  
  return <button>Click me!</button>  
}
```

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

Autre possibilité : create-react-app

create-react-app est un projet visant à vous familiariser avec React en un rien de temps, et toute application React qui doit dépasser une seule page trouvera que create-react-app répond à ce besoin.

Vous commencez par utiliser npx, qui est un moyen simple de télécharger et d'exécuter des commandes Node.js sans les installer. npx est livré avec npm depuis la version 5.2.

```
npx create-react-app todolist
```

Autre possibilité : create-react-app

create-react-app créé une structure de fichiers dans le dossier que vous avez indiqué (todolist dans ce cas) et initialisé un dépôt Git .

Il a également ajouté quelques commandes dans le fichier package.json, de sorte que vous pouvez immédiatement démarrer l'application en allant dans le dossier et exécuter **npm start**.

En plus de npm start, create-react-app a ajouté quelques autres commandes :

- **npm run build:** pour construire les fichiers de l'application React dans le dossier build, prêt à être déployé sur un serveur
- **npm test:** pour exécuter la suite de tests en utilisant **Jest**

Autre possibilité : create-react-app

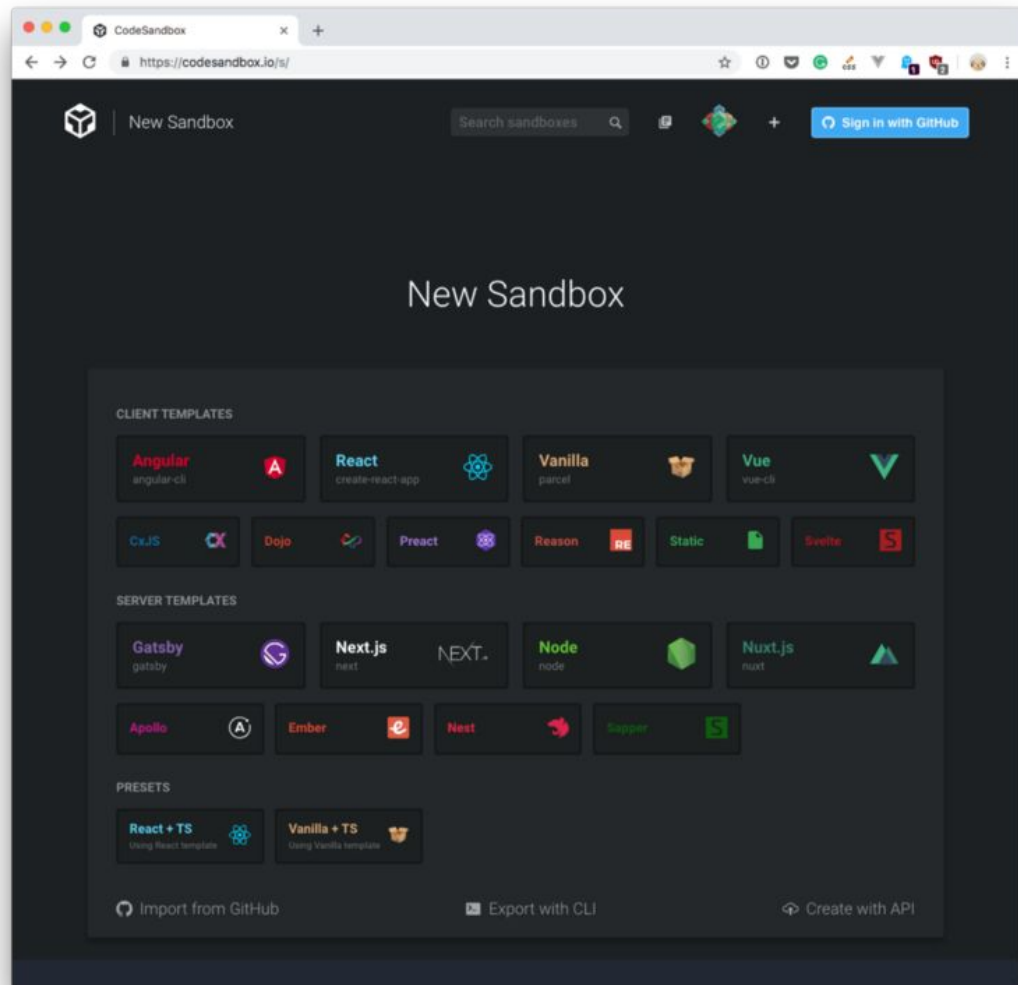
Si vous avez déjà installé une application React à l'aide d'une ancienne version de React, vérifiez d'abord la version en ajoutant `console.log(React.version)` dans votre application, vous pouvez mettre à jour en exécutant

`yarn add react@16.7`, et `yarn` vous invitera à mettre à jour (choisissez la dernière version disponible). Répétez pour `yarn add react-dom@16.7` (changez "16.7" avec la version la plus récente de React pour le moment)

Code Sandbox

Un moyen facile d'avoir la structure create-react-app, sans l'installer, est d'aller sur <https://codesandbox.io/s> et de choisir "React".

CodeSandbox est un excellent moyen de démarrer un projet React sans avoir à l'installer localement.



Codepen

<https://codepen.io/>

Les « pens » Codepen sont parfaits pour les projets rapides avec un seul fichier JavaScript, tandis que les « projets » sont parfaits pour les projets avec plusieurs fichiers, comme ceux que nous utiliserons le plus lors de la création d'applications React.

Une chose à noter est que dans Codepen, en raison de son fonctionnement interne, vous n'utilisez pas les modules ES classiques **import**, mais plutôt d'importer par exemple **useState**, donc on y utilise :

```
const { useState } = React
```

et non pas :

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

Concepts JS

ECMAScript 6



avec React

Variables

Une variable doit être déclarée avant de pouvoir l'utiliser. Il y a 3 façons de le faire, en utilisant **var**, **let** ou **const**, et ces 3 manières diffèrent dans la façon dont vous pouvez interagir avec la variable plus tard.

Jusqu'à ES2015, **var** était le seul constructeur disponible pour définir les variables.

Variables

Une variable initialisée avec **var** en dehors de toute fonction est affecté à l'objet global, a une portée globale et est visible partout.

Une variable initialisée avec **var** à l'intérieur d'une fonction est assignée à cette fonction, elle est locale et n'est visible qu'à l'intérieur, tout comme un paramètre de fonction.

Variables

Il est important de comprendre qu'un bloc (identifié par une paire d'accolades) ne définit pas une nouvelle portée.

Une nouvelle portée n'est créée que lorsqu'une fonction est créée, car **var** n'a pas de portée de bloc, mais de portée de fonction.

Variables

let est une nouvelle fonctionnalité introduite dans ES2015 et il s'agit essentiellement d'une version à portée de bloc de `var`. Sa portée est limitée au bloc, à l'instruction ou à l'expression où elle est définie et à tous les blocs internes contenus.

Les développeurs JavaScript modernes peuvent choisir d'utiliser uniquement `let` et abandonner complètement l'utilisation de `var`.

Définir **let** en dehors de toute fonction – contrairement à `var` – ne crée pas de variable globale.

Variables

Les variables déclarées avec **var** ou **let** peuvent être modifiées ultérieurement dans le programme et réaffectées. Une fois que **const** est initialisée, sa valeur ne peut plus jamais être modifiée et ne peut pas être réaffectée à une valeur différente.

const ne fournit pas l'immutabilité, mais s'assure simplement que la référence ne peut pas être modifiée.

const a une portée de bloc, identique à **let**.

Les développeurs JavaScript modernes peuvent choisir de toujours utiliser **const** pour les variables qui n'ont pas besoin d'être réaffectées plus tard dans le programme.

Pourquoi? Parce que nous devrions toujours utiliser la construction la plus simple disponible pour éviter de faire des erreurs sur la route.

Fonctions Fléchées

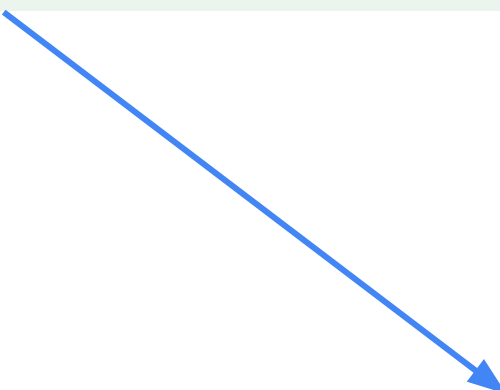
Les fonctions fléchées ont été introduites dans ES6 / ECMAScript 2015, et depuis leur introduction, elles ont changé à jamais l'apparence (et le fonctionnement) du code JavaScript.

À mon avis, ce changement était si accueillant que vous voyez maintenant rarement l'utilisation du mot-clé `function` dans les bases de code modernes.

Visuellement, c'est un changement simple et bienvenu, qui vous permet d'écrire des fonctions avec une syntaxe plus courte :

Fonctions Fléchées

```
const myFunction = function() {  
  //...  
}
```



```
const myFunction = () => {  
  //...  
}
```

Fonctions Fléchées

Si le corps de la fonction ne contient qu'une seule instruction, vous pouvez omettre les crochets et tout écrire sur une seule ligne :

```
const myFunction = () => doSomething()
```

Les paramètres sont passés entre parenthèses :

```
const myFunction = (param1, param2) => doSomething(param1, param2)
```

Fonctions Fléchées

S'il n'y a qu'un seul paramètre, on peut même oublier les parenthèses :

```
const myFunction = param => doSomething(param)
```

Retour Implicite

Les fonctions fléchées permettent d'avoir un retour implicite : les valeurs sont retournées sans avoir à utiliser le mot-clé **return**.

Cela fonctionne lorsqu'il y a une instruction d'une ligne dans le corps de la fonction :

```
const myFunction = () => 'test'
```

```
myFunction() // 'test'
```

Retour Implicite

Un autre exemple, lors du retour d'un objet, n'oubliez pas d'envelopper les accolades entre parenthèses pour éviter qu'il ne soit considéré comme les crochets du corps de la fonction d'enveloppement :

```
const myFunction = () => ({ value: 'test' })
```

```
myFunction() //{value: 'test'}
```

this

this est un concept qui peut être compliqué à appréhender, car il varie beaucoup selon le contexte et varie également selon le mode de JavaScript (mode strict ou non).

Il est important de clarifier ce concept car les fonctions fléchées se comportent très différemment des fonctions normales.

Lorsqu'il est défini comme une méthode d'un objet, dans une fonction régulière **this fait référence** à l'objet, vous pouvez donc faire :

this

```
const car = {  
  model: 'Fiesta',  
  manufacturer: 'Ford',  
  fullName: function() {  
    return `${this.manufacturer} ${this.model}`  
  }  
}
```

l'appel de **car.fullName()** renverra “Ford Fiesta”

this

Le **scope** de **this** dans les fonctions fléchées est **hérité** du contexte d'exécution.

Une fonction fléchée ne lie pas du tout **this**, donc sa valeur sera recherchée dans la pile des appels. Ici dans le code suivant, **car.fullName()** ne fonctionnera pas et renverra la chaîne “undefined undefined”

this

```
const car = {  
  model: 'Fiesta',  
  manufacturer: 'Ford',  
  fullName: () => {  
    return `${this.manufacturer} ${this.model}`  
  }  
}
```

this

Pour cette raison, les fonctions fléchées ne sont pas adaptées en tant que méthodes d'objet.

Les fonctions fléchées ne peuvent pas non plus être utilisées comme constructeurs, lorsque l'instanciation d'un objet lèvera une `TypeError`.

C'est là que les fonctions normales doivent être utilisées à la place, lorsque le contexte dynamique n'est pas nécessaire .

C'est également un problème lors de la gestion des événements. Le **DOM Event Listener** établit le **this** comme élément cible, et si vous comptez sur `this` dans un gestionnaire d'événement, une fonction régulière est nécessaire :

this

```
const link = document.querySelector('#link')
link.addEventListener('click', () => {
  // this === window
})
```

```
const link = document.querySelector('#link')
link.addEventListener('click', function() {
  // this === link
})
```

Rest et spread

Vous pouvez “étendre” un tableau, un objet ou une chaîne de caractère avec l’opérateur spread `...`

Exemple avec un array :

```
const a = [1, 2, 3]
```

```
const b = [...a, 4, 5, 6] //nouveau tableau
```

```
const c = [...a] //copie de tableau
```

Rest et spread

Cela fonctionne également avec les objets :

```
const newObj = {...oldObj}
```

Avec les String, il crée un array séparant chaque caractère :

```
const hey = 'hey'  
const arrayized = [...hey] // ['h', 'e', 'y']
```

Rest et spread

Cet opérateur a des applications pratiques. La plus utile est la possibilité d'utiliser un tableau comme argument de fonction d'une manière très simple :

```
const f = (foo, bar) => {}  
const a = [1, 2]  
f(...a)
```

Rest et spread

L'élément **rest** `...` est utile lorsqu'on destructure les tableaux :

```
const numbers = [1, 2, 3, 4, 5]  
[first, second, ...others] = numbers
```

et lorsqu'on “étale” (spread) les éléments :

```
const numbers = [1, 2, 3, 4, 5]  
const sum = (a, b, c, d, e) => a + b + c + d + e  
const sumOfNumbers = sum(...numbers)
```

Rest et spread

ES2018 introduit l'utilisation de rest pour les objets :

```
const { first, second, ...others } = {  
  first: 1,  
  second: 2,  
  third: 3,  
  fourth: 4,  
  fifth: 5  
}  
  
first // 1  
second // 2  
others // { third: 3, fourth: 4, fifth: 5 }
```


Rest et spread

Les propriétés du **spread** permettent de créer un nouvel objet en combinant les propriétés de l'objet passées après l'opérateur de propagation :

```
const items = { first, second, ...others }  
items //{ first: 1, second: 2, third: 3, fourth: 4, fifth: 5 }
```

Déstructuration d'objets et de tableaux

Avec un un objet, en utilisant la syntaxe de déstructuration, vous pouvez extraire quelques valeurs et les mettre dans des variables nommées :

```
const person = {  
  firstName: 'Tom',  
  lastName: 'Cruise',  
  actor: true,  
  age: 54 //made up  
}
```

```
const { firstName: name, age } = person //name: Tom, age: 54
```

Maintenant name et age contiennent ces valeurs

Déstructuration d'objets et de tableaux

Cette syntaxe fonctionne également sur les tableaux :

```
const a = [1, 2, 3, 4, 5]  
const [first, second] = a
```

Cette instruction crée 3 nouvelles variables en obtenant les éléments d'indice 0, 1, 4 du tableau a:

```
const [first, second, , , fifth] = a
```

Template literals

Les modèles de littéraux sont une nouvelle fonctionnalité ES2015/ES6 qui vous permet de travailler avec des chaînes d'une manière nouvelle par rapport à ES5 et aux versions antérieures.

La syntaxe à première vue est très simple, il suffit d'utiliser des backticks au lieu de guillemets simples ou doubles :

```
const a_string = `something`
```

Template literals

Ils sont uniques car ils offrent de nombreuses fonctionnalités que les chaînes normales construites avec des guillemets n'offrent pas, en particulier :

- ils offrent une excellente syntaxe pour définir des chaînes multilignes
- ils fournissent un moyen facile d'interpoler (insérer) des variables et des expressions dans des chaînes
- ils vous permettent de créer des DSL avec des balises de modèle (DSL signifie Domain Specific Language, et il est par exemple utilisé dans React by Styled Components, pour définir le CSS pour un composant)

Nous allons les aborder plus en détail.

Multiline strings

Avant l'ES6, pour créer une chaîne s'étendant sur deux lignes, vous deviez utiliser le caractère `\` en fin de ligne :

```
const string =  
  'first part \  
  second part'
```

Cela permet de créer une chaîne sur 2 lignes, mais elle est rendue sur une seule ligne : **first part second part**

Multiline strings

Pour rendre également la chaîne sur plusieurs lignes, vous devez explicitement ajouter `\n` à la fin de chaque ligne, comme ceci :

```
const string =  
    'first line\n \  
second line'
```

```
const string = 'first line\n' + 'second line'
```

Multiline strings

Les littéraux de modèle rendent les chaînes multilignes beaucoup plus simples.

Une fois qu'un modèle littéral est ouvert avec le backtick, il vous suffit d'appuyer sur Entrée pour créer une nouvelle ligne, sans caractères spéciaux, et elle est rendue telle quelle :

```
const string = `Salut  
tout  
le  
monde!`
```

```
const string = `First  
                Second`
```


Multiline strings

Nous avons un petit problème dans le second exemple c'est que l'espace est pris en compte et décale énormément le mot Second à l'affichage.

Pour éviter cela, utiliser la méthode `trim()` juste après le backtick de fermeture et placez une première ligne vide.

```
const string =  
`  
First  
Second`.trim()
```

Interpolation

Les littéraux de modèle offrent un moyen simple d'interpoler des variables et des expressions dans des chaînes.

Vous le faites en utilisant la syntaxe **`${variable}`**:

```
const myVariable = 'bleu'  
const string = `Chat ${myVariable}` //Chat bleu
```

Interpolation

à l'intérieur de `${}` vous pouvez ajouter n'importe quoi, même des expressions :

```
const string = `something ${1 + 2 + 3}`  
const string2 = `something ${foo() ? 'x' : 'y'}`
```

Les Classes

En 2015, la norme ECMAScript 6 (ES6) a introduit les classes.

JavaScript a un moyen assez rare d'implémenter l'héritage : **l'héritage prototypique**. L'héritage de prototype est différent de l'implémentation de l'héritage de la plupart des autres langages de programmation populaires, qui est basée sur les classes.

Les personnes venant de Java ou Python ou d'autres langages avaient du mal à comprendre les subtilités de l'héritage prototypique, alors le comité ECMAScript a décidé de saupoudrer de sucre syntaxique sur l'héritage prototypique afin qu'il ressemble à la façon dont l'héritage basé sur les classes fonctionne dans d'autres implémentations populaires.

C'est important : JavaScript sous le capot est toujours le même, et vous pouvez accéder à un prototype d'objet de la manière habituelle.

Définition de Classe

Voilà à quoi ressemble une classe

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
  
    hello() {  
        return 'Salut je m\'appelle' + this.name + '.'  
    }  
}
```

Définition de Classe

Une classe a un identifiant, que nous pouvons utiliser pour créer de nouveaux objets en utilisant **new ClassIdentifier()**.

Lorsque l'objet est initialisé, la méthode **constructor** est appelée, avec tous les paramètres passés.

Une classe a aussi autant de méthodes qu'elle en a besoin. Dans ce cas **hello** est une méthode et peut être appelée sur tous les objets dérivés de cette classe :

```
const jojo = new Person('Jojo')  
jojo.hello()
```

Héritage de Classe

Une classe peut étendre une autre classe et les objets initialisés à l'aide de cette classe héritent de toutes les méthodes des deux classes.

Si la classe héritée possède une méthode portant le même nom que l'une des classes supérieures dans la hiérarchie, la méthode la plus proche est prioritaire :

Héritage de Classe

```
class Person {  
  constructor(name) {  
    this.name = name  
  }  
  
  hello() {  
    return 'Salut je m\'appelle ' +  
this.name + '.'  
  }  
}
```

A l'intérieur d'une classe, vous pouvez référencer la classe parente appelant `super()`.

```
class Programmer extends Person {  
  hello() {  
    return super.hello() + ' Je suis un joueur.'  
  }  
}  
  
const jojo = new Programmer('Jojo')  
jojo.hello()
```


Méthodes statiques

Normalement, les méthodes sont définies pour une instance, pas pour une classe.

Les méthodes statiques sont exécutées par la classe directement et non pas par une instance:

```
class Person {  
    static personHello() {  
        return 'Hello'  
    }  
}
```

```
Person.personHello() //Hello
```

Getters et setters

Vous pouvez ajouter des méthodes préfixées par **get** ou **set** pour créer un **getter** et un **setter**, qui sont deux morceaux de code différents qui sont exécutés en fonction de ce que vous faites : accéder à la variable ou modifier sa valeur.

Getters et setters

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
  
    set name(value) {  
        this.name = value  
    }  
  
    get name() {  
        return this.name  
    }  
}
```

Getters et setters

Si vous n'avez qu'un getter, la propriété ne peut pas être définie et toute tentative de la faire sera ignorée.

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
  
    get name() {  
        return this.name  
    }  
}
```

Getters et setters

Si vous n'avez qu'un setter, vous pouvez modifier la valeur mais pas y accéder de l'extérieur

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
  
    set name() {  
        this.name = value  
    }  
}
```

Callbacks

Les ordinateurs sont asynchrones par conception.

Asynchrone signifie que les choses peuvent se produire indépendamment du flux principal du programme.

Dans les ordinateurs grand public actuels, chaque programme s'exécute pendant un intervalle de temps spécifique, puis il arrête son exécution pour laisser un autre programme poursuivre son exécution. Cette chose fonctionne dans un cycle si rapide qu'il est impossible de le remarquer, et nous pensons que nos ordinateurs exécutent de nombreux programmes simultanément, mais c'est une illusion (sauf sur les machines multiprocesseurs).

Callbacks

Les programmes utilisent en interne des interruptions , un signal qui est émis vers le processeur pour attirer l'attention

Simplement gardez à l'esprit qu'il est normal que les programmes soient asynchrones et arrêtent leur exécution jusqu'à ce qu'ils aient besoin d'attention, et que l'ordinateur puisse exécuter d'autres choses entre-temps. Lorsqu'un programme attend une réponse du réseau, il ne peut pas arrêter le processeur tant que la requête n'est pas terminée.

Callbacks

Normalement, les langages de programmation sont synchrones et certains offrent un moyen de gérer l'asynchronisme, dans le langage ou via des bibliothèques. C, Java, C#, PHP, Go, Ruby, Swift, Python, ils sont tous synchrones par défaut. Certains d'entre eux gèrent l'async en utilisant des threads, engendrant un nouveau processus.

JavaScript est synchrone par défaut et est monothread. Cela signifie que le code ne peut pas créer de nouveaux threads et s'exécuter en parallèle.

Callbacks

Les lignes de code sont exécutées en série, l'une après l'autre, par exemple :

```
const a = 1
const b = 2
const c = a * b
console.log(c)
doSomething()
```

Callbacks

Mais JavaScript est né à l'intérieur du navigateur, son travail principal, au début, était de répondre aux actions des utilisateurs, comme `onClick`, `onMouseOver`, `onChange`, `onSubmit` etc. Comment pourrait-il le faire avec un modèle de programmation synchrone ?

La réponse était dans son environnement. Le navigateur fournit un moyen de le faire en fournissant un ensemble d'API pouvant gérer ce type de fonctionnalité.

Plus récemment, Node.js a introduit un environnement non bloquant pour étendre ce concept à l'accès aux fichiers, aux appels réseau, etc.

Callbacks

Vous ne pouvez pas savoir quand un utilisateur va cliquer sur un bouton, vous définissez donc un gestionnaire d'événements pour l'événement `click` . Ce gestionnaire d'événement accepte une fonction, qui sera appelée lorsque l'événement est déclenché :

```
document.getElementById('button').addEventListener('click', () => {  
  //item clicked  
})
```

C'est ce qu'on appelle un **callback**

Callbacks

Un callback est une fonction simple qui est transmise en tant que valeur à une autre fonction et ne sera exécutée que lorsque l'événement se produira. Nous pouvons le faire car JavaScript a des fonctions de première classe, qui peuvent être affectées à des variables et transmises à d'autres fonctions (appelées **fonctions d'ordre supérieur**)

Il est courant d'encapsuler tout votre code client dans un **load eventListener** sur l'objet **window**, qui exécute la fonction de rappel uniquement lorsque la page est prête :

Callbacks

```
window.addEventListener('load', () => {  
  //window loaded  
  //fonctions à charger  
})
```

Callbacks

Les callbacks sont utilisés partout, pas seulement dans les événements DOM.

Un exemple courant consiste à utiliser des minuteries :

```
setTimeout(() => {  
  // se lance après 2 secondes  
}, 2000)
```

Gestion des erreurs dans les Callbacks

Comment gérer les erreurs avec les rappels ? Une stratégie très courante consiste à utiliser ce que Node.js a adopté : le premier paramètre de toute fonction de rappel est l'objet d'erreur **error-first callbacks**

S'il n'y a pas d'erreur, l'objet est null. S'il y a une erreur, il contient une description de l'erreur et d'autres informations.

Gestion des erreurs dans les Callbacks

```
fs.readFile('/file.json', (err, data) => {  
  if (err !== null) {  
    //handle error  
    console.log(err)  
    return  
  }  
  
  //no errors, process data  
  console.log(data)  
})
```


Le problème des Callbacks

Les rappels sont parfaits pour les cas simples !

Cependant chaque callback ajoute un niveau d'imbrication, et quand vous avez beaucoup de callbacks, le code commence à se compliquer très vite :

Le problème des Callbacks

4 niveaux d'imbrication

```
window.addEventListener('load', () => {  
  document.getElementById('button').addEventListener('click', () => {  
    setTimeout(() => {  
      items.forEach(item => {  
        //your code here  
      })  
    }, 2000)  
  })  
})
```

Alternative aux Callbacks

À partir de ES6, JavaScript a introduit plusieurs fonctionnalités qui nous aident avec du code asynchrone qui n'implique pas l'utilisation de rappels :

- Promises (ES6)
- Async/Await (ES8)

Promises

Les promesses sont un moyen de traiter le code asynchrone, sans écrire trop de rappels dans votre code.

Bien qu'ils existent depuis des années, ils ont été standardisés et introduits dans ES2015, et maintenant ils ont été remplacés dans ES2017 par des fonctions asynchrones.

Les fonctions asynchrones utilisent l'API des promesses comme bloc de construction, il est donc fondamental de les comprendre même si dans un code plus récent, vous utiliserez probablement des fonctions asynchrones au lieu de promesses.

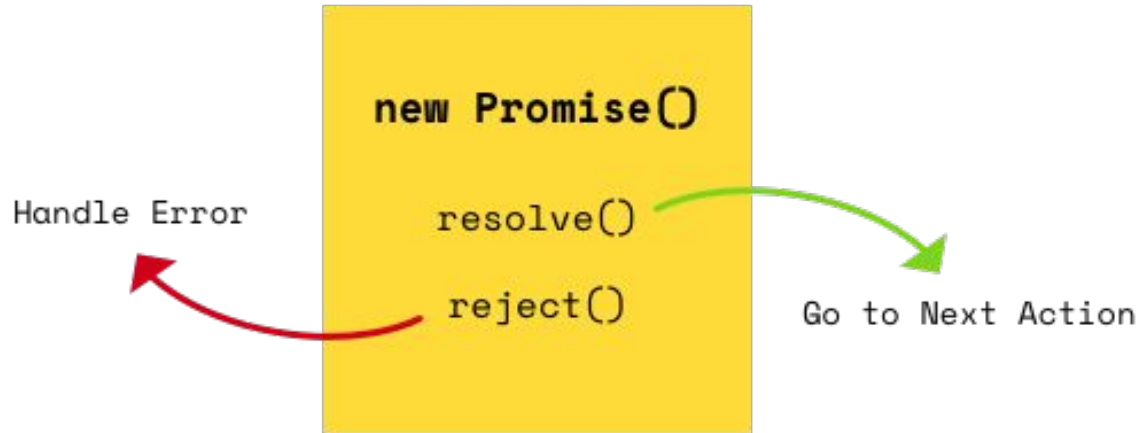
(Versions de JS : https://www.w3schools.com/Js/js_versions.asp)

Comment fonctionnent les Promesses ?

Une fois qu'une promesse a été appelée, elle démarre en état attente (**pending state**) . Cela signifie que la fonction appelante continue l'exécution, pendant qu'elle attend la promesse de faire son propre traitement, et donne à la fonction appelante un retour d'informations.

À ce stade, la fonction appelante attend qu'elle renvoie la promesse dans un état résolu , ou dans un état rejeté , mais comme vous le savez, JavaScript est asynchrone, donc la fonction continue son exécution pendant que la promesse fonctionne .

Comment fonctionnent les Promesses ?



Quelle API de JS utilise les Promesses ?

En plus de votre propre code et de votre code de bibliothèque, les promesses sont utilisées par les API Web modernes standard telles que Fetch.

Il est peu probable qu'en JavaScript moderne, vous n'utilisiez *pas de* promesses, nous allons pour cela les aborder.

Créer une Promesse ?

L'API Promise expose un constructeur Promise, que vous initialisez à l'aide de `new Promise()` :

Créer une Promesse ?

```
let done = true

const isItDoneYet = new Promise((resolve, reject) => {
  if (done) {
    const workDone = 'Here is the thing I built'
    resolve(workDone)
  } else {
    const why = 'Still working on something else'
    reject(why)
  }
})
```

Créer une Promesse ?

Comme vous pouvez le voir, la promesse vérifie la constante **done** globale, et si c'est vrai, nous retournons une promesse résolue, sinon une promesse rejetée.

À l'aide de **resolve** et **reject** nous pouvons communiquer en retour une valeur, dans le cas ci-dessus, nous renvoyons simplement une chaîne, mais cela pourrait également être un objet.

Utiliser une Promesse ?

```
const isItDoneYet = new Promise()  
//...  
  
const checkIfItsDone = () => {  
  isItDoneYet  
    .then(ok => {  
      console.log(ok)  
    })  
    .catch(err => {  
      console.error(err)  
    })  
}
```

Utiliser une Promesse ?

Le lancement de **checkIfItsDone()** exécutera la promesse **isItDoneYet()** et attendra qu'elle se résolve, en utilisant le callback **then**, et s'il y a une erreur, il la traitera dans le callback **catch**.

Enchaîner les Promesses ?

Une promesse peut être retournée à une autre promesse, créant une chaîne de promesses.

Un excellent exemple de chaînage de promesses est donné par l' API Fetch, une couche au-dessus de l'API XMLHttpRequest, que nous pouvons utiliser pour obtenir une ressource et mettre en file d'attente une chaîne de promesses à exécuter lorsque la ressource est récupérée.

L'API Fetch est un mécanisme basé sur la promesse, et l'appel `fetch()` équivaut à définir notre propre promesse en utilisant `new Promise()`.

```
const status = response => {  
  if (response.status >= 200 && response.status < 300) {  
    return Promise.resolve(response)  
  }  
  return Promise.reject(new Error(response.statusText))  
}
```

```
const json = response => response.json()
```

```
fetch('/todos.json')  
  .then(status)  
  .then(json)  
  .then(data => {  
    console.log('Request succeeded with JSON response', data)  
  })  
  .catch(error => {  
    console.log('Request failed', error)  
  })
```

Enchaîner les Promesses ?

Dans cet exemple, nous appelons **fetch()** pour obtenir une liste des éléments TODO du fichier **todos.json** trouvé à la racine du domaine, et nous créons une chaîne de promesses.

Lancer **fetch()** renvoie une [réponse](#), qui a de nombreuses propriétés, et parmi celles que nous référençons :

- **status**, une valeur numérique représentant le code d'état HTTP
- **statusText**, un message d'état, qui est OK si la demande aboutit

Enchaîner les Promesses ?

- **response** a également une méthode `json()`, qui renvoie une promesse qui se résoudra avec le contenu du corps traité et transformé en JSON.
- Donc, selon ces promesses, voici ce qui se passe : la première promesse de la chaîne est une fonction que nous avons définie, appelée **status()**, qui vérifie l'état de la réponse et s'il ne s'agit pas d'une réponse réussie (entre 200 et 299), il rejette la promesse.
- Cette opération fera en sorte que la chaîne de promesses **ignorera** toutes les promesses enchaînées répertoriées et passera directement au **catch()** en bas, en enregistrant le texte **Request failed** accompagné du message d'erreur.

Enchaîner les Promesses ?

Si cela réussit, il appelle la fonction **json()** que nous avons définie. Étant donné que la promesse précédente, une fois réussie, renvoyait l'objet `response`, nous l'obtenons en entrée de la deuxième promesse.

Dans ce cas, nous renvoyons les données JSON traitées, donc la troisième promesse reçoit directement le JSON :

```
.then((data) => {  
    console.log('Request succeeded with JSON response', data)  
})
```

Gestion des erreurs

Dans l'exemple ci-dessus, dans la section précédente, nous avons eu un `catch` qui était annexé à la chaîne des promesses.

Lorsqu'un élément de la chaîne de promesses échoue et génère une erreur ou rejette la promesse, le contrôle passe à la déclaration `catch()` la plus proche en bas de la chaîne.

Gestion des erreurs

```
new Promise((resolve, reject) => {  
  throw new Error('Error')  
}).catch(err => {  
  console.error(err)  
})
```

// or

```
new Promise((resolve, reject) => {  
  reject('Error')  
}).catch(err => {  
  console.error(err)  
})
```

Erreurs en cascade

Si à l'intérieur du `catch()` une erreur est levée, on peut en ajouter une seconde pour gérer le premier, et ainsi de suite.

```
new Promise((resolve, reject) => {  
  throw new Error('Error')  
})  
  .catch(err => {  
    throw new Error('Error')  
  })  
  .catch(err => {  
    console.error(err)  
  })
```

Orchestrer les promesses avec `Promise.all()`

Si vous devez synchroniser différentes promesses, `Promise.all()` aide à définir une liste de promesses et à exécuter quelque chose lorsqu'elles sont toutes résolues.

Orchestrer les promesses avec `Promise.all()`

```
const f1 = fetch('/something.json')
const f2 = fetch('/something2.json')

Promise.all([f1, f2])
  .then(res => {
    console.log('Array of results', res)
  })
  .catch(err => {
    console.error(err)
  })
```

Orchestrer les promesses avec `Promise.all()`

La syntaxe d'affectation de déstructuration ES2015 vous permet également de le faire

```
Promise.all([f1, f2]).then(([res1, res2]) => {  
  console.log('Results', res1, res2)  
}))
```

Orchestrer les promesses avec `Promise.race()`

`Promise.race()` s'exécute dès que l'une des promesses que vous lui transmettez est résolue, et il exécute le callback qui y est joint avec le résultat de la dernière promesse résolue.

Orchestrer les promesses avec `Promise.race()`

```
const promiseOne = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'one')
})
const promiseTwo = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'two')
})

Promise.race([promiseOne, promiseTwo]).then(result => {
  console.log(result) // 'two'
})
```

Async/Await

JavaScript a évolué en très peu de temps des rappels aux promesses (ES2015), et depuis ES2017, le JavaScript asynchrone est encore plus simple avec la syntaxe `async/await`.

Les fonctions asynchrones sont une combinaison de promesses et de générateurs, et fondamentalement, elles constituent une abstraction de niveau supérieur par rapport aux promesses. Attention : **`async/await` est juste construit sur des promesses** .

Pourquoi Async/Await a été introduit ?

Ils réduisent le passe-partout autour des promesses et la limitation « ne pas rompre la chaîne » des promesses enchaînées.

Lorsque les promesses ont été introduites dans ES2015, elles étaient destinées à résoudre un problème de code asynchrone, et elles l'ont fait, mais au cours des 2 années qui ont séparé ES2015 et ES2017, il était clair que les *promesses ne pouvaient pas être la solution finale* .

Des promesses ont été introduites pour résoudre le fameux problème nommé *callback hell*, mais elles ont introduit une complexité par leur nature aussi et leur syntaxe.

Pourquoi Async/Await a été introduit ?

C'étaient de bonnes primitives autour desquelles une meilleure syntaxe pouvait être exposée aux développeurs, donc quand le moment était venu, nous avons eu des fonctions asynchrones .

Ils donnent l'impression que le code est synchrone, mais il est asynchrone et non bloquant dans les coulisses.

Comment ça fonctionne ?

Une fonction async renvoie une promesse :

```
const doSomethingAsync = () => {  
  return new Promise(resolve => {  
    setTimeout(() => resolve('Fin du process'), 3000)  
  })  
}
```

Comment ça fonctionne ?

Lorsque vous souhaitez appeler cette fonction, vous ajoutez **await**, et le code appelant s'arrêtera jusqu'à ce que la promesse soit résolue ou rejetée .



Remarque : la fonction client doit être définie comme `async`. Voici un exemple :

```
const doSomething = async () => {  
  console.log(await doSomethingAsync())  
}
```

Testons ce code

```
const doSomethingAsync = () => {  
  return new Promise(resolve => {  
    setTimeout(() => resolve('I did something'), 3000)  
  })  
}  
  
const doSomething = async () => {  
  console.log(await doSomethingAsync())  
}  
  
console.log('Before')  
doSomething()  
console.log('After')
```

Le code est plus simple à lire

Comme vous pouvez le voir dans l'exemple ci-dessus, ce code semble plus simple. Comparez-le au code utilisant des promesses simples, avec des fonctions de chaînage et de rappel.

Et c'est un exemple très simple, les principaux avantages se produiront lorsque le code est beaucoup plus complexe.

Par exemple, voici comment obtenir une ressource JSON et l'analyser à l'aide de promesses :

Le code est plus simple à lire

```
const getFirstUserData = () => {  
  return fetch('/users.json') // get users list  
    .then(response => response.json()) // parse JSON  
    .then(users => users[0]) // pick first user  
    .then(user => fetch(`/users/${user.name}`)) // get user data  
    .then(userResponse => userResponse.json()) // parse JSON  
}  
  
getFirstUserData()
```

Le code est plus simple à lire

```
const getFirstUserData = async () => {  
  const response = await fetch('/users.json') // get users list  
  const users = await response.json() // parse JSON  
  const user = await users[0] // pick first user  
  const userResponse = await fetch(`/users/${user.name}`) // get  
user data  
  const userData = await userResponse.json() // parse JSON  
  return userData  
}  
  
getFirstUserData()
```

Plusieurs fonctions asynchrones en série

Les fonctions asynchrones peuvent être enchaînées très facilement, et la syntaxe est bien plus lisible qu'avec des promesses : testons le code suivant

```
const promiseToDoSomething = () => {  
  return new Promise(resolve => {  
    setTimeout(() => resolve('I did something'), 10000)  
  })  
}
```

```
const watchOverSomeoneDoingSomething = async () => {  
  const something = await promiseToDoSomething()  
  return something + ' and I watched'  
}
```

```
const watchOverSomeoneWatchingSomeoneDoingSomething = async () => {  
  const something = await watchOverSomeoneDoingSomething()  
  return something + ' and I watched as well'  
}
```

```
watchOverSomeoneWatchingSomeoneDoingSomething().then(res => {  
  console.log(res)  
})
```

Débogage plus facile

Le débogage des promesses est difficile car le débogueur ne franchira pas le code asynchrone.

Async/await rend cela très facile car pour le compilateur, c'est comme du code synchrone.

Module ES

ES Modules est la norme ECMAScript pour travailler avec des modules.

Alors que Node.js utilise la norme CommonJS depuis des années, le navigateur n'a jamais eu de système de module, car chaque décision importante telle qu'un système de module doit d'abord être standardisée par ECMAScript puis implémentée par le navigateur.

Ce processus de normalisation achevé avec ES6 et les navigateurs, cette norme a été implémentée, fonctionnant de la même manière, et maintenant les modules ES sont pris en charge dans Chrome, Safari, Edge et Firefox (depuis la version 60).

Module ES

Les modules sont très intéressants, car ils vous permettent d'encapsuler toutes sortes de fonctionnalités et d'exposer ces fonctionnalités à d'autres fichiers JavaScript, en tant que bibliothèques.

Syntaxe des module ES

La syntaxe pour importer un module est :

```
import package from 'module-name'
```

tandis que CommonJS utilise :

```
const package = require('module-name')
```


Syntaxe des module ES

Un module est un fichier JavaScript qui **exporte** une ou plusieurs valeurs (objets, fonctions ou variables), en utilisant le mot-clé **export**. Par exemple, ce module exporte une fonction qui renvoie une chaîne en majuscule :

majuscule.js

```
export default str => str.toUpperCase()
```

Syntaxe des module ES

Dans cet exemple, le module définit un seul **export par défaut** , il peut donc s'agir d'une fonction anonyme. Sinon, il faudrait un nom pour le distinguer des autres exportations.

Désormais, **tout autre module JavaScript** peut importer la fonctionnalité offerte par majuscule.js en l'important.

Une page HTML peut ajouter un module en utilisant un tag **<script>** avec l'attribut **type="module"** :

Syntaxe des module ES

Il est important de noter que tout script chargé avec `type="module"` est chargé en [mode strict](#).

Dans cet exemple, le module `majuscule.js` définit une **exportation par défaut**, donc lorsque nous l'importons, nous pouvons lui attribuer un nom que nous préférons :

```
import toUpperCase from './uppercase.js'
```

```
toUpperCase('test') //'TEST'
```

Syntaxe des module ES

Vous pouvez également utiliser un chemin absolu pour l'import de module, pour référencer des modules définis sur un autre domaine :

```
import toUpperCase from  
'https://stock-of-modules-example.glitch.me/majuscule.js'
```

```
import { foo } from '/majuscule.js'  
import { foo } from '../majuscule.js'
```

Autres options d'import/export

Nous avons vu cet exemple :

```
export default str => str.toUpperCase()
```

Cela crée une exportation par défaut. Dans un fichier, cependant, vous pouvez exporter plus d'une chose, en utilisant cette syntaxe :

```
const a = 1  
const b = 2  
const c = 3  
  
export { a, b, c }
```

Autres options d'import/export

Puis importer toutes ces fonctionnalités en utilisant :

```
import * from 'module'
```

Vous pouvez importer seulement quelques-unes de ces exportations, en utilisant l'affectation de déstructuration :

```
import { a } from 'module'  
import { a, b } from 'module'
```

Autres options d'import/export

Vous pouvez renommer n'importe quelle importation, plus facilement, avec **as** :

```
import { a, b as two } from 'module'
```

Vous pouvez importer l'exportation par défaut et toute exportation qui n'est pas par défaut par son nom, comme dans cette importation React :

```
import React, { Component } from 'react'
```

CORS

Les modules sont récupérés à l'aide de CORS. Cela signifie que si vous référencez des scripts d'autres domaines, ils doivent avoir un en-tête CORS valide qui permet le chargement entre sites (comme Access-Control-Allow-Origin: *)

```
res.setHeader("Access-Control-Allow-Origin", "*");
```


Concepts React : Single Page Applications

Aujourd'hui, popularisée par les frameworks JavaScript frontend modernes comme React, une application est généralement construite comme une application à page unique : vous ne chargez le code de l'application (HTML, CSS , JavaScript) qu'une seule fois, et lorsque vous interagissez avec l'application, ce qui se passe généralement est que JavaScript intercepte les événements du navigateur et au lieu de faire une nouvelle demande au serveur qui renvoie ensuite un nouveau document, le client demande du JSON ou effectue une action sur le serveur mais la page que l'utilisateur voit n'est jamais complètement effacée et se comporte plus comme une application de bureau.

Concepts React : Single Page Applications

Les applications à page unique sont construites en JavaScript (ou au moins compilées en JavaScript) et fonctionnent dans le navigateur.

La technologie est toujours la même, mais la philosophie et certains éléments clés du fonctionnement de l'application sont différents.

Concepts React : Single Page Applications

Quelques exemples :

- Gmail
- Google Maps
- Facebook
- Twitter
- Google Drive

Avantages et Inconvénients des SPA

- Une SPA est beaucoup plus rapide pour l'utilisateur, car au lieu d'attendre que la communication client-serveur se produise et que le navigateur restitue la page, vous pouvez désormais avoir un retour instantané. C'est la responsabilité du créateur de l'application, mais vous pouvez avoir des transitions et des spinners et tout type d'amélioration UX qui sont certainement meilleures que le flux de travail traditionnel.
- En plus de rendre l'expérience plus rapide pour l'utilisateur, le serveur consommera moins de ressources car vous pouvez vous concentrer sur la fourniture d'une API efficace au lieu de créer les mises en page côté serveur.

Avantages et Inconvénients des SPA

Cela le rend idéal si vous créez également une application mobile au-dessus de l'API, car vous pouvez réutiliser complètement votre code côté serveur existant.

Les applications à page unique sont faciles à transformer en applications Web progressives, ce qui vous permet à son tour de fournir une mise en cache locale et de prendre en charge des expériences hors ligne pour vos services

Avantages et Inconvénients des SPA

Les SPA sont mieux utilisées lorsqu'il n'y a pas besoin de référencement (optimisation pour les moteurs de recherche). Par exemple pour les applications qui fonctionnent derrière une connexion.

Les moteurs de recherche, tout en s'améliorant chaque jour, ont toujours du mal à indexer les sites construits avec une approche SPA plutôt que les pages traditionnelles rendues par le serveur. C'est le cas des blogs. Si vous comptez sur les moteurs de recherche, ne vous embêtez même pas à créer une application d'une seule page sans avoir également une partie rendue par le serveur.

Avantages et Inconvénients des SPA

Lors du codage d'un SPA, vous allez écrire beaucoup de JavaScript. Étant donné que l'application peut durer longtemps, vous devrez faire beaucoup plus attention aux fuites de mémoire possibles. Cela peut provoquer un problème de mémoire qui va beaucoup plus augmenter l'utilisation de la mémoire du navigateur et cela va provoquer une expérience désagréablement lente si vous n'en prenez pas soin.

Les SPA sont parfaits pour travailler en équipe. Les développeurs backend peuvent se concentrer uniquement sur l'API, et les développeurs frontend peuvent se concentrer sur la création de la meilleure expérience utilisateur, en utilisant l'API intégrée au backend.

Avantages et Inconvénients des SPA

Par contre, les applications à page unique reposent fortement sur JavaScript. Cela peut rendre l'utilisation d'une application exécutée sur des appareils à faible consommation d'énergie une mauvaise expérience en termes de vitesse.

En outre, certains de vos visiteurs peuvent simplement désactiver JavaScript, et vous devez également prendre en compte l'accessibilité pour tout ce que vous créez.

Passer outre la navigation

Puisque vous vous débarrassez de la navigation par défaut du navigateur, les URL doivent être gérées manuellement.

Cette partie d'une application s'appelle le routeur. Certains frameworks s'en chargent déjà pour vous (comme Ember), d'autres nécessitent des bibliothèques qui feront ce travail (comme React Router).

Passer outre la navigation

Quel est le problème? Au début, il s'agissait d'une réflexion après coup pour les développeurs créant des applications à page unique.

Cela a causé le problème courant du « bouton de retour cassé » : lors de la navigation dans l'application, l'URL n'a pas changé (puisque la navigation par défaut du navigateur a été piratée) et en appuyant sur le bouton de retour, une opération courante que les utilisateurs effectuent pour accéder à l'écran précédent, peut être déplacé vers un site Web que vous avez visité il y a longtemps.

Passer outre la navigation

Ce problème peut désormais être résolu à l'aide de l'API History proposée par les navigateurs, mais la plupart du temps, vous utiliserez une bibliothèque qui utilise en interne cette API, comme **React Router**

Déclaratif

Qu'est-ce que cela signifie lorsque vous lisez que React est déclaratif ? Vous rencontrerez des articles décrivant React comme une **approche déclarative de la création d'interfaces utilisateur** .

React a rendu son «approche déclarative» assez populaire et directe, de sorte qu'elle a pénétré le monde frontal avec React.

Déclaratif

Ce n'est vraiment pas un nouveau concept, mais React a pris la construction d'interfaces utilisateur de manière beaucoup plus déclarative qu'avec les modèles HTML :

- vous pouvez créer des interfaces Web sans même toucher directement le DOM
- vous pouvez avoir un système d'événements sans avoir à interagir avec les événements DOM réels.

Déclaratif

- Le contraire de déclaratif est **impératif** . Un exemple courant d'approche impérative consiste à rechercher des éléments dans le DOM à l'aide d'événements jQuery ou DOM. Vous dites exactement au navigateur ce qu'il doit faire, au lieu de lui dire ce dont vous avez besoin.
- L'approche déclarative de React fait abstraction de cela pour nous. Nous disons simplement à React que nous voulons qu'un composant soit rendu d'une manière spécifique, et nous n'avons jamais à interagir avec le DOM pour le référencer plus tard.

Immuabilité

Un concept que vous rencontrerez probablement lors de la programmation dans React est l'immuabilité.

C'est un sujet controversé, mais quoi que vous pensiez du concept d'immuabilité, React et la plupart de son écosystème l'utilise, vous devez donc au moins comprendre pourquoi c'est si important et ses implications.

En programmation, une variable est immuable lorsque sa valeur ne peut pas changer après sa création.

Immuabilité

En programmation, une variable est immuable lorsque sa valeur ne peut pas changer après sa création.

Vous utilisez déjà des variables immuables sans le savoir lorsque vous manipulez une chaîne. Les chaînes sont immuables par défaut, lorsque vous les modifiez en réalité, vous créez une nouvelle chaîne et l'affectez au même nom de variable.

Une variable immuable ne peut jamais être modifiée. Pour mettre à jour sa valeur, vous créez une nouvelle variable.

Immuabilité

Il en va de même pour les objets et les tableaux.

Au lieu de modifier un tableau, pour ajouter un nouvel élément, vous créez un nouveau tableau en concaténant l'ancien tableau et le nouvel élément.

Un objet n'est jamais mis à jour, mais copié avant de le modifier.

Cela s'applique à React dans de nombreux endroits.

Par exemple, vous ne devez jamais muter la propriété **state** d'un composant directement, mais seulement à travers la méthode **setState()**.

Immuabilité : Redux ne mute pas les états

Il utilise uniquement des réducteurs (des fonctions). Gérer notre état sans redux ou alternatives peut être difficile. Imaginez que nous devions vérifier sur chaque composant si l'utilisateur est authentifié ou non. Pour gérer ce cas d'utilisation, nous devons passer des props à travers chaque composant et suite à la croissance de l'application, il est tout simplement impossible de gérer notre état de cette manière. Et c'est là que redux entre en scène.

Redux est une bibliothèque indépendante qui nous aide à gérer notre état en donnant à nos composants l'état dont il a besoin via un store central. Redux stocke tout l'état de notre application dans une arborescence d'objets immuable.

Pourquoi Redux ?

Il y a plusieurs raisons dont les plus importantes sont :

- Les mutations peuvent être centralisées ce qui améliore vos capacités de débogage et réduit les sources d'erreurs.
- Le code semble plus propre et plus simple à comprendre. Vous ne vous attendez jamais à ce qu'une fonction change une valeur sans que vous le sachiez, ce qui vous donne une **prévisibilité**. Lorsqu'une fonction ne mute pas d'objets mais renvoie simplement un nouvel objet, cela s'appelle une fonction pure.
- La bibliothèque peut optimiser le code car, par exemple, JavaScript est plus rapide lors de l'échange d'une ancienne référence d'objet contre un objet entièrement nouveau, plutôt que de muter un objet existant. Cela vous améliore les **performances**

Pureté

En JavaScript, lorsqu'une fonction ne mute pas d'objets mais renvoie simplement un nouvel objet, cela s'appelle une fonction pure.

Une fonction ou une méthode, pour être appelée *pure* , ne devrait pas provoquer d'effets secondaires et devrait renvoyer la même sortie lorsqu'elle est appelée plusieurs fois avec la même entrée.

Une fonction pure prend une entrée et renvoie une sortie sans changer l'entrée ni rien d'autre.

Pureté

Sa sortie n'est déterminée que par les arguments. Vous pouvez appeler cette fonction 1M fois, et étant donné le même ensemble d'arguments, la sortie sera toujours la même.

React applique ce concept aux composants. Un composant React est un composant pur lorsque sa sortie ne dépend que de ses props.

Tous les composants fonctionnels sont des composants purs :

```
const Button = props => {  
  return <button>{props.message}</button>  
}
```

Pureté

Les composants de classe peuvent être purs si leur sortie ne dépend que des props :

```
class Button extends React.Component {  
  render() {  
    return <button>{this.props.message}</button>  
  }  
}
```

Composition

En programmation, la composition vous permet de créer des fonctionnalités plus complexes en combinant des fonctions petites et ciblées.

Par exemple, pensez à utiliser `map()` pour créer un nouveau tableau à partir d'un ensemble initial, puis filtrer le résultat à l'aide `filter()`:

```
const list = ['Apple', 'Orange', 'Egg']  
list.map(item => item[0]).filter(item => item === 'A') //'A'
```

Composition

Dans React, la composition vous permet d'avoir des avantages assez intéressants.

Vous créez des composants petits et légers et les utilisez pour *composer* plus de fonctionnalités par-dessus. Comment?

Passer des méthodes en tant que props

Un composant peut se concentrer sur le suivi d'un événement de clic, par exemple, et sur ce qui se passe réellement lorsque l'événement de clic se produit dépendra du composant du conteneur :

```
const Button = props => {
  return <button onClick={props.onClickHandler}>{props.text}</button>
}

const LoginButton = props => {
  return <Button text="Login" onClickHandler={props.onClickHandler} />
}

const Container = () => {
  const onClickHandler = () => {
    alert('clicked')
  }

  return <LoginButton onClickHandler={onClickHandler} />
}
```

Propriétés `props.children`

Les propriétés `props.children` vous permettent d'injecter des composants à l'intérieur d'autres composants.

Le composant doit sortir `props.children` dans son JSX.

```
const Sidebar = props => {  
  return <aside>{props.children}</aside>  
}
```

Propriétés `props.children`

Puis vous intégrez les composants directement :

```
<Sidebar>  
  <Link title="First link" />  
  <Link title="Second link" />  
</Sidebar>
```

Composants d'ordre supérieur

Lorsqu'un composant reçoit un composant en tant que props et renvoie un composant, il est appelé composant d'ordre supérieur.

Nous les verrons par la suite.

DOM virtuel

De nombreux frameworks existants, avant l'arrivée de React, manipulaient directement le DOM à chaque changement.

Tout d'abord, qu'est-ce que le DOM ?

Le DOM (*Document Object Model*) est une représentation arborescente de la page, à partir du `<html>`, descendant dans chaque enfant, qui sont appelés **nodes**.

Il est conservé dans la mémoire du navigateur et directement lié à ce que vous voyez dans une page. Le DOM a une API que vous pouvez utiliser pour le parcourir, accéder à chaque nœud, les filtrer, les modifier.

DOM virtuel

L'API est la syntaxe familière que vous avez probablement vue plusieurs fois, si vous n'utilisiez pas l'API abstraite fournie par jQuery :

```
document.getElementById(id)
document.getElementsByTagName(name)
document.createElement(name)
parentNode.appendChild(node)
element.innerHTML
element.style.left
element.setAttribute()
element.getAttribute()
element.addEventListener()
```

DOM virtuel

React conserve une copie de la représentation DOM, pour ce qui concerne le rendu React : le DOM virtuel

Chaque fois que le DOM change, le navigateur doit effectuer deux opérations intensives : **repaint** (changements visuels ou de contenu d'un élément qui n'affectent pas la mise en page et le positionnement par rapport aux autres éléments) et **reflow** (recalculer la mise en page d'une partie de la page – ou la mise en page entière).

React utilise un DOM virtuel pour aider le navigateur à utiliser moins de ressources lorsque des modifications doivent être apportées à une page.

DOM virtuel

Quand vous appelez **setState()** sur un composant, en spécifiant un état différent du précédent, React marque ce composant comme **dirty** . C'est la clé : React ne se met à jour que lorsqu'un composant change explicitement d'état.

Ce qui se passe ensuite est :

- React met à jour le DOM virtuel par rapport aux composants marqués **dirty** (avec quelques vérifications supplémentaires, comme le déclenchement **shouldComponentUpdate()**)
- Exécute l'algorithme de différence pour réconcilier les changements
- Met à jour le vrai DOM

Utilité du DOM virtuel : les lots

L'essentiel est que React regroupe une grande partie des modifications et effectue une mise à jour unique du vrai DOM, en modifiant tous les éléments qui doivent être modifiés en même temps. Exécution en une seule fois.

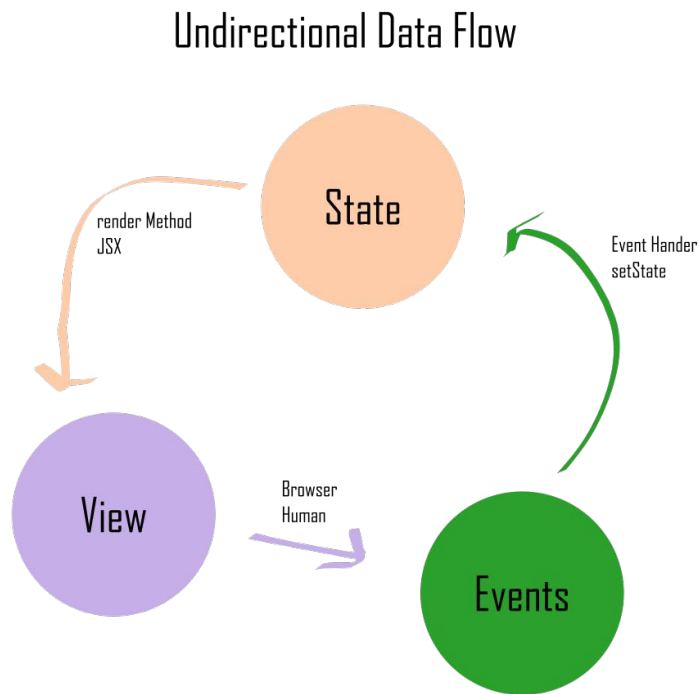
Flux de données unidirectionnel

En travaillant avec React, vous pourriez rencontrer le terme flux de données unidirectionnel. Qu'est-ce que ça veut dire?

Le flux de données unidirectionnel n'est pas un concept unique à React, mais en tant que développeur JavaScript, c'est peut-être la première fois que vous l'entendez.

En général, ce concept signifie que les données ont un, et un seul, moyen d'être transférées vers d'autres parties de l'application.

Flux de données unidirectionnel



Flux de données unidirectionnel

Dans React, cela signifie que :

- l'état est passé à la vue et aux composants enfants
- les actions sont déclenchées par la vue
- les actions peuvent mettre à jour l'état
- le changement d'état est passé à la vue et aux composants enfants

La vue est le résultat de l'état de l'application. L'état ne peut changer que lorsque des actions se produisent. Lorsque des actions se produisent, l'état est mis à jour.

Flux de données unidirectionnel

Grâce aux liaisons unidirectionnelles, les données ne peuvent pas circuler dans le sens inverse (comme cela se produirait avec les liaisons bidirectionnelles, par exemple), et cela présente certains avantages clés :

- c'est moins sujet aux erreurs, car vous avez plus de contrôle sur vos données
- c'est plus facile à déboguer, car vous savez ce *qui* vient d'*où*
- c'est plus efficace, car la bibliothèque sait déjà quelles sont les limites de chaque partie du système

Flux de données unidirectionnel

Un état appartient toujours à un composant. Toutes les données affectées par cet état ne peuvent affecter que les composants inférieurs : ses enfants.

Changer l'état d'un composant n'affectera jamais son parent, ses frères ou tout autre composant de l'application : uniquement ses enfants.

C'est la raison pour laquelle l'état est souvent déplacé vers le haut dans l'arborescence des composants, afin qu'il puisse être partagé entre les composants qui doivent y accéder.

JSX dans React

JSX est une technologie introduite par React.

Bien que React puisse fonctionner parfaitement sans utiliser JSX, c'est une technologie idéale pour travailler avec des composants, donc React profite beaucoup de JSX.

Au début, vous pourriez penser qu'utiliser JSX revient à mélanger HTML et JavaScript (et comme vous le verrez CSS).

Mais ce n'est pas vrai, car ce que vous faites réellement lorsque vous utilisez la syntaxe JSX, c'est d'écrire une syntaxe déclarative de ce que devrait être une interface utilisateur de composant.

JSX dans React

Et vous décrivez cette interface utilisateur sans utiliser de chaînes, mais en utilisant JavaScript, ce qui vous permet de faire beaucoup de choses intéressantes.

JSX dans React

Voici comment définir une balise h1 contenant une chaîne :

```
const element = <h1>Hello, world!</h1>
```

Cela ressemble à un étrange mélange de JavaScript et de HTML, mais en réalité, il s'agit uniquement de JavaScript.

Ce qui ressemble à du HTML, est en fait du sucre syntaxique pour définir les composants et leur positionnement à l'intérieur du balisage.

JSX dans React

A l'intérieur d'une expression JSX, les attributs peuvent être insérés très facilement :

```
const myId = 'test'  
const element = <h1 id={myId}>Hello, world!</h1>
```

JSX dans React

Voici un extrait JSX qui encapsule dans une **div** deux composants

```
<div>
  <BlogPostsList />
  <Sidebar />
</div>
```

Remarquez comment les 2 composants sont dans un div. Pourquoi? Parce que la fonction `render()` ne peut renvoyer qu'un seul node, donc au cas où vous voudriez renvoyer 2 frères et sœurs, ajoutez simplement un parent. Il peut s'agir de n'importe quelle balise, pas seulement div.

Transpiler JSX

Un navigateur ne peut pas exécuter de fichiers JavaScript contenant du code JSX. Ils doivent d'abord être transformés en JS standard.

Comment? En effectuant un processus appelé **transpilation** .

Nous avons déjà dit que JSX est facultatif, car pour chaque ligne JSX, une alternative JavaScript simple correspondante est disponible, et c'est vers cela que JSX est **transpilé**.

Par exemple, les deux constructions suivantes sont équivalentes

Transpiler JSX

```
ReactDOM.render(  
  React.DOM.div(  
    { id: 'test' },  
    React.DOM.h1(null, 'A title'),  
    React.DOM.p(null, 'A paragraph')  
  ),  
  document.getElementById('myapp')  
)
```

JS Pur

Transpiler JSX

```
ReactDOM.render(  
  <div id="test">  
    <h1>A title</h1>  
    <p>A paragraph</p>  
  </div>,  
  document.getElementById('myapp')  
)
```

JSX

Transpiler JSX

Cet exemple très basique n'est que le point de départ, mais vous pouvez déjà voir à quel point la syntaxe JS simple est plus compliquée par rapport à l'utilisation de JSX.

Au moment d'écrire ces lignes, le moyen le plus d'effectuer la **transpilation** consiste à utiliser **Babel**, qui est l'option par défaut lors de l'exécution **create-react-app**, donc si vous l'utilisez, ne vous inquiétez pas, tout se passe sous le capot pour vous.

Si vous n'utilisez pas **create-react-app** vous devez configurer Babel vous-même.

JS dans JSX

JSX accepte tout type de JavaScript qui y est mélangé.

Chaque fois que vous avez besoin d'ajouter du JS, placez-le simplement entre des accolades `{}`. Par exemple, voici comment utiliser une valeur constante définie ailleurs :

```
const paragraph = 'A paragraph'
ReactDOM.render(
  <div id="test">
    <h1>A title</h1>
    <p>{paragraph}</p>
  </div>,
  document.getElementById('myapp')
)
```

JS dans JSX

Les accolades peuvent contenir n'importe quel code JS :

```
const paragraph = 'A paragraph'  
ReactDOM.render(  
  <table>  
    {rows.map((row, i) => {  
      return <tr>{row.text}</tr>  
    })}  
  </table>,  
  document.getElementById('myapp')  
)
```

**JS imbriqué
dans JSX
lui-même
imbriqué dans
JS puis
imbriqué dans
JSX**

HTML dans JSX

JSX ressemble beaucoup à HTML, mais c'est en fait de la syntaxe XML.

En fin de compte, vous rendez du HTML, nous devons donc connaître quelques différences entre la façon dont vous définiriez certaines choses en HTML et la façon dont vous les définiriez dans JSX.

Fermer toutes les balises

Tout comme en XHTML, si vous l'avez déjà utilisé, vous devez fermer toutes les balises : plus de `
` mais plutôt la balise à fermeture automatique : `
`

(même chose pour les autres balises)

camelCase est le nouveau standard

En HTML, vous trouverez des attributs sans aucune casse (par exemple `onchange`). Dans JSX, ils sont renommés en leur équivalent `camelCase` :

- `onchange` => `onChange`
- `onclick` => `onClick`
- `onsubmit` => `onSubmit`

class devient className

Comme à l'origine, JSX est du JS, **class** est un mot réservé.

```
<p class="description">
```

```
<p className="description">
```

```
for
```

```
htmlFor
```

CSS dans React

- JSX fournit un bon moyen de définir CSS.
- Si vous avez un peu d'expérience avec les styles HTML en ligne, à première vue, vous vous retrouverez 10 ou 15 ans en arrière, dans un monde où le CSS en ligne était tout à fait normal (de nos jours, il est diabolisé et n'est généralement qu'une "solution rapide").
- Le style JSX n'est pas pareil : tout d'abord, au lieu d'accepter une chaîne contenant des propriétés CSS, le JSX n'accepte qu'un objet.

CSS dans React

Cela signifie que vous définissez des propriétés dans un objet

```
var divStyle = {  
  color: 'white'  
}
```

```
ReactDOM.render(<div style={divStyle}>Hello World!</div>, mountNode)
```

```
ReactDOM.render(<div style={{ color: 'white' }}>Hello World!</div>,  
mountNode)
```


CSS dans React

Les valeurs CSS que vous écrivez dans JSX sont légèrement différentes du CSS simple :

- les noms des propriétés des clés sont **camelCased**
- les valeurs ne sont que des chaînes
- vous séparez chaque tuple par une virgule

Pourquoi préférer JSX aux CSS/SASS/LESS ?

CSS est un **problème non résolu** . Depuis sa création, des dizaines d'outils autour de lui sont montés puis sont tombés. Le principal problème avec JS est qu'il n'y a pas de portée et qu'il est facile d'écrire du CSS qui n'est en aucun cas appliqué, donc une « correction rapide » peut avoir un impact sur les éléments qui ne doivent pas être touchés.

JSX permet aux composants (définis dans React par exemple) d'encapsuler complètement leur style.

Est-ce une solution incontournable ?

Les styles en ligne dans JSX sont bons jusqu'à ce que vous ayez besoin de

1. écrire des requêtes multimédias
2. animations de style
3. des pseudo-classes de référence (par exemple `:hover`)
4. des pseudo-éléments de référence (par exemple `::first-letter`)

Bref, ils couvrent les bases, mais ce n'est pas la solution finale.

Formulaires dans JSX

JSX ajoute quelques modifications au fonctionnement des formulaires HTML, dans le but de faciliter les choses pour le développeur.

value et **defaultValue**

L'attribut **value** contient toujours la valeur actuelle du champ.

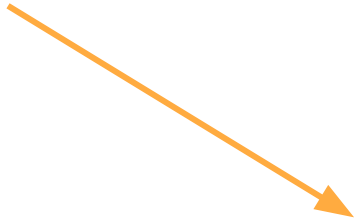
L'attribut **defaultValue** contient la valeur par défaut définie lors de la création du champ.

*Cela aide à résoudre certains comportements étranges de l'interaction régulière du [DOM](#) lors de l'inspection **input.value** et **input.getAttribute('value')** retournant un la valeur actuelle et un la valeur par défaut d'origine.*

Formulaires dans JSX

Cela s'applique également au champ **textarea**

```
<textarea>Some text</textarea>
```

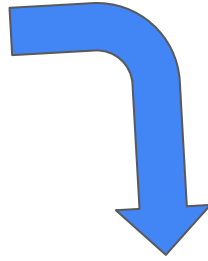


```
<textarea defaultValue={'Some text'} />
```

Formulaires dans JSX

Et pour le champ **select**

```
<select>  
  <option value="x" selected>  
    ...  
  </option>  
</select>
```



```
<select defaultValue="x">  
  <option value="x">...</option>  
</select>
```

onChange plus cohérent

Pour passer une fonction à l'attribut **onChange**, vous pouvez vous connecter à des événements sur les champs de formulaire.

Il fonctionne de manière cohérente dans tous les champs, même **radio**, **select** et **checkbox** peuvent déclencher un événement **onChange**.

onChange se déclenche également lors de la saisie d'un caractère dans les champs **input** ou **textarea**.

Valeur des attributs

Dans JSX, une opération courante consiste à attribuer des valeurs aux attributs.

Manuellement :

```
<div>  
  <BlogPost title={data.title} date={data.date} />  
</div>
```

Avec un spread :

```
<div>  
  <BlogPost {...data} />  
</div>
```

Les propriétés de data seront ainsi utilisées

Comment boucler dans JSX ?

Si vous avez un ensemble d'éléments sur lesquels vous devez effectuer une boucle pour générer un partiel JSX, vous pouvez créer une boucle, puis ajouter JSX à un tableau :

```
const elements = [] //..some array

const items = []

for (const [index, value] of elements.entries()) {
  items.push(<Element key={index} />)
}
```

Comment boucler dans JSX

Maintenant, lors du rendu du JSX, vous pouvez intégrer le tableau items simplement en l'enveloppant entre accolades

```
const elements = ['one', 'two',  
  'three'];  
  
const items = []  
  
for (const [index, value] of  
  elements.entries()) {  
  items.push(<li  
    key={index}>{value}</li>  
  )  
  
  return (  
    <div>  
      {items}  
    </div>  
  )  
}
```

Comment boucler dans JSX ?

Vous pouvez faire la même chose directement dans le JSX, en utilisant `map` au lieu d'une boucle `for` :

```
const elements = ['one', 'two', 'three'];
return (
  <ul>
    {elements.map((value, index) => {
      return <li key={index}>{value}</li>
    })}
  </ul>
)
```

Composants

Un composant est une pièce isolée de l'interface. Par exemple, dans une page d'accueil de blog typique, vous pouvez trouver le composant Sidebar et le composant Blog Posts List. Ils sont à leur tour composés de composants eux-mêmes, vous pouvez donc avoir une liste de composants d'articles de blog, chacun pour chaque article de blog, et chacun avec ses propres propriétés particulières.

React rend les choses très simples : tout est un composant.

Composants

Même les balises HTML simples sont des composants autonomes et sont ajoutées par défaut.

Les 2 lignes suivantes sont équivalentes, elles font la même chose. Un avec JSX , un sans, en injectant `<h1>Hello World!</h1>` dans un élément avec l'id app.

Composants

```
import React from 'react'
import ReactDOM from 'react-dom'

ReactDOM.render(<h1>Hello World!</h1>, document.getElementById('app'))

ReactDOM.render(
  React.DOM.h1(null, 'Hello World!'),
  document.getElementById('app')
)
```

Composants

Ici, React.DOM nous a exposé un composant `h1`. Quelles autres balises HTML sont disponibles ? Tous! Vous pouvez inspecter ces offres du React.DOM en le saisissant dans la console du navigateur :

(la liste est plus longue)

Les composants intégrés sont intéressants, mais vous passerez outre rapidement. Ce en quoi React excelle, c'est de nous permettre de composer une interface utilisateur en élaborant des composants personnalisés.

Composants personnalisés

Il existe 2 façons de définir un composant dans React :

1. Un composant fonctionnel

```
const BlogPostExcerpt = () => {  
  return (  
    <div>  
      <h1>Title</h1>  
      <p>Description</p>  
    </div>  
  )  
}
```


Composants personnalisés

2. Un composant de classe

```
import React, { Component } from 'react'

class BlogPostExcerpt extends Component {
  render() {
    return (
      <div>
        <h1>Title</h1>
        <p>Description</p>
      </div>
    )
  }
}
```

Composants personnalisés

Auparavant, les composants de classe étaient le seul moyen de définir un composant qui avait son propre état et pouvaient accéder aux méthodes de cycle de vie afin que vous puissiez faire des choses lorsque le composant a été rendu, mis à jour ou supprimé pour la première fois.

React Hooks a changé cela, donc nos composants de fonction sont maintenant beaucoup plus adaptables et il y a de moins en moins de composants de classe à l'avenir, bien que ce soit toujours un moyen parfaitement valable de créer des composants.

Composants personnalisés

Il existe également une troisième syntaxe qui utilise la syntaxe ES5, sans les classes :

(rare)

```
import React from 'react'

React.createClass({
  render() {
    return (
      <div>
        <h1>Title</h1>
        <p>Description</p>
      </div>
    )
  }
})
```

État : définir l'état par défaut d'un composant

Dans le constructeur de composant, initialisez **this.state**.
Par exemple, le composant `BlogPostExcerpt` peut avoir un État **clicked**:

```
class BlogPostExcerpt extends Component {  
  constructor(props) {  
    super(props)  
    this.state = { clicked: false }  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>Title</h1>  
        <p>Description</p>  
      </div>  
    )  
  }  
}
```

Comment accéder à l'état ?

On peut accéder à l'état de `clicked` par **`this.state.clicked`** :

```
class BlogPostExcerpt extends Component {  
  constructor(props) {  
    super(props)  
    this.state = { clicked: false }  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>Title</h1>  
        <p>Description</p>  
        <p>Clicked: {this.state.clicked}</p>  
      </div>  
    )  
  }  
}
```

Changement de l'état ?

Un état de doit jamais être changé avec :

```
this.state.clicked = true
```

Au lieu de cela, vous devriez toujours utiliser **setState()** à la place, en lui passant un objet :

```
this.setState({ clicked: true })
```


Changement de l'état ?

L'objet peut contenir un sous-ensemble ou un sur-ensemble de l'état. Seules les propriétés que vous transmettez seront mutées, celles omises seront laissées dans leur état actuel.

Pourquoi toujours utiliser `setState()` ?

La raison est qu'en utilisant cette méthode, React sait que l'état a changé. Il lancera ensuite la série d'événements qui conduiront au re-rendu du composant, ainsi que toute mise à jour du DOM .

Flux de données unidirectionnel

Un état appartient toujours à un composant. Toutes les données affectées par cet état ne peuvent affecter que les composants inférieurs : ses enfants.

Changer l'état d'un composant n'affectera jamais son parent, ses frères ou tout autre composant de l'application : uniquement ses enfants.

C'est la raison pour laquelle l'état est souvent déplacé vers le haut dans l'arborescence des composants.

Déplacer l'état vers le haut dans l'arborescence

En raison de la règle de flux de données unidirectionnel, si deux composants doivent partager un état, l'état doit être déplacé vers un ancêtre commun.

Souvent, l'ancêtre le plus proche est le meilleur endroit pour gérer l'État, mais ce n'est pas une règle obligatoire.

L'état est transmis aux composants qui ont besoin de cette valeur via les props :

```
class Converter extends React.Component {  
  constructor(props) {  
    super(props)  
    this.state = { currency: '€' }  
  }  
  
  render() {  
    return (  
      <div>  
        <Display currency={this.state.currency} />  
        <CurrencySwitcher currency={this.state.currency} />  
      </div>  
    )  
  }  
}
```

Déplacer l'état vers le haut dans l'arborescence

L'état peut être muté par un composant enfant en transmettant une fonction de mutation en tant que props :

```
class Converter extends React.Component {  
  constructor(props) {  
    super(props)  
    this.state = { currency: '€' }  
  }  
  
  handleChangeCurrency = event => {  
    this.setState({ currency: this.state.currency === '€' ? '$' : '€' })  
  }  
  
  render() {  
    return (  
      <div>  
        <Display currency={this.state.currency} />  
        <CurrencySwitcher  
          currency={this.state.currency}  
          handleChangeCurrency={this.handleChangeCurrency}  
        />  
      </div>  
    )  
  }  
}
```

```
const CurrencySwitcher = props => {  
  return (  
    <button onClick={props.handleChangeCurrency}>  
      Current currency is {props.currency}. Change it!  
    </button>  
  )  
}  
  
const Display = props => {  
  return <p>Current currency is {props.currency}</p>  
}
```

Les props

Les **props** sont la façon dont les composants obtiennent leurs propriétés. À partir du composant supérieur, chaque composant enfant obtient ses props du parent.

Dans un composant de fonction, les props sont tout ce qu'il passe, et ils sont disponibles en ajoutant props comme argument de la fonction :

Les props

```
const BlogPostExcerpt = props => {  
  return (  
    <div>  
      <h1>{props.title}</h1>  
      <p>{props.description}</p>  
    </div>  
  )  
}
```

Les props

Dans un composant de classe, les props sont passés par défaut. Il n'est pas nécessaire d'ajouter quoi que ce soit de spécial, et ils sont accessibles en tant que **this.props** dans une instance de composant.

Les props

```
import React, { Component } from 'react'

class BlogPostExcerpt extends Component {
  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <p>{this.props.description}</p>
      </div>
    )
  }
}
```

Les props

La transmission de props aux composants enfants est un excellent moyen de transmettre des valeurs dans une application. Un composant contient des données (càd a un état) ou reçoit des données via ses props.

Ça se complique quand :

- vous devez accéder à l'état d'un composant à partir d'un enfant situé à plusieurs niveaux (tous les enfants précédents doivent agir comme un pass-through, même s'ils n'ont pas besoin de connaître l'état, ce qui complique les choses)
- vous devez accéder à l'état d'un composant à partir d'un composant totalement indépendant.

Valeur par défaut pour les props

Si une valeur n'est pas requise, nous devons spécifier une valeur par défaut si elle est manquante lors de l'initialisation du composant.

```
BlogPostExcerpt.propTypes = {  
  title: PropTypes.string,  
  description: PropTypes.string  
}
```

```
BlogPostExcerpt.defaultProps = {  
  title: '',  
  description: ''  
}
```

Certains outils comme ESLint ont la capacité d'imposer la définition des `defaultProps` pour un composant avec certains `propTypes` non explicitement requis.

Comment sont passés les props ?

Lors de l'initialisation d'un composant, transmettez les props d'une manière similaire aux attributs HTML :

```
const desc = 'A description'  
//...  
<BlogPostExcerpt title="A blog post" description={desc} />
```

Nous avons passé le titre en chaîne, et description comme variable.

Children

Un prop spécial est **children**. Il contient la valeur de tout ce qui est passé dans le body du composant :

```
<BlogPostExcerpt title="A blog post" description="{desc}">  
  Something  
</BlogPostExcerpt>
```

Dans ce cas, à l'intérieur BlogPostExcerpt nous pourrions accéder à "Something" en recherchant **this.props.children**.

Children

Alors que les props permettent à un composant de recevoir des propriétés de son parent, d'être "instruit" d'imprimer certaines données par exemple, l'état permet à un composant de prendre vie lui-même et d'être indépendant de l'environnement qui l'entoure.

Composants de présentation vs composants de conteneur

Dans React, les composants sont souvent divisés en 2 grands compartiments : les **composants de présentation** et les **composants de conteneur** .

Chacun de ceux-ci a ses caractéristiques uniques.

Les composants de présentation sont principalement concernés par la génération d'un balisage à présenter.

Ils ne gèrent aucun type d'état, à l'exception de l'état lié à la présentation

Les composants de conteneur sont principalement concernés par les opérations « backend ».

Composants de présentation vs composants de conteneur

Ils peuvent gérer l'état de divers sous-composants. Ils peuvent envelopper plusieurs composants de présentation. Ils pourraient s'interfacer avec Redux.

Pour simplifier la distinction, nous pouvons dire que les **composants de présentation sont concernés par l'apparence** , les **composants de conteneur sont concernés par le fonctionnement des choses** .

Composants de présentation vs composants de conteneur

Par exemple, il s'agit d'un composant de présentation. Il obtient des données de ses accessoires et se concentre uniquement sur l'affichage d'un élément :

```
const Users = props => (  
  <ul>  
    {props.users.map(user => (  
      <li>{user}</li>  
    ))}  
  </ul>  
)
```

Composants de présentation vs composants de conteneur

D'autre part, s'il s'agit d'un composant de conteneur. Il gère et stocke ses propres données et utilise le composant de présentation pour les afficher.

Axios , est une bibliothèque cliente HTTP, et aide à envoyer des requêtes HTTP et gérer les réponses

```
class UsersContainer extends React.Component {  
  constructor() {  
    this.state = {  
      users: []  
    }  
  }  
  
  componentDidMount() {  
    axios.get('/users').then(users =>  
      this.setState({ users: users }))  
  }  
  
  render() {  
    return <Users users={this.state.users} />  
  }  
}
```

State vs Props

Dans un composant React, les **props** sont des variables qui lui sont transmises par son composant parent. **Le State** par contre est toujours variable, mais directement initialisé et géré par le composant.

L'état peut être initialisé par props.

Par exemple, un composant parent peut inclure un composant enfant en appelant `<ChildComponent />`

Le parent peut passer une prop en utilisant cette syntaxe :

```
<ChildComponent color=green />
```

State vs Props

Dans le constructeur `ChildComponent`, nous pourrions accéder à la prop :

```
class ChildComponent extends React.Component {  
  constructor(props) {  
    super(props)  
    console.log(props.color)  
  }  
}
```

et toute autre méthode de cette classe peut référencer les props en utilisant `this.props`.

State vs Props

Les props peuvent être utilisés pour définir l'état interne en fonction d'une valeur de prop dans le constructeur, comme ceci :

```
class ChildComponent extends React.Component {  
  constructor(props) {  
    super(props)  
    this.state.colorName = props.color  
  }  
}
```


State vs Props

Bien sûr, un composant peut également initialiser l'état sans utiliser les props.

Dans ce cas, il n'y a rien d'utile à faire, mais imaginez faire quelque chose de différent en fonction de la valeur de la prop, il est probablement préférable de définir une valeur d'état.

Les props ne doivent jamais être modifiés dans un composant enfant, donc s'il se passe quelque chose qui modifie une variable, cette variable doit appartenir à l'état du composant.

State vs Props

Les props sont également utilisés pour permettre aux composants enfants d'accéder aux méthodes définies dans le composant parent.

C'est un bon moyen de centraliser la gestion de l'état dans le composant parent, et d'éviter que les enfants aient besoin d'avoir leur propre état.

La plupart de vos composants afficheront simplement des informations basées sur les props qu'ils ont reçus et resteront **sans état** .

Types de Props

Étant donné que JavaScript est un langage typé dynamiquement, nous n'avons pas vraiment de moyen d'appliquer le type d'une variable au moment de la compilation, et si nous passons des types invalides, ils échoueront à l'exécution ou donneront des résultats étranges si les types sont compatibles mais pas ce que nous attendons.

Flow et TypeScript aident beaucoup, mais React a un moyen d'aider directement avec les types de props, et même avant d'exécuter le code, nos outils (éditeurs, linters) peuvent détecter quand nous passons les mauvaises valeurs :

Types de Props

```
import PropTypes from 'prop-types'
import React from 'react'

class BlogPostExcerpt extends Component {
  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <p>{this.props.description}</p>
      </div>
    )
  }
}

BlogPostExcerpt.propTypes = {
  title: PropTypes.string,
  description: PropTypes.string
}

export default BlogPostExcerpt
```

Quels types pouvons-nous utiliser ?

Voici les types fondamentaux que nous pouvons accepter :

- `PropTypes.array`
- `PropTypes.bool`
- `PropTypes.func`
- `PropTypes.nombre`
- `PropTypes.object`
- `PropTypes.string`
- `PropTypes.symbole`

Quels types pouvons-nous utiliser ?

Nous pouvons accepter l'un des deux types avec `oneOfType` :

```
PropTypes.oneOfType([  
  PropTypes.string,  
  PropTypes.number  
]),
```

Les tableaux ont une syntaxe spéciale que nous pouvons utiliser pour accepter un tableau d'un type particulier :

```
PropTypes.arrayOf(PropTypes.string)
```

Quels types pouvons-nous utiliser ?

Nous pouvons aussi composer des propriétés d'objet en utilisant :

```
PropTypes.shape({  
  color: PropTypes.string,  
  fontSize: PropTypes.number  
})
```

Ajouter **isRequired** entraînera le renvoi d'une erreur si cette propriété est manquante :

```
PropTypes.arrayOf(PropTypes.string).isRequired,  
PropTypes.string.isRequired,
```

Fragments de React

Remarquez comment à chaque fois les valeurs de retour sont encapsulées dans une **div**

En effet, un composant ne peut renvoyer qu'un seul élément, et si vous en voulez plus d'un, vous devez l'envelopper avec une autre balise conteneur.

Ceci, cependant, amène une **div** inutile dans la sortie. Vous pouvez éviter cela en utilisant **React.Fragment**:

Fragments de React

```
import React, { Component } from 'react'

class BlogPostExcerpt extends Component {
  render() {
    return (
      <React.Fragment>
        <h1>{this.props.title}</h1>
        <p>{this.props.description}</p>
      </React.Fragment>
    )
  }
}
```

Fragments de React

Il a y une
syntaxe plus
simple `<></>`
mais qui
n'est gérée
que dans les
versions
récentes et
Babel 7+ :

```
import React, { Component } from 'react'

class BlogPostExcerpt extends Component {
  render() {
    return (
      <>
        <h1>{this.props.title}</h1>
        <p>{this.props.description}</p>
      </>
    )
  }
}
```

Events

React fournit un moyen simple de gérer les événements. On peut dire au revoir à **addEventListener**.

Dans la partie précédente sur le State, nous avons vu cet exemple :

```
const CurrencySwitcher = props => {  
  return (  
    <button onClick={props.handleChangeCurrency}>  
      Current currency is {props.currency}. Change it!  
    </button>  
  )  
}
```

Events

Si vous utilisez JavaScript depuis un certain temps, c'est exactement comme les anciens gestionnaires d'événements JavaScript, sauf que cette fois, vous définissez tout en JavaScript, pas dans votre HTML, et vous passez une fonction, pas une chaîne.

Les noms d'événements réels sont un peu différents car dans React, vous utilisez camelCase pour tout, donc **onclick** devient **onClick**, **onsubmit** devient **onSubmit**.

Pour référence, il s'agit du code HTML à l'ancienne avec des événements JavaScript mélangés :

```
<button onClick="handleChangeCurrency()">...</button>
```

Gestionnaire d'événements

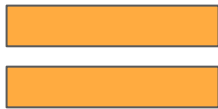
C'est une convention d'avoir des gestionnaires d'événements définis en tant que méthodes sur la classe Component :

```
class Converter extends React.Component {  
  handleChangeCurrency = event => {  
    this.setState({ currency: this.state.currency === '€' ? '$' : '€' })  
  }  
}
```

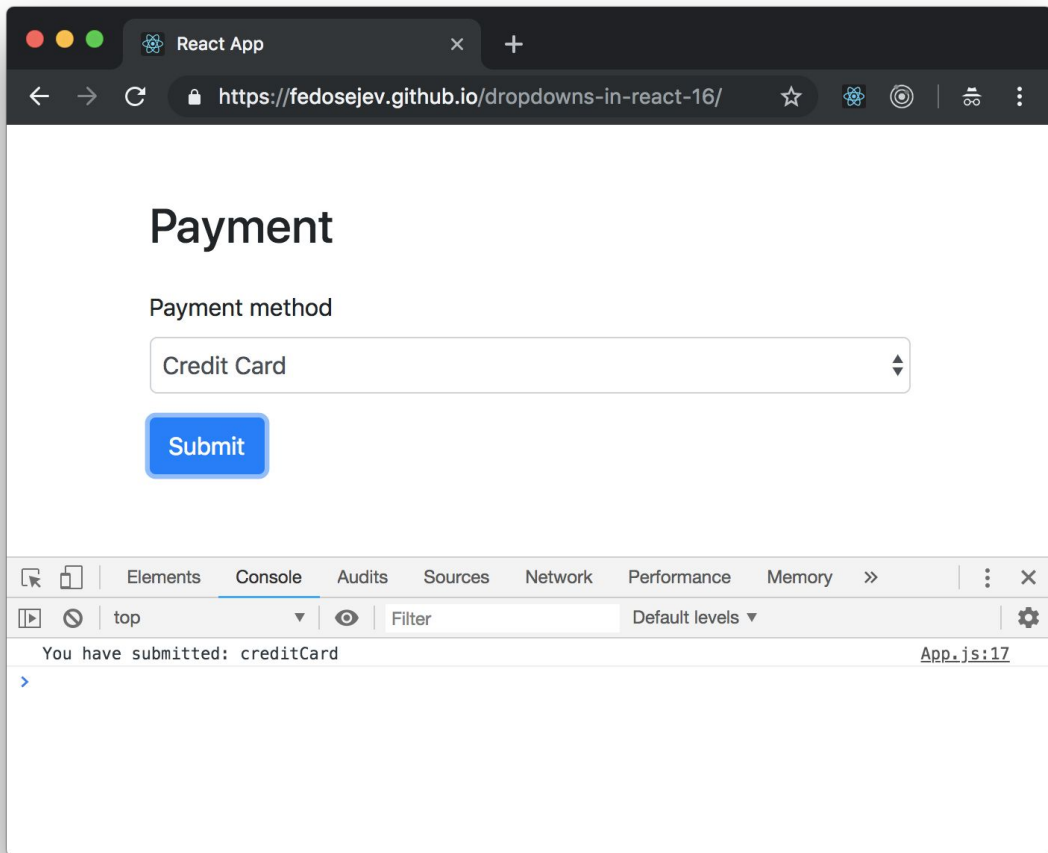
ES6 et **this** dans les arrow functions

this n'est défini que si vous définissez des méthodes en tant que fonctions fléchées, sinon on doit le lier à la classe :

```
class Converter extends  
React.Component {  
  handleClick = e => {  
    /* ... */  
  }  
  //...  
}
```



```
class Converter extends React.Component {  
  constructor(props) {  
    super(props)  
    this.handleClick = this.handleClick.bind(this)  
  }  
  handleClick(e) {}  
}
```



Formulaires avec React

Formulaires avec React

Il existe deux manières principales de gérer les formulaires dans React, qui diffèrent sur un niveau fondamental : la façon dont les données sont gérées.

- si les données sont manipulées par le DOM, on les appelle des **composants non contrôlés**
- si les données sont gérées par les composants, nous les appelons **composants contrôlés**

Formulaires avec React

Comme vous pouvez l'imaginer, les composants contrôlés sont ce que vous utiliserez la plupart du temps. L'état du composant est la seule source de vérité, plutôt que le DOM. Certains champs de formulaire sont intrinsèquement incontrôlés en raison de leur comportement, comme le champ `<input type="file">`.

Lorsque l'état d'un élément change dans un champ de formulaire géré par un composant, nous le suivons à l'aide de l'attribut **onChange**.

Formulaires avec React

```
class Form extends React.Component {
  constructor(props) {
    super(props)
    this.state = { username: '' }
  }

  handleChange(event) {}

  render() {
    return (
      <form>
        Username:
        <input
          type="text"
          value={this.state.username}
          onChange={this.handleChange}
        />
      </form>
    )
  }
}
```

Formulaires avec React

Afin de définir le nouvel état, nous devons lier **this** à la méthode **handleChange**, sinon **this** n'est pas accessible depuis cette méthode :

```
class Form extends React.Component {
  constructor(props) {
    super(props)
    this.state = { username: '' }
    this.handleChange =
this.handleChange.bind(this)
  }

  handleChange(event) {
    this.setState({ value: event.target.value })
  }

  render() {
    return (
      <form>
        <input
          type="text"
          value={this.state.username}
          onChange={this.handleChange}
        />
      </form>
    )
  }
}
```

Formulaires avec React

De même, nous utilisons l'attribut **onSubmit** dans le formulaire pour appeler la méthode **handleSubmit** lors de la soumission du formulaire :

```
class Form extends React.Component {
  constructor(props) {
    super(props)
    this.state = { username: '' }
    this.handleChange = this.handleChange.bind(this)
    this.handleSubmit = this.handleSubmit.bind(this)
  }

  handleChange(event) {
    this.setState({ value: event.target.value })
  }

  handleSubmit(event) {
    alert(this.state.username)
    event.preventDefault()
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type="text"
          value={this.state.username}
          onChange={this.handleChange}
        />
        <input type="submit" value="Submit" />
      </form>
    )
  }
}
```

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

function Form() {
  const [searchString, setSearchString] = useState();
  return (
    <div className="App">
      <input
        type="text"
        value={searchString}
        onChange={ (e) => setSearchString(e.target.value) }
      />
    </div>
  );
}

export default Form;
```

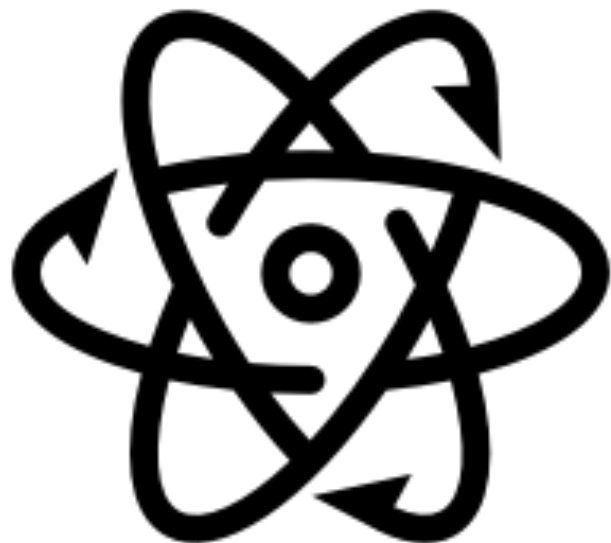
Formulaires avec React

Par exemple avec un **textarea** :

```
<textarea value={this.state.address} onChange={this.handleChange} />
```

et pour un **select** :

```
<select value="{this.state.age}" onChange="{this.handleChange}">  
  <option value="chien">Chien</option>  
  <option value="chat">Chat</option>  
</select>
```

Les Hooks de React

Les hooks permettent aux composants de fonction d'avoir un état et de répondre également aux événements du cycle de vie , et de rendre les composants de classe obsolètes.

Ils permettent également aux composants de fonction d'avoir un bon moyen de gérer les événements.

l'API `useState()`

En utilisant l'API `useState()`, vous pouvez créer une nouvelle variable d'état et avoir un moyen de la modifier. `useState()` accepte la valeur initiale de l'élément d'état et renvoie un tableau contenant la variable d'état et la fonction que vous appelez pour modifier l'état.

Puisqu'il renvoie un tableau, nous utilisons une déstructuration de tableau pour accéder à chaque élément individuel, comme ceci :

```
const [count, setCount] = useState(0)
```

Voici un exemple pratique :

```
import { useState } from 'react'

const Counter = () => {
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click
me</button>
    </div>
  )
}

ReactDOM.render(<Counter />, document.getElementById('app'))
```

l'API `useState()`

Vous pouvez en ajouter autant d'appels `useState()` que vous voulez, pour créer autant de variables d'état que vous le souhaitez.

Assurez-vous simplement de l'appeler au niveau supérieur d'un composant (pas dans un `if` ou dans tout autre bloc).

Accéder au Hook du cycle de vie

Une autre caractéristique très importante des Hooks est de permettre aux composants fonctionnels d'avoir accès aux hooks du cycle de vie.

Auparavant, avec les composants de classe, nous utilisions les événements **componentDidMount**, **componentWillUnmount** et **componentDidUpdate**.

Avec les hooks, maintenant, nous n'utilisons que l'API **useEffect()** qui accepte une fonction comme argument.

Accéder au Hook du cycle de vie

La fonction s'exécute lorsque le composant est rendu pour la première fois et à chaque nouveau rendu/mise à jour ultérieur.

React met d'abord à jour le DOM, puis appelle toute fonction transmise à **useEffect()**. Le tout sans bloquer le rendu de l'interface utilisateur même en bloquant le code, **contrairement à l'ancien componentDidMount et componentDidUpdate**, ce qui rend les applications plus rapides.

```

import { useEffect, useState } from 'react'

const CounterWithNameAndSideEffect = () => {
  const [count, setCount] = useState(0)
  const [name, setName] = useState('Flavio')
  useEffect(() => {
    console.log(`Hi ${name} you clicked ${count} times`)
  })
  return (
    <div>
      <p>
        Hi {name} you clicked {count} times
      </p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
      <button onClick={() => setName(name === 'Jackie' ? 'Loulou' : 'Jackie')}>
        Change name
      </button>
    </div>
  )
}

export default CounterWithNameAndSideEffect;

```


Accéder au Hook du cycle de vie

useEffect() peut être appelé plusieurs fois, ce qui est agréable pour séparer une logique non liée (quelque chose qui empoisonne les événements du cycle de vie des composants de classe).

Depuis le **useEffect()** les fonctions sont exécutées à chaque nouveau rendu/mise à jour ultérieur, nous pouvons dire à React d'ignorer une exécution, à des fins de performances, en ajoutant un deuxième paramètre qui est un tableau contenant une liste de variables d'état à surveiller. React ne réexécutera l'effet secondaire que si l'un des éléments de ce tableau change.

Accéder au Hook du cycle de vie

```
useEffect(  
  () => {  
    console.log(`Hi ${name} you clicked ${count} times`)  
  },  
  [name, count]  
)
```

Accéder au Hook du cycle de vie

De même, vous pouvez dire à React de n'exécuter l'effet secondaire qu'une seule fois (au moment du montage), en passant un tableau vide :

```
useEffect(() => {  
  console.log(`Component mounted`)  
  console.log(name, count)  
}, [])
```

Gérer les événements dans les composants de fonction

Nous pouvons utiliser l'API `useCallback()` intégrée

```
const Button = () => {  
  const handleClick = useCallback(() => {  
    //...do something  
  })  
  return <button onClick={handleClick} />  
}
```

Gérer les événements dans les composants de fonction

Tout paramètre utilisé à l'intérieur de la fonction doit être passé à travers un deuxième paramètre pour **useCallback()**, dans un tableau :

```
const Button = () => {  
  let name = '' //... add logic  
  const handleClick = useCallback(  
    () => {  
      //...do something  
    },  
    [name]  
  )  
  return <button onClick={handleClick} />  
}
```

```

import React, { useState } from 'react'

const Form = () => {
  const [age, setAge] = useState(99)
  const handleClick = () => setAge(age + 1)
  const someValue = "someValue"
  const doSomething = () => {
    return someValue
  }
  return (
    <div>
      <Age age={age} handleClick={handleClick}/>
      <Instructions doSomething={doSomething} />
    </div>
  )
}

const Age = ({ age, handleClick }) => {
  return (
    <div>
      <div style={{ border: '2px', background: "papayawhip", padding: "1rem" }}>
        Today I am {age} Years of Age
      </div>
      <pre> - click the button below 🖱️ </pre>
      <button onClick={handleClick}>Get older! </button>
    </div>
  )
}

const Instructions = React.memo((props) => {
  return (
    <div style={{ background: 'black', color: 'yellow', padding: "1rem" }}>
      <p>Follow the instructions above as closely as possible</p>
    </div>
  )
})

export default Form;

```

Dans l'exemple ci-dessus, le composant parent, `< Age / >`, est mis à jour (et restitué) chaque fois que **vous** cliquez sur le bouton Get Older.

Et aussi par conséquent, le composant enfant `<Instructions / >` est également restitué car la prop **doSomething** reçoit un nouveau rappel avec une nouvelle référence.

Il est à noter que même si le composant enfant Instructions utilise `React.memo` pour optimiser les performances, il est toujours re-rendu.

Alors maintenant, comment cela peut-il être corrigé pour empêcher `< Instructions / >` d'être re-rendu inutilement?

```

import React, { useState, useCallback } from 'react';

const App = () => {
  const [age, setAge] = useState(99)
  const handleClick = () => setAge(age + 1)
  const someValue = "someValue"
  const doSomething = useCallback(() => {
    return someValue
  }, [someValue])

  return (
    <div>
      <Age age={age} handleClick={handleClick} />
      <Instructions doSomething={doSomething} />
    </div>
  )
}

const Age = ({ age, handleClick }) => {
  return (
    <div>
      <div style={{ border: '2px', background: "papayawhip", padding: "1rem" }}>
        Today I am {age} Years of Age
      </div>
      <pre> - click the button below 🐼 </pre>
      <button onClick={handleClick}>Get older! </button>
    </div>
  )
}

const Instructions = React.memo((props) => {
  return (
    <div style={{ background: 'black', color: 'yellow', padding: "1rem" }}>
      <p>Follow the instructions above as closely as possible</p>
    </div>
  )
})

export default App;

```


Activer la communication entre les composants à l'aide de hooks personnalisés

La possibilité d'écrire vos propres hook est la fonctionnalité qui va considérablement modifier la façon dont vous écrivez des applications React à l'avenir.

En utilisant des hooks personnalisés, vous disposez d'un autre moyen de partager l'état et la logique entre les composants, ajoutant une amélioration significative aux modèles des props de rendu et des composants d'ordre supérieur. Ce qui est toujours pratique, mais maintenant avec les hooks personnalisés, ils ont moins de pertinence dans de nombreux cas d'utilisation.

Activer la communication entre les composants à l'aide de hooks personnalisés

Comment créer un hook personnalisé ?

Un hook est juste une fonction qui commence conventionnellement par `use`. Il peut accepter un nombre arbitraire d'arguments et retourner tout ce qu'il veut.

Exemples:

```
const useGetData() {  
  //...  
  return data  
}
```

```
const useGetUser(username) {  
  //...const user = fetch(...)  
  //...const userData = ...  
  return [user, userData]  
}
```

Activer la communication entre les composants à l'aide de hooks personnalisés

Dans vos propres composants, vous pouvez utiliser le crochet comme ceci :

```
const MyComponent = () => {  
  const data = useGetData()  
  const [user, userData] = useGetUser('jojo')  
  //...  
}
```

Quand exactement ajouter des hooks au lieu de fonctions régulières doit être déterminé en fonction des cas d'utilisation, et seule l'expérience le dira.

Fractionnement de code

Les applications JavaScript modernes peuvent être assez énormes en termes de taille de paquet. Vous ne voulez pas que vos utilisateurs aient à télécharger un package de 1 Mo de JavaScript (votre code et les bibliothèques que vous utilisez) juste pour charger la première page.

Mais c'est ce qui se passe par défaut lorsque vous expédiez une application Web moderne construite avec le regroupement Webpack.

Fractionnement de code

Cet ensemble contiendra du code qui pourrait ne jamais s'exécuter car l'utilisateur ne s'arrête que sur la page de connexion et ne voit jamais le reste de votre application.

Le fractionnement de code consiste à ne charger le JavaScript dont vous avez besoin qu'au moment où vous en avez besoin.

Cela améliore :

- les performances de votre application
- l'impact sur la mémoire, et donc l'utilisation de la batterie sur les appareils mobiles
- la taille des KiloBytes (ou MegaBytes) téléchargés

Fractionnement de code

React 16.6.0, publié en octobre 2018, a introduit un moyen de fractionner le code qui devrait remplacer chaque outil ou bibliothèque précédemment utilisé : **React.lazy** et **Suspense** .

React.lazy et **Suspense** constituent le moyen idéal pour charger “paresseusement” une dépendance et ne la charger qu'en cas de besoin.

Commençons avec **React.lazy**. Vous l'utilisez pour importer n'importe quel composant :

Fractionnement de code

```
import React from 'react'

const TodoList = React.lazy(() => import('./TodoList'))

export default () => {
  return (
    <div>
      <TodoList />
    </div>
  )
}
```

Fractionnement de code

Le composant `TodoList` sera ajouté dynamiquement à la sortie dès qu'il sera disponible. Webpack créera un bundle séparé pour cela et se chargera de le charger si nécessaire.

Suspense est un composant que vous pouvez utiliser pour envelopper n'importe quel composant chargé “paresseusement” :

Fractionnement de code

```
import React from 'react'

const TodoList = React.lazy(() => import('./TodoList'))

export default () => {
  return (
    <div>
      <React.Suspense>
        <TodoList />
      </React.Suspense>
    </div>
  )
}
```

Fractionnement de code

Il s'occupe de gérer la sortie pendant que le composant chargé paresseux est récupéré et rendu.

Utilisez son prop **fallback** pour sortir du JSX ou une sortie de composant :

```
...  
    <React.Suspense fallback={<p>Please wait</p>}>  
      <TodoList />  
    </React.Suspense>  
...
```

LOW 0

HIGH 5

MID 3



```
0. function binarySearch(list, item) {  
1.   let low = 0;  
2.   let high = list.length - 1;  
3.  
4.   while (low ≤ high) {  
5.     const mid = Math.round((low + high) / 2);  
6.     const guess = list[mid];  
7.  
8.     if (guess === item) {
```



drag to rewind

2 Exemples de Code

Construction d'un compteur

```
const Button = ({ increment }) => {  
  return <button>+{increment}</button>  
}
```

```
const App = () => {  
  let count = 0  
  
  return (  
    <div>  
      <Button increment={1} />  
      <Button increment={10} />  
      <Button increment={100} />  
      <Button increment={1000} />  
      <span>{count}</span>  
    </div>  
  )  
}
```

```
ReactDOM.render(<App />, document.getElementById('app'))
```

Ajoutons la fonctionnalité qui nous permet de changer le compte en cliquant sur les boutons, en ajoutant une prop **onClickFunction**:

```
const Button = ({ increment, onClickFunction }) => {
  const handleClick = () => {
    onClickFunction(increment)
  }
  return <button onClick={handleClick}>+{increment}</button>
}

const App = () => {
  let count = 0

  const incrementCount = increment => {
    //TODO
  }

  return (
    <div>
      <Button increment={1} onClickFunction={incrementCount} />
      <Button increment={10} onClickFunction={incrementCount} />
      <Button increment={100} onClickFunction={incrementCount} />
      <Button increment={1000} onClickFunction={incrementCount} />
      <span>{count}</span>
    </div>
  )
}

ReactDOM.render(<App />, document.getElementById('app'))
```

Ici, chaque élément Button a 2 props : **increment** et **onClickFunction**. Nous créons 4 boutons différents, avec 4 valeurs d'incrémentations : 1, 10, 100, 1000.

Lorsque vous cliquez sur le bouton du composant Button, la fonction **incrementCount** est appelée.

Cette fonction doit incrémenter le compteur local. Comment pouvons-nous faire ça? Nous pouvons utiliser des hooks :

```
const { useState } = React

const Button = ({ increment, onClickFunction }) => {
  const handleClick = () => {
    onClickFunction(increment)
  }
  return <button onClick={handleClick}>+{increment}</button>
}

const App = () => {
  const [count, setCount] = useState(0)

  const incrementCount = increment => {
    setCount(count + increment)
  }

  return (
    <div>
      <Button increment={1} onClickFunction={incrementCount} />
      <Button increment={10} onClickFunction={incrementCount} />
      <Button increment={100} onClickFunction={incrementCount} />
      <Button increment={1000} onClickFunction={incrementCount} />
      <span>{count}</span>
    </div>
  )
}

ReactDOM.render(<App />, document.getElementById('app'))
```


useState() initialise la variable `count` à 0 et nous fournit la méthode **setCount()** pour mettre à jour sa valeur.

Nous les utilisons les deux dans l'implémentation de la méthode **incrementCount()**, qui appelle **setCount()** pour la mise à jour de la valeur à la valeur existante de `count`, plus l'incrément passé par chaque composant `Button`.

Récupérer les infos des utilisateurs de GitHub via l'API

Exemple d'un formulaire qui accepte un nom d'utilisateur GitHub et une fois qu'il reçoit un événement **submit**, il demande à l'API GitHub les informations de l'utilisateur et les imprime.

Ce code crée un réutilisable composant **Card**. Lorsque vous entrez un nom dans le champ **input** géré par le composant **Form**, ce nom est *lié à son état* .

Lorsqu'on appuie sur **Ajouter une carte** , le formulaire de saisie est effacé en effaçant l'état **userName** du composant **Form**.

Récupérer les infos des utilisateurs de GitHub via l'API

L'exemple utilise, en plus de React, la bibliothèque Axios. C'est une bibliothèque utile et légère pour gérer les requêtes réseau. Installez-la localement en utilisant **npm install axios**.

Nous commençons par créer le composant **Card**, celui qui affichera notre image et les détails recueillis à partir de GitHub. Il obtient ses données via des props, en utilisant

- **props.avatar_url** : l'avatar de l'utilisateur
- **props.name** : l'identifiant
- **props.blog** : l'URL du site Web de l'utilisateur

Récupérer les infos des utilisateurs de GitHub via l'API

```
const Card = props => {  
  return (  
    <div style={{ margin: '1em' }}>  
      <img alt="avatar" style={{ width: '70px' }}  
src={props.avatar_url} />  
      <div>  
        <div style={{ fontWeight: 'bold' }}>{props.name}</div>  
        <div>{props.blog}</div>  
      </div>  
    </div>  
  )  
}
```

Récupérer les infos des utilisateurs de GitHub via l'API

Nous créons une liste de ces composants, qui sera transmise par un composant parent dans les props cards à **CardList**, qui itère dessus en utilisant `map()` et affiche une liste de cartes :

```
const CardList = props => (  
  <div>  
    {props.cards.map(card => (  
      <Card {...card} />  
    ))}  
  </div>  
)
```

Récupérer les infos des utilisateurs de GitHub via l'API

Le composant parent est App, qui stocke le tableau cards dans son propre état, géré à l'aide du Hook **useState()**:

```
const App = () => {  
  const [cards, setCards] = useState([])  
  
  return (  
    <div>  
      <CardList cards={cards} />  
    </div>  
  )  
}
```

Récupérer les infos des utilisateurs de GitHub via l'API

Nous devons maintenant avoir un moyen de demander à GitHub les détails d'un seul nom d'utilisateur.

Nous le ferons en utilisant un composant **Form**, où nous gérons notre propre état (**username**), et nous demandons à GitHub des informations sur un utilisateur utilisant ses API publiques, via Axios :

```

const Form = props => {
  const [username, setUsername] = useState('')

  handleSubmit = event => {
    event.preventDefault()

    axios.get(`https://api.github.com/users/${username}`).then(resp => {
      props.onSubmit(resp.data)
      setUsername('')
    })
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={username}
        onChange={event => setUsername(event.target.value)}
        placeholder="GitHub username"
        required
      />
      <button type="submit">Add card</button>
    </form>
  )
}

```


Récupérer les infos des utilisateurs de GitHub via l'API

Lorsque le formulaire est soumis, nous appelons l'événement **handleSubmit**, et après l'appel réseau, nous appelons **props.onSubmit** vers le parent (App) les données que nous avons obtenues de GitHub.

Nous l'ajoutons à App, en passant une méthode pour ajouter une nouvelle carte à la liste des cartes, **addNewCard**, et sa prop **onSubmit**:

```
const App = () => {  
  const [cards, setCards] = useState([])  
  
  addNewCard = cardInfo => {  
    setCards(cards.concat(cardInfo))  
  }  
  
  return (  
    <div>  
      <Form onSubmit={addNewCard} />  
      <CardList cards={cards} />  
    </div>  
  )  
}
```

```

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

const Card = props => {
  return (
    <div style={{ margin: '1em' }}>
      <img alt="avatar" style={{ width: '70px' }} src={props.avatar_url} />
      <div>
        <div style={{ fontWeight: 'bold' }}>{props.name}</div>
        <div>{props.blog}</div>
      </div>
    </div>
  )
}

const CardList = props => <div>{props.cards.map(card => <Card {...card} />)}</div>

const Form = props => {
  const [username, setUsername] = useState('')

  const handleSubmit = event => {
    event.preventDefault()

    axios
      .get(`https://api.github.com/users/${username}`)
      .then(res => {
        props.onSubmit({resp.data})
        setUsername('')
      })
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={username}
        onChange={event => setUsername(event.target.value)}
        placeholder="Github username"
        required
      />
      <button type="submit">Add card</button>
    </form>
  )
}

const Toto = () => {
  const [cards, setCards] = useState([])

  const addNewCard = cardInfo => {
    setCards(cards.concat(cardInfo))
  }

  return (
    <div>
      <Form onSubmit={addNewCard} />
      <CardList cards={cards} />
    </div>
  )
}

export default Toto;

```