



React JS



Speaker :



Renaud Dubuis

[View profile](#)

[LinkedIn](#)

Approche pédagogique.

Participants. (Tour de table)

Le tour de table initial favorise la création du groupe et permet la contextualisation des réponses aux questions individuelles.

Evaluation des pré-requis.

Le tour de table initial et les premiers exercices ont pour but l'évaluation effective des participants au regard des pré-requis.

Récapitulatif (matinal).

Chaque journée commence par un exercice collectif de restitution des concepts abordés la veille. L'objectif étant de renforcer (répétition) les connaissances acquises pour les utiliser comme socle lors de la journée à venir.

Concertation personnelle.

Le formateur passera assister les participants individuellement aussi souvent que possible.

Les participants sont invités à le solliciter pendant la journée de formation ou pour revenir sur un point particulier en fin de journée.

Travaux Pratiques.

L'acquisition des concepts abordés est découpée proportionnellement:

- **60%** de manipulation pratique.
- **40%** d'appports théoriques.

La mise en oeuvre des travaux pratiques se déroule en 3 phases, permettant la reproduction en autonomie des exercices.

1. Enoncé > 2. Démonstration > 3. Manipulation

Version numérique du support de formation.

Pour renforcer le confort de lecture et de manipulation (**copier/coller, liens cliquables, coloration du code**) le support est également distribué en version numérique. **au format PDF.** (sous license Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)).

Les références des sources utilisées pour la création de ce document pédagogique sont disponibles sur demande .

Présentation des participants.

Afin d'animer ensemble la formation et de constituer notre groupe il est utile de nous présenter en fournissant les informations lés pour le déroulement de ce stage.

- Identité.
- Rôle au sein du groupe.
- Positionnement sur les pré requis.
- Attente de cette formation.
- Précision.

Exemple

(identité) Bonjour je suis Renaud Dubuis.

(rôle) Je suis l'animateur de notre formation. (pré requis) En tant qu'architecte Font-End je suis amené à en maîtriser les différents outils. Je connais très bien HTML, CSS et JavaScript.

(attente) Je souhaite partager avec vous ces compétences par cette formation.

(précision) Etant dyslexique, j'ai parfois des soucis d'orthographe, je vous prie de m'en excuser.

Organisation pratique : repas, pause, temps de questions/réponses.

Lors d'une formation INTRA (dans les locaux de l'entreprise) les horaires sont adaptés en fonction de votre rythme habituels.

9:00 : Début de formation.

10:30: pause de la matinée (15 à 20 minutes)

Pause déjeuner: (choisir un horaire et une durée)

15:30: pause de l'après midi. (15 à 20 minutes)

17:00 : Temps des questions spécifiques.

17:30 : fin de journée.

Le récapitulatif matinal est noté et tenu à jour par le formateur dans un fichier nommé **RECAPITULATIF.md**. Ce fichier vous sera remis à la fin de la formation.

Les questions posées font soit:

1. L'objet d'une réponse immédiate.
2. L'objet d'une prise de note dans un fichier nommé **QUESTIONS.md** pour une réponse ultérieure avant les pauses ou en fin de journée, ou plus tard dans la formation.

Introduction du modèle de formation :

L'acquisition des concepts abordés est découpée proportionnellement :

- 60% de manipulation pratique.
- 40% d'apports théoriques.

La mise en oeuvre des travaux pratiques se déroule en 3 phases, permettant la reproduction en autonomie des exercices.

1. Enoncé > 2. Démonstration > 3.Manipulation.

1. Enoncé Verbal, avec l'appui du support de cours ou d'une documentation extérieure, telle qu'une documentation en ligne ou croquis.

2. Démonstration Le formateur illustre pratiquement le point expliqué.

Durant ces deux premières phases il est recommandé de :

Se concentrer sur le concept abordé. Prendre des notes.

NE PAS essayer reproduire les manipulations en même temps que le formateur.

3. Manipulation Vous reproduisez les manipulations en vous appuyant sur vos prises de notes. **N'hésitez pas à solliciter le formateur pour vous assister.**



Références à l'ouvrage et autres références

La formation est illustrée par la projection d'une version numérique (pdf) de votre support et l'utilisation d'autres ressources pertinentes (site internet, démonstration).

Autres références

- [reactjs officiel](#)
- [codementor.io](#)
- [Playground](#)
- [Playground JSX](#)
- [Playground No JSX](#)
- [node.js](#)
- [npmjs](#)
- [Webpack](#)

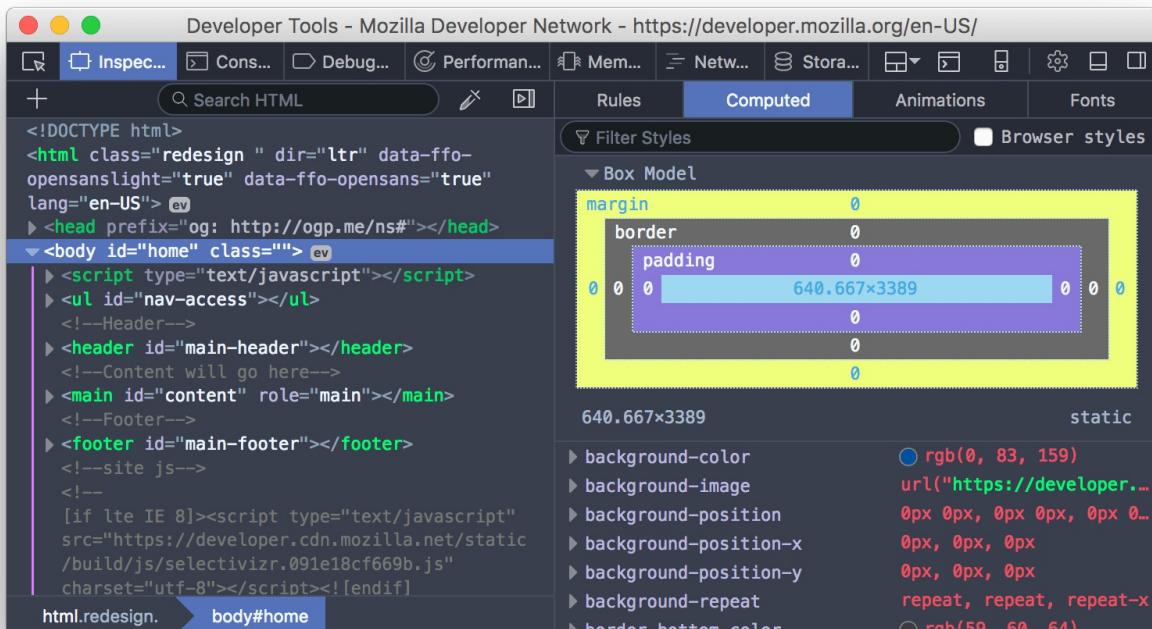
Configurer un environnement de développement moderne.

Pour programmer en JavaScript, un éditeur de texte suffit, il est intéressant de connaître l'offre en outillage autour de Node.

Un bon environnement telle que Chrome Examinez, modifiez, et déboguez du HTML, du CSS, et du JavaScript sur ordinateur, et sur mobile.

- Inspecteur
- Console web
- Débogueur JavaScript
- Moniteur réseau
- Performances

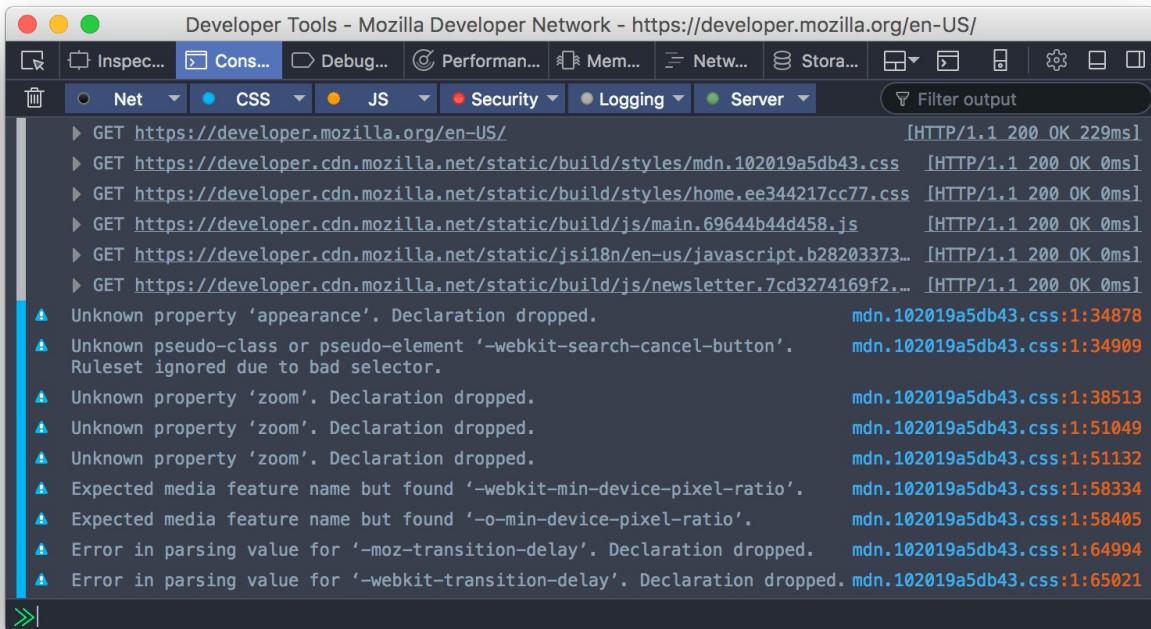
Inspecteur



Permet de voir et modifier une page en HTML et en CSS. Permet de

visualiser différents aspects de la page y compris les animations, l'agencement de la grille.

Console Web



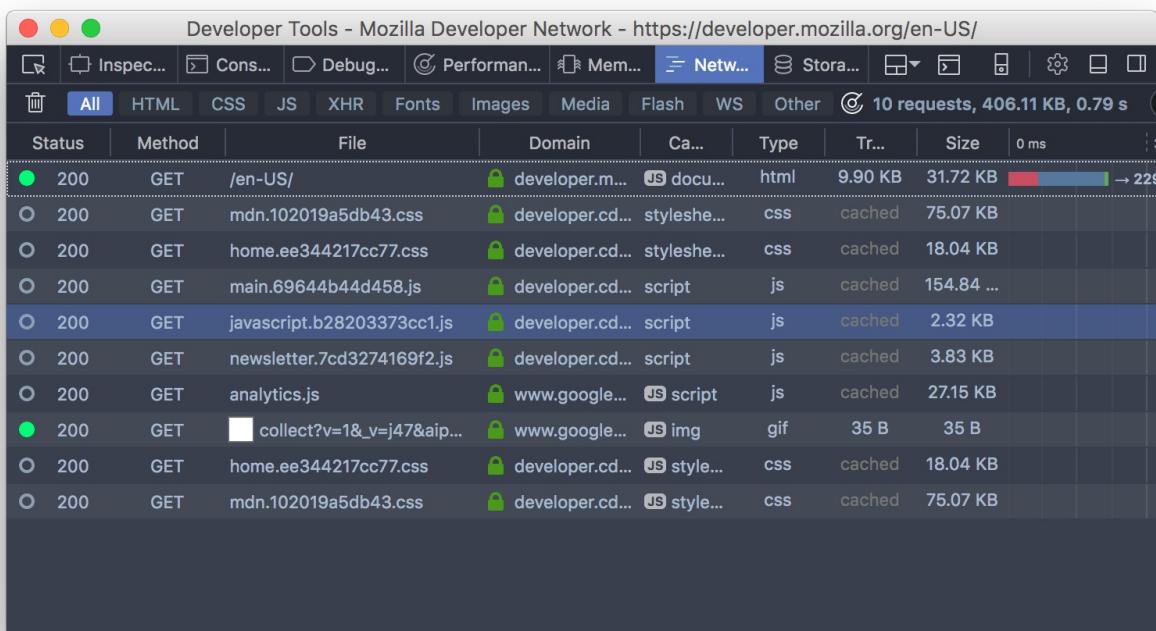
Affiche les messages émis par la page web. Permet également d'interagir avec la page via JavaScript.

Débogueur JavaScript

```
var dmp = new diff_match_patch();
function diff(text1, text2, opts) {
    text1 = text1 || "";
    text2 = text2 || "";
    opts = opts || {};
    opts.timeout = opts.timeout || 1;
    opts.cost = opts.cost || 2;
    opts.cleanup = opts.cleanup || "semantic";
    var ms_start = (new Date()).getTime();
    var d = dmp.diff_main(text1, text2);
    var ms_end = (new Date()).getTime();
    if (opts.cleanup === "semantic") {
        dmp.diff_cleanupSemantic(d);
    }
    if (opts.cleanup === "efficiency") {
        dmp.diff_cleanupEfficiency(d);
    }
}
```

Permet de parcourir, stopper, examiner et modifier le code JavaScript s'exécutant dans une page.

Réseau



The screenshot shows the Mozilla Developer Network Network tab with 10 requests listed:

Status	Method	File	Domain	Ca...	Type	Tr...	Size	Time
200	GET	/en-US/	developer.m...	JS docu...	html	9.90 KB	31.72 KB	→ 229 ms
200	GET	mdn.102019a5db43.css	developer.cd...	styleshe...	css	cached	75.07 KB	
200	GET	home.ee344217cc77.css	developer.cd...	styleshe...	css	cached	18.04 KB	
200	GET	main.69644b44d458.js	developer.cd...	script	js	cached	154.84 ...	
200	GET	javascript.b28203373cc1.js	developer.cd...	script	js	cached	2.32 KB	
200	GET	newsletter.7cd3274169f2.js	developer.cd...	script	js	cached	3.83 KB	
200	GET	analytics.js	www.google...	JS script	js	cached	27.15 KB	
200	GET	collect?v=1&_v=j47&aip...	www.google...	JS img	gif	35 B	35 B	
200	GET	home.ee344217cc77.css	developer.cd...	JS style...	css	cached	18.04 KB	
200	GET	mdn.102019a5db43.css	developer.cd...	JS style...	css	cached	75.07 KB	

Permet d'inspecter les requêtes réseau lors du chargement de la page.

Outils indispensables pour le développeur.

Pour mener à bien sa conception le développeur nécessite différents types d'outils

Il est possible de classer les outils selon trois catégories :

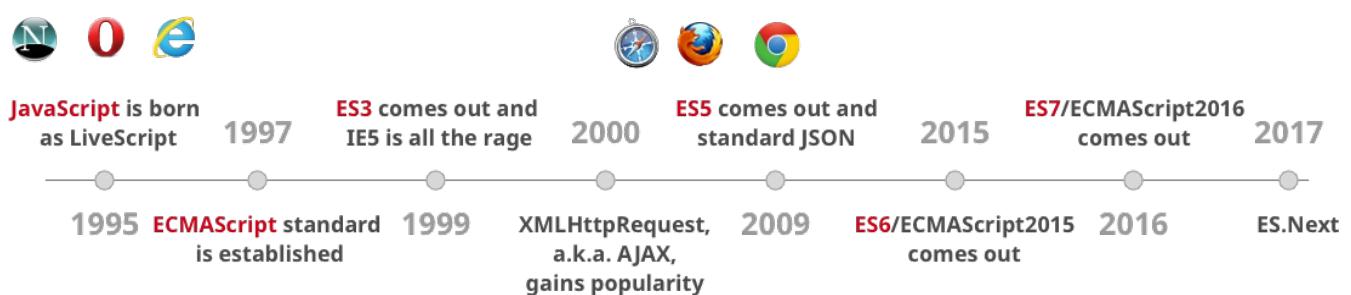
- **Logique** Dépendances tiers , tel que les différents frameworks.
- **Technique** ayant un impact sur la plate-forme de développement pour l'automatisation et le contrôle des tâches par exemple.
- **Productivité** toute autre solution non indispensable permettant d'accélérer le de développement, allant du simple plugin à une documentation efficace.

Logique : le développeur front-end maîtrise au moins un framework CSS dont **bootstrap** demeure le choix par défaut.

Technique : en dehors du choix de l'IDE de nombreux utilitaires ayant pour socle commun **NodeJS** fluidifient le développement.

Logique : Les points d'entrée vers la documentation sont une des clés de la productivité on citera [http.awesome.re](http://awesome.re) et <http://devdocs.io>.

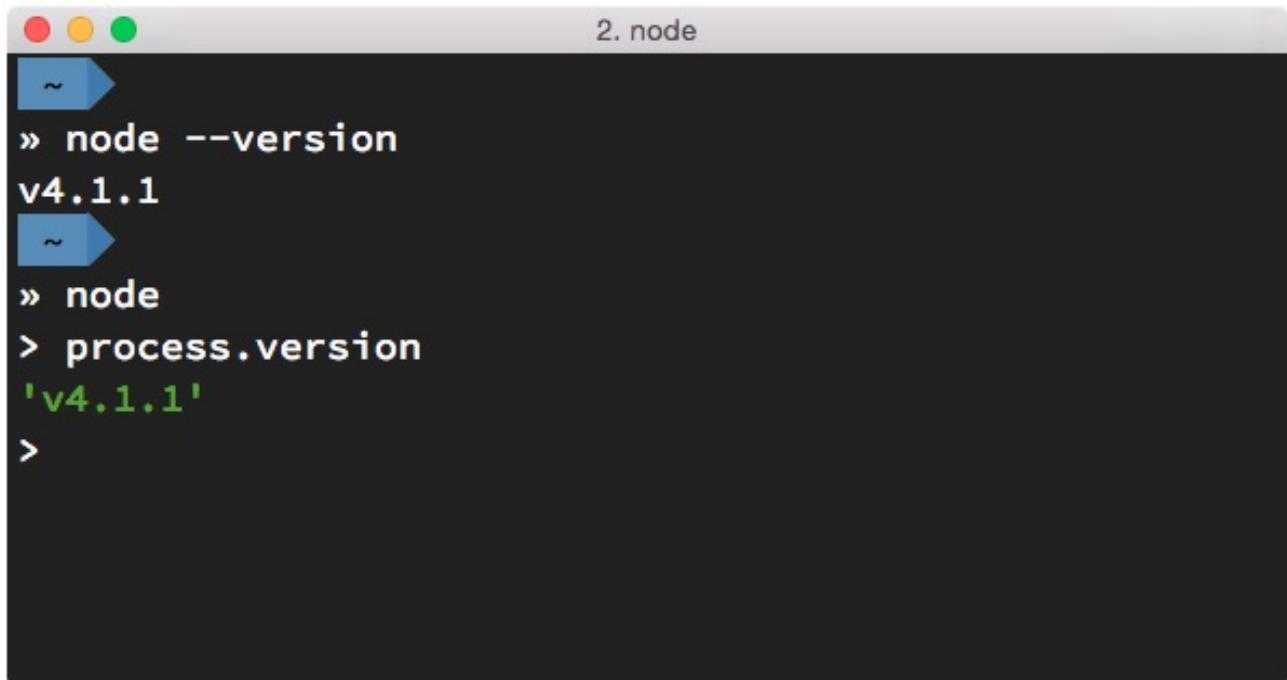
Prendre en considération l'évolution du langage



Node.js utilitaire de développement.

Installation de Node.JS :

Il est fortement recommandé d'utiliser un interpréteur de commandes (terminal ou shell). Les systèmes d'exploitation modernes en proposent un, y compris les versions récentes de Windows.



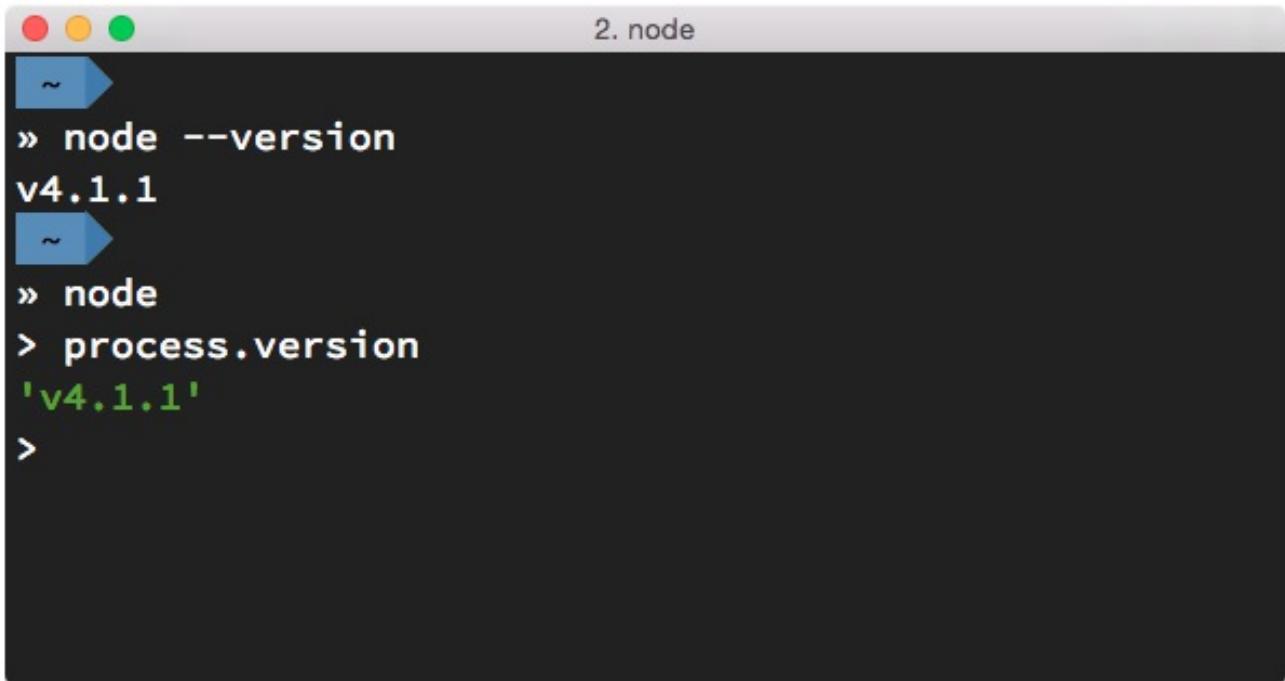
A screenshot of a macOS terminal window titled "2. node". The window has a dark background and light-colored text. It shows the following command-line session:

```
~ » node --version
v4.1.1
~ » node
> process.version
'v4.1.1'
>
```

Si vous n'utilisez pas encore de terminal, voici une liste de recommandations non exhaustive pour vous aider :

- *OS X* : iTerm2, Terminal.app.
- *Linux* : GNOME Shell, Terminator.
- *Windows* : PowerShell, Console.

Aisance avec l'invite de commande windows.



```
2. node
~
» node --version
v4.1.1
~
» node
> process.version
'v4.1.1'
>
```

Il est utile de pouvoir ouvrir rapidement une invite de commande pointant directement sur le répertoire voulu.

Manipulation 1

MAJ + CLIQUE DROIT > Ouvrir une invite de commande au dossier

Manipulation 2

A l'aide de l'explorateur windows, se placer dans le dossier **workshops** Saisir **cmd + ENTER** dans la barre d'adresse

Manipulation 3

Depuis **VS Code** choisir **menu>Afficher>Terminal Intégré**

Environnement de développement. IDE et plug-ins.

Outils de développement les autres logiciels :

Pour programmer avec angular en JavaScript, un éditeur de texte suffit, il est intéressant de connaître l'offre en outillage autour de Node.

Nous utiliserons VS Code.

cf. pratique

- Git cf. ressources attention à ajouter **git au PATH windows!**
- Node.js cf. ressources
- VS Code cf. ressources

Vérifier des installations :

cf. pratique

Vérifier l'installation de git, node et npm (installation par node), ouvrir une invite de commande et saisir les commandes suivantes :

À noter Pour travailler avec les outils de l'ecosystem Angular2 il faut une version de **Node 4+ et NPM 3+**

```
$> node --version  
x.x.x  
  
$> npm --version  
x.x.x  
  
$> git --version  
x.x.x
```

✓ Installation réussie !

Installer des modules tiers

Il existe deux modes d'installation avec npm :

global (machine)

```
$> npm install --global MODULE_NAME  
$> npm i -g MODULE_NAME
```

local (dossier courant)

```
$> npm install MODULE_NAME  
$> npm i MODULE_NAME
```

Installation des modules

```
$> npm i -g yarn eslint create-react-app react-native-cli json-server lite-server re
```

Initialisation

L'initialisation d'un projet Node passe par la création du fichier `package.json` et ce, quelle que soit sa taille.

`npm init`

L'utilisation de la commande `npm init` est une bonne habitude à prendre pour débuter tout projet Node.

La commande démarre une série de questions interactives.

Certaines réponses seront pré-remplies, par exemple si un dépôt Git ou un fichier README sont détectés.

À l'issue de la série de questions, le fichier `package.json` sera créé dans le répertoire courant.

Ensuite libre à vous de le compléter avec d'autres éléments optionnels de configuration.

Dépendances

Il existe plusieurs types de dépendances, chacune ayant sa propre utilité :

- *dependencies* : dépendances utiles à un fonctionnement en production ;
- *devDependencies* : dépendances utiles uniquement dans le cadre du développement, par exemple pour exécuter des tests ou s'assurer de la qualité du code ou encore empaqueter le projet ;
- *optionalDependencies* : dépendances dont l'installation ne sera pas nécessairement satisfaite, notamment pour des raisons de compatibilité. En général votre code prévoira que le chargement de ces modules via `require()` pourra échouer en prévoyant le traitement des exceptions avec un `try {} catch ()` ;
- *peerDependencies* : module dont l'installation vous est recommandée ; pratique couramment employée dans le cas de *plugins*. +
Par exemple, si votre projet A installe `gulp-webserver` en `devDependencies` et que `gulp-webserver` déclare `gulp` en `peerDependencies`, npm vous recommandera d'installer également `gulp` en tant que `devDependencies` de votre projet A .

npm en résumé

Commande	Signification
npm install modulename	Installe le module indiqué dans le répertoire node_modules de l'application. Ce module ne sera accessible que pour l'application dans laquelle il est installé. Pour utiliser ce module dans une autre application Node, il faudra l'installer, de la même façon, dans cette autre application, ou l'installer en global (voir l'option -g ci-dessous).
npm install modulename -g	Installe le module indiqué en global, il est alors accessible pour toutes les applications Node.
npm install modulename@version	Installe la version indiquée du module. Par exemple, npm install connect@2.7.3 pour installer la version 2.7.3 du module connect.
npm install	Installe dans le répertoire node_modules, les modules indiqués dans la clé dependencies du fichier package.json. Par exemple, le fichier package.json est de la forme suivante : { "dependencies": { "express": "3.2.6", "jade": "*", "stylus": "*" } } Ceci indique de charger la version 3.2.6 d'Express, avec les dernières versions de Jade et de Stylus, lorsque la commande npm install sera lancée.
npm start	Démarre l'application Node indiquée dans la clé start, elle-même incluse dans la clé scripts. Par exemple, le fichier package.json est de la forme suivante : { "scripts": { "start": "node app" } } Ceci indique d'exécuter la commande node app, lorsque la commande npm start sera lancée.

Commande	Signification
npm uninstall modulename	Supprime le module indiqué, s'il a été installé en local dans node_modules.
npm update modulename	Met à jour le module indiqué avec la dernière version.
npm update	Met à jour tous les modules déjà installés, avec la dernière version.
npm outdated	Liste les modules qui sont dans une version antérieure à la dernière version disponible.
npm ls	Affiche la liste des modules installés en local dans node_modules, avec leurs dépendances.
npm ls -g	Similaire à npm ls, mais affiche les modules installés en global.
npm ll	Similaire à npm ls, mais affiche plus de détails.
npm ll modulename	Affiche les détails sur le module indiqué.
npm search name	Recherche sur Internet les modules possédant le mot name dans leur nom ou description. Plusieurs champs name peuvent être indiqués, séparés par un espace. Par exemple, npm search html5 pour rechercher tous les modules ayant html5 dans leur nom ou description.
npm link modulename	Il peut parfois arriver qu'un module positionné en global soit malgré tout inaccessible par require(). Cette commande permet alors de lier le module global à un répertoire local (dans node_modules) de façon à le rendre accessible.
	sur Internet.

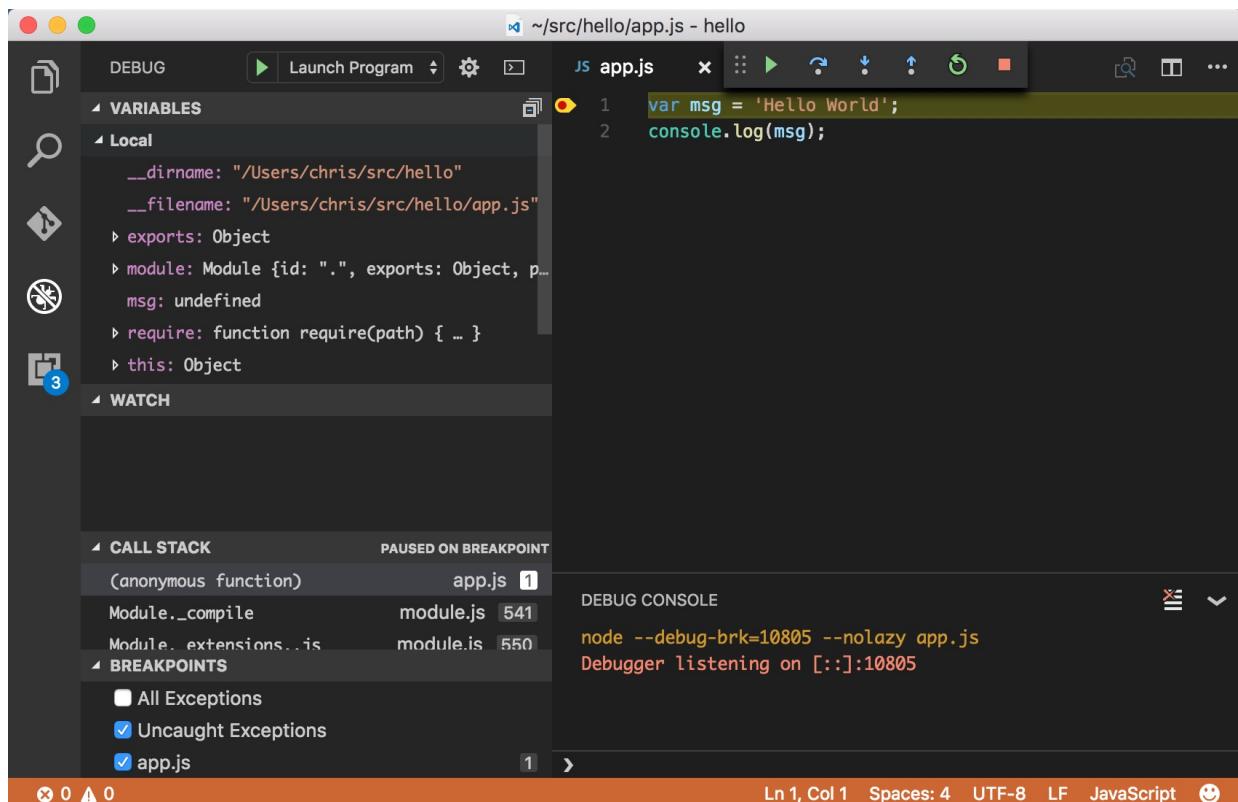
Structure de projet

Chaque développeur possèdera sa propre manière de ranger et d'organiser son code.

```
├── bin  
├── config  
├── data  
├── dist  
├── doc  
└── lib  
    └── models  
├── node_modules  
├── src  
    ├── assets  
    │   ├── images  
    │   ├── js  
    │   └── less  
    ├── routes  
    └── views  
├── tests  
    ├── fixtures  
    ├── functional  
    └── unit  
└── package.json  
└── README
```

Présentation de l'éditeur, les plug-ins indispensables.

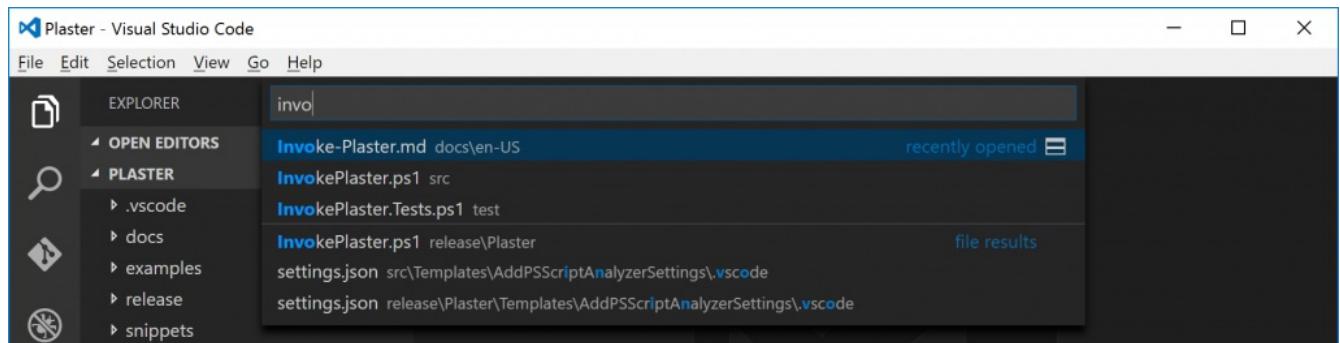
VS Code apporte toutes les fonctionnalités d'un éditeur moderne, sans nécessiter un investissement de formation.



Fonctionnalités principales:

- Palette de commande
- Gestion des fichiers et des projets
- "Snippets"
- Console
- Débuggeur
- Terminal intégré
- Intégration des plugins

La palette de commande

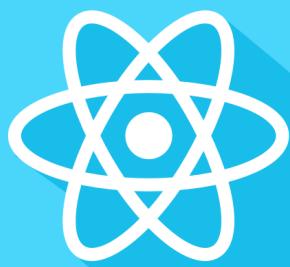


C'est le centre de contrôle de votre IDE Pour ouvrir la palette de commande, appuyez sur **Ctrl+Maj+P**, tapez le nom d'une commande, les suggestions les plus cohérentes s'affichent dans une liste, validez avec la touche ENTRÉE.

- Utilisation des commandes
- Saisie intuitive

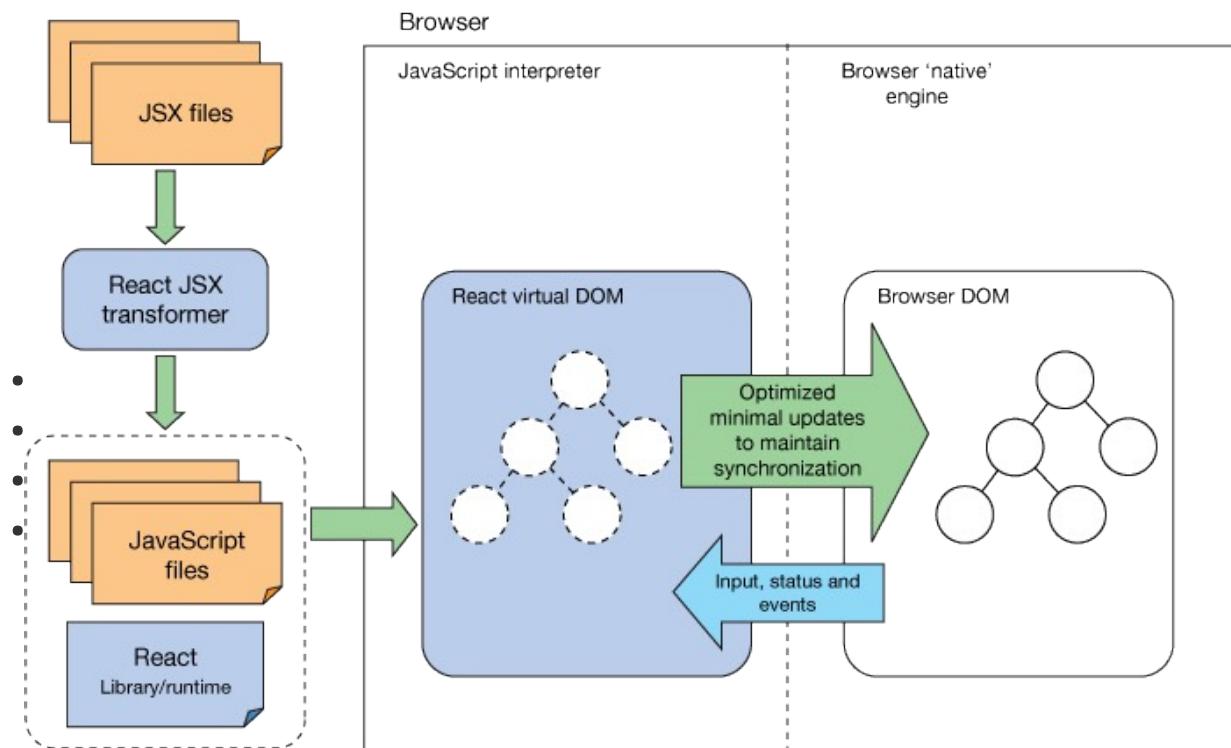


Rappels



React

Principe.



Principe

React (appelé aussi React.js) est un moteur de rendu JavaScript qui se démarque par une architecture voulue efficace et performante.

Initialement créé par Facebook pour développer *le fil d'actualité de son réseau social*. **React est publié en open source en mai 2013**, sous Licence Apache 2.0.

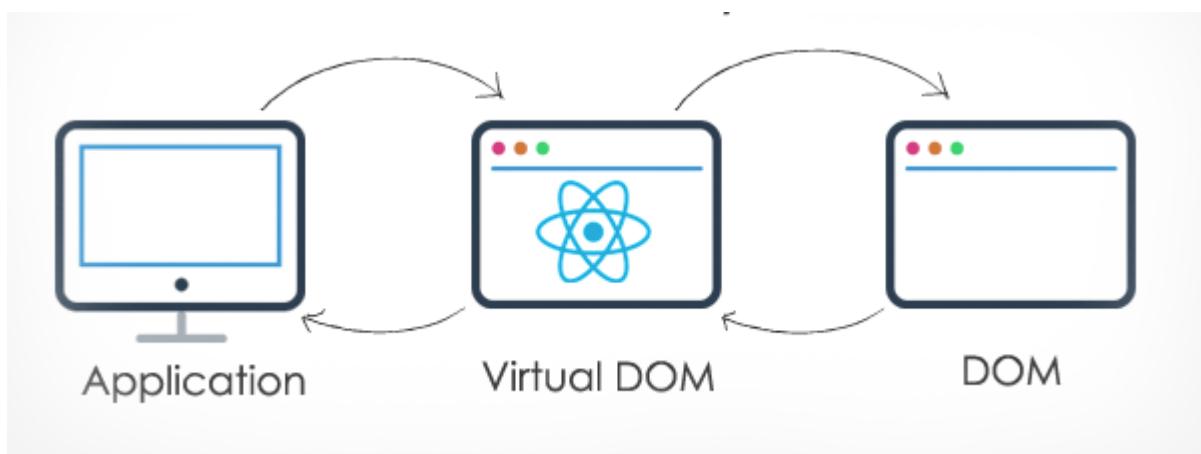
La logique *best of breed*

React cherche à offrir la meilleure réponse technologique à une problématique précise.

Rendre le DOM rapidement

React agit comme un **moteur de rendu** intermédiaire. Il s'agit d'**une librairie JavaScript** et non d'un framework.

En termes de performance, **React optimise les opérations** sur le DOM en utilisant un **DOM virtuel**.



Pour **exprimer la structure du Virtual DOM**, React utilise JSX. Un langage qui étend **JavaScript (subset)** avec une syntaxe déclarative permettant de définir le mode de rendu HTML du composant.

- ReactJS permet de fabriquer des composants (Web).
- Un composant ReactJS génère du code (HTML) à chaque changement d'état.
- ReactJS ne gère que la partie interface de l'application Web (Vue).
- ReactJS peut être utilisé avec une autres bibliothèque ou un framework (AngularJS).

En tant que **Librairie JavaScript ReactJS** satisfait aux problématiques de développement en utilisant l'écosystème industrialisé moderne

Rappels des composants des RIA

Les fondamentaux. HTML, CSS, JavaScript. Le DOM.

HTML signifie « **HyperText Markup Language** » qu'on peut traduire par « langage de balises pour l'hypertexte ». Il est utilisé afin de créer et de représenter le contenu d'une page web. D'autres technologies sont utilisées avec HTML pour décrire la présentation d'une page (**CSS**) et/ou ses fonctionnalités interactives (**JavaScript**).

Support des fonctionnalités HTML5

Référence HTML

Cascading Style Sheets (CSS) est un langage de feuille de style utilisé afin de décrire la présentation d'un document écrit en HTML ou en XML (on inclut ici les langages basés sur XML comme SVG ou XHTML). CSS décrit la façon dont les éléments doivent être affichés, à l'écran, sur du papier ou sur autre support.

CSS est l'un des langages principaux du Web et a été standardisé par le [W3C](#). Ce standard évolue sous forme de niveaux (levels), CSS1 est désormais considéré comme obsolète, CSS2.1 correspond à la recommandation et CSS3, qui est découpé en modules plus petits est en voie de standardisation.

Référence CSS

Le standard pour JavaScript est [ECMAScript](#). En 2012, tous les navigateurs modernes supportent complètement ECMAScript 5.1.

Tables de compatibilité JavaScript

Les anciens navigateurs supportent au minimum ECMAScript 3. **Une sixième version majeure du standard a été finalisée et publiée le 17 juin 2015.** Cette version s'intitule officiellement **ECMAScript 2015 mais est encore fréquemment appelée ECMAScript 6 ou ES6**. Étant donné que les standards ECMAScript sont édités sur un rythme annuel, cette documentation fait référence à la dernière version en cours de rédaction, actuellement c'est ECMAScript 2017.

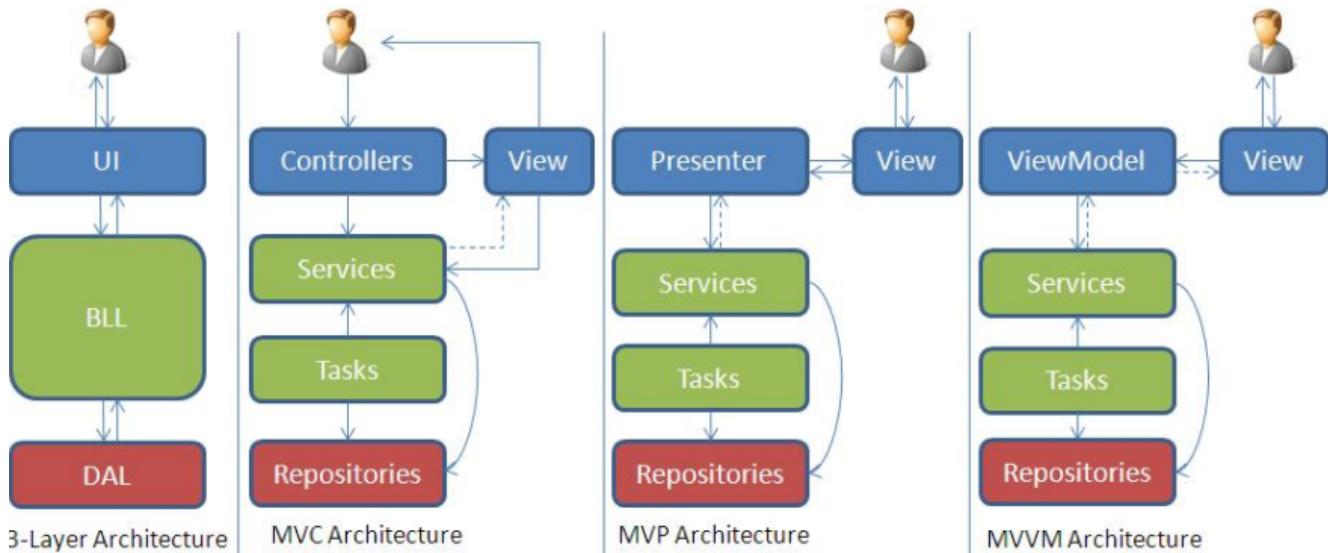
Référence JavaScript

JavaScript (qui est souvent abrégé en “JS”) est un langage de script léger, orienté objet, principalement connu comme le langage de script des pages web.

Il est aussi utilisé dans de nombreux environnements extérieurs aux navigateurs web tels que node.js ou Apache CouchDB.

C'est un langage à objets utilisant le concept de prototype, disposant d'un typage faible et dynamique qui permet de programmer suivant plusieurs paradigmes de programmation : **fonctionnelle, impérative et orientée objet**. Apprenez-en plus sur [JavaScript](#).

Design patterns applicatifs classiques. Limitations des applications JavaScript.



Architecture 3 tiers

L'architecture trois tiers, ou architecture à trois couches est l'application du modèle plus général qu'est le multi-tier. L'architecture logique du système est divisée en trois niveaux ou couches :

- **Couche présentation :** Elle correspond à la partie de l'application visible et interactive avec les utilisateurs. On parle d'interface homme machine.
- **Couche métier :** Partie fonctionnelle de l'application, qui implémente la « logique », et décrit les opérations que l'application opère sur les données en fonction des requêtes des utilisateurs.
- **Couche accès aux données:** Accès aux données propres au système, ou gérées par un autre système.

C'est une extension du modèle client-serveur.

Architecture MVC (Modèle-Vue-Contrôleur)

Modèle d'architecture logicielle destiné aux interfaces graphiques lancé en 1978 et très populaire pour les applications web. Le modèle est composé de trois types de modules ayant trois responsabilités différentes: les modèles, les vues et les contrôleurs.

- **Modèle :** Encapsulation des données ainsi que de la logique relative : validation, lecture et enregistrement.
- **Vue :** Partie visible d'une interface graphique.
- **Contrôleur :** Module de traitement des actions utilisateur, modifiant les données du modèle et appelant le rendu de la vue.

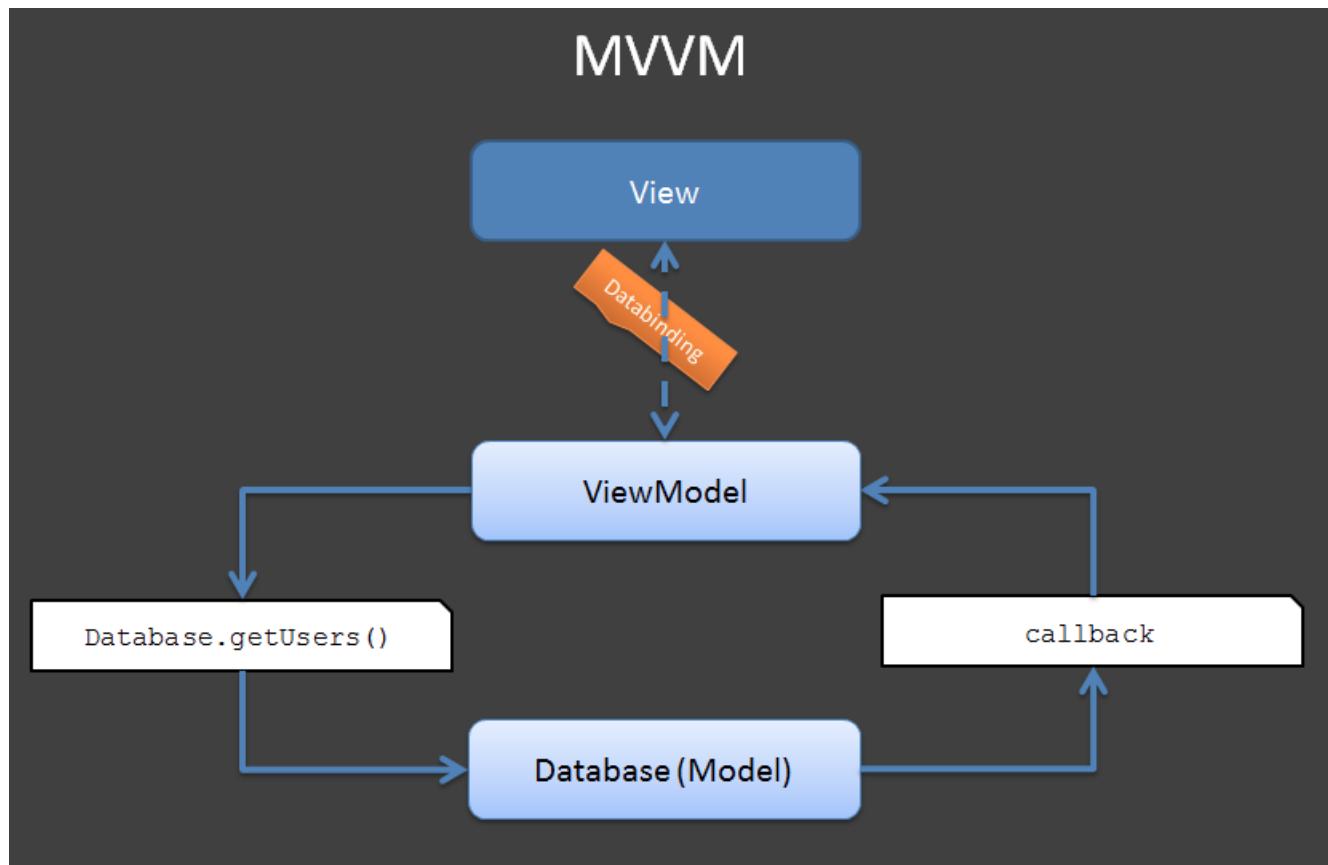
Architecture MVP (Model View Presenter)

Le modèle-vue-présentation est considéré comme un **dérivé de l'architecture modèle-vue-contrôleur**.

Il garde les mêmes principes que MVC en éliminant les interactions entre la vue et le modèle qui seront effectuées par le **biais de la présentation, organisant les données à afficher dans la vue**.

Architecture MVVM (Model View ViewModel)

Le modèle-vue-vue modèle est une architecture et une méthode de conception.

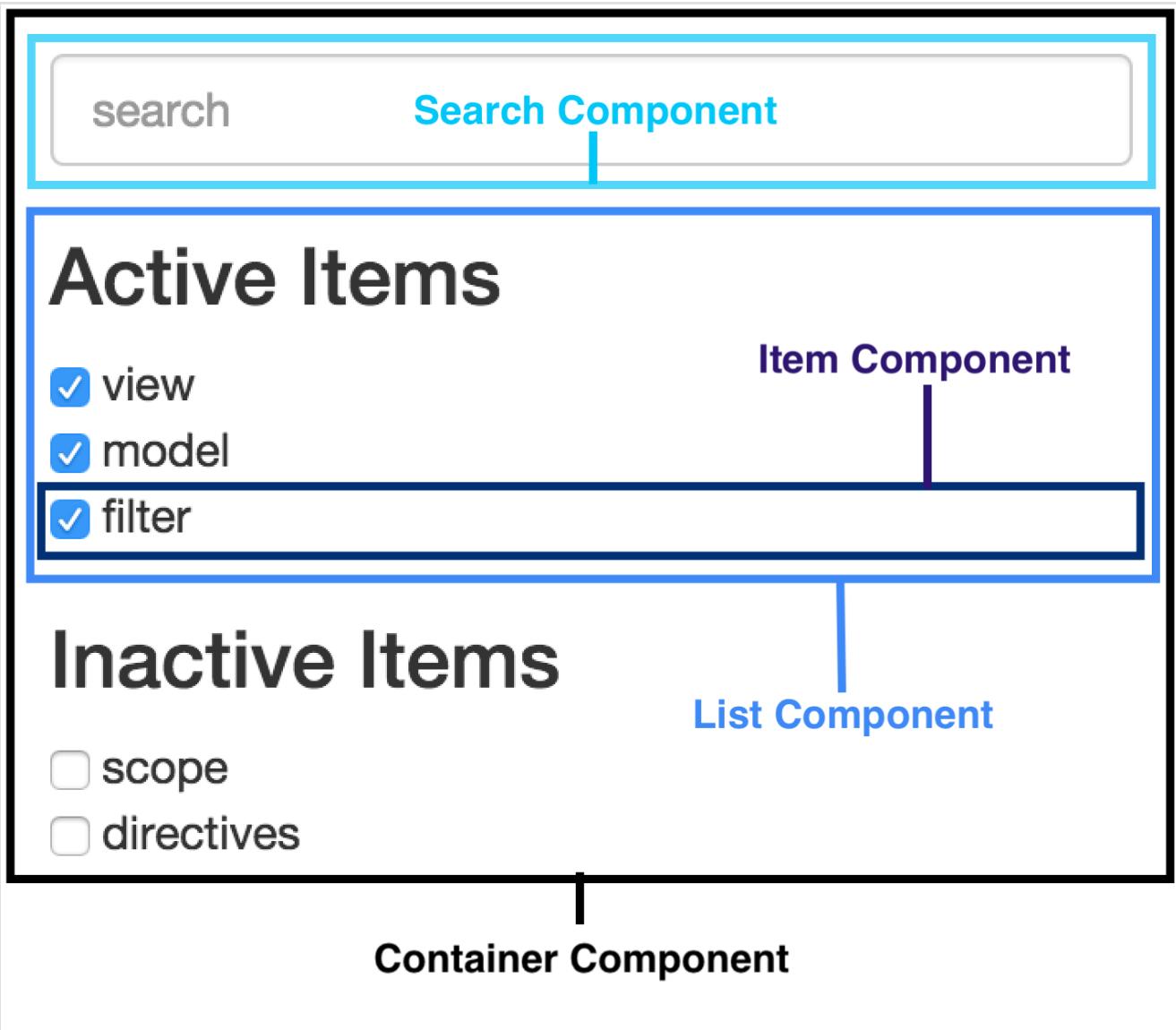


Cette méthode permet, tel le modèle MVC (modèle-vue-contrôleur), de séparer la vue de la logique et de l'accès aux données en **accentuant les principes de binding et d'événements**.

Component-based Architecture

La conception basée sur les composants vise la séparation des préoccupations par rapport aux fonctionnalités de l'application.

Il s'agit d'une approche basée sur la réutilisation pour définir, implémenter et composer des composants indépendants à couplage libre dans les systèmes.



On considère les composants comme faisant partie de la plateforme de départ pour l'orientation des services.

Les composants peuvent produire ou consommer des événements et peuvent être utilisés pour des architectures événementielles (EDA Event Driven Architecture).



Web Component

A propos des Web Components



Composants d’interface graphique réutilisables, qui ont été créés en utilisant des technologies Web (issues du standard).

Les [Composants Web](#) sont constitués de plusieurs technologies distinctes. Ils font partie du navigateur, et donc ils ne nécessitent pas de bibliothèques externes comme jQuery ou Dojo.

Un composant Web existant peut être utilisé sans l’écriture de code, en ajoutant simplement une déclaration d’importation à une page HTML. Les Composants Web utilisent les nouvelles capacités standards de navigateur, ou celles en cours de développement.

Les Composants Web sont constitués de ces quatre technologies (bien que chacun peut être utilisé séparément):

- [Custom Elements](#): pour créer et enregistrer de nouveaux éléments HTML et les faire reconnaître par le navigateur,
- [HTML Templates](#): squelettes pour des éléments HTML instanciables,
- [Shadow DOM](#): ce qui sera public ou privé dans vos éléments,
- [HTML Imports](#): pour packager ses composants (CSS, JavaScript, etc.)

Une démonstration minimalist dans chrome

Cet exemple utilise la syntaxe ES6

```
<body>
  <simple-increment data-step="5"></simple-increment>
  <script>
    class SimpleIncrement extends HTMLElement {
      constructor(args) {
        super();
      }

      increment(){
        return (this.count < this.max) ?(this.count += Number(this.step)):this.count
      }

      createdCallback() { // 1 Called after the element is created.
        [this.count,this.max,this.step] = [0,100,this.dataset.step];
        this.innerHTML = `<button><i>${this.count}</i> of ${this.max}</button>`;
      }

      attachedCallback() { // 2 Called when the element is attached to the document
        this.querySelector('button').addEventListener('click', (evt) =>
          evt.target.firstChild.textContent = this.increment());
      }

      attributeChangedCallback(){}
      detachedCallback(){}
    }
    document.registerElement('simple-increment', SimpleIncrement)
  </script>
</body>
```

En résumé, un Web Component est un fragment fonctionnel d'interface encapsulé dans :

Un modèle générique `HTMLElement`.

Un bloc logique `class`.

Un système de rendu `document.registerElement` ici le DOM.

Un cycle de vie exposé par le système de rendu.



Application JavaScript

Outils indispensables pour le développeur Front End.

Il est possible de classer les outils selon trois catégories :

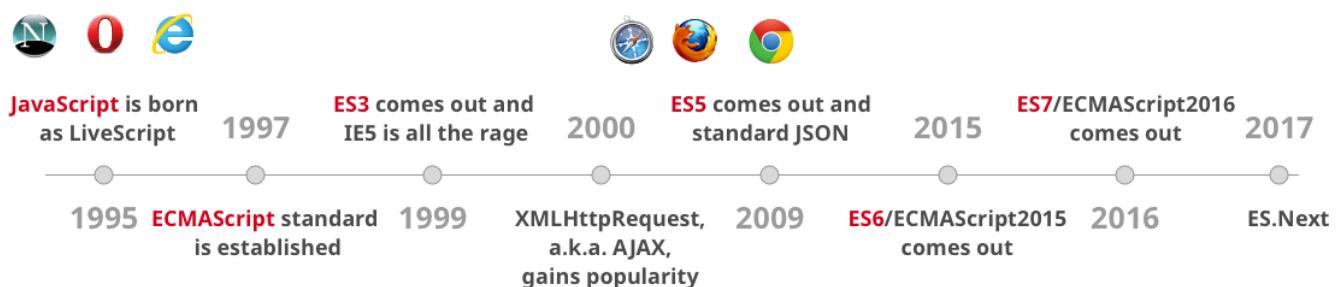
- **Logique** dépendances tiers ,telles que les différents frameworks.
- **Technique** ayant un impact sur la plate-forme de développement pour l'automatisation et le contrôle des tâches par exemple.
- **Productivité** toute autre solution non indispensable permettant d'accélérer le développement, allant du simple plugin à une documentation efficace.

Logique : le développeur front-end maîtrise au moins un [framework CSS](#), [bootstrap](#) demeure le choix par défaut.

Technique : en dehors du choix de l'IDE de nombreux utilitaires ayant pour [socle commun NodeJS](#) fluidifient le développement.

Logique : Les points d'entrée vers la documentation sont une des clés de la productivité on citera [http.awesom.re](http://awesom.re) et <http://devdocs.io>.

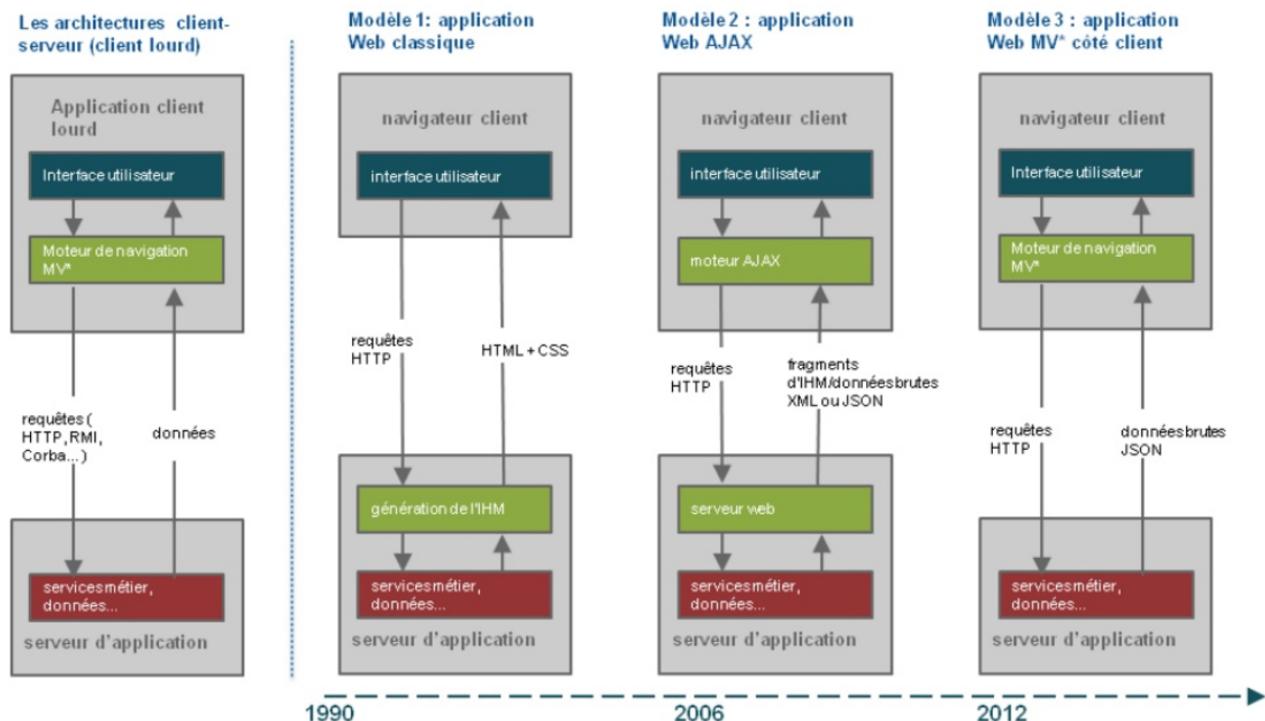
Prendre en considération l'évolution du langage



Par anticipation et les autres frameworks JavaScript se rapprochent de la future pattern de développement qu'apporteront les [Web Components](#)

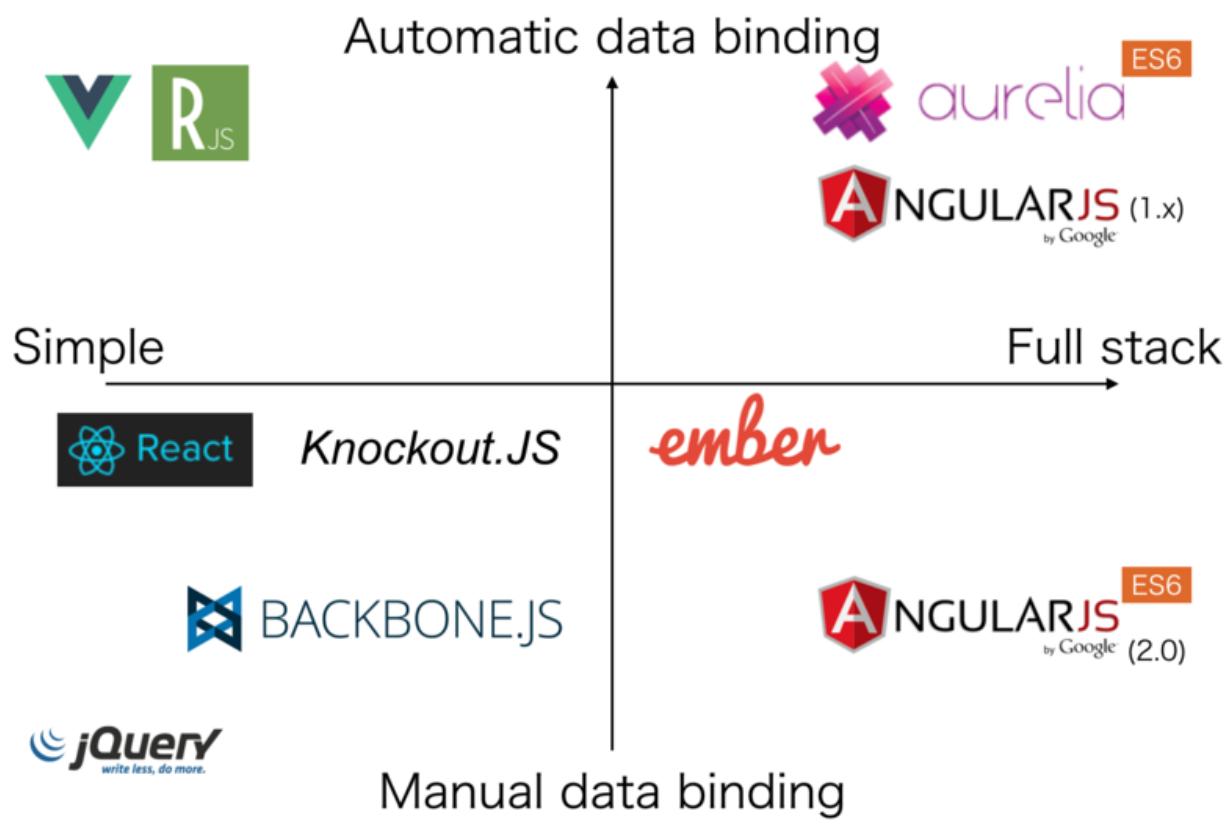
Ecosystème des frameworks JavaScript.

Evolution des développements Front End.

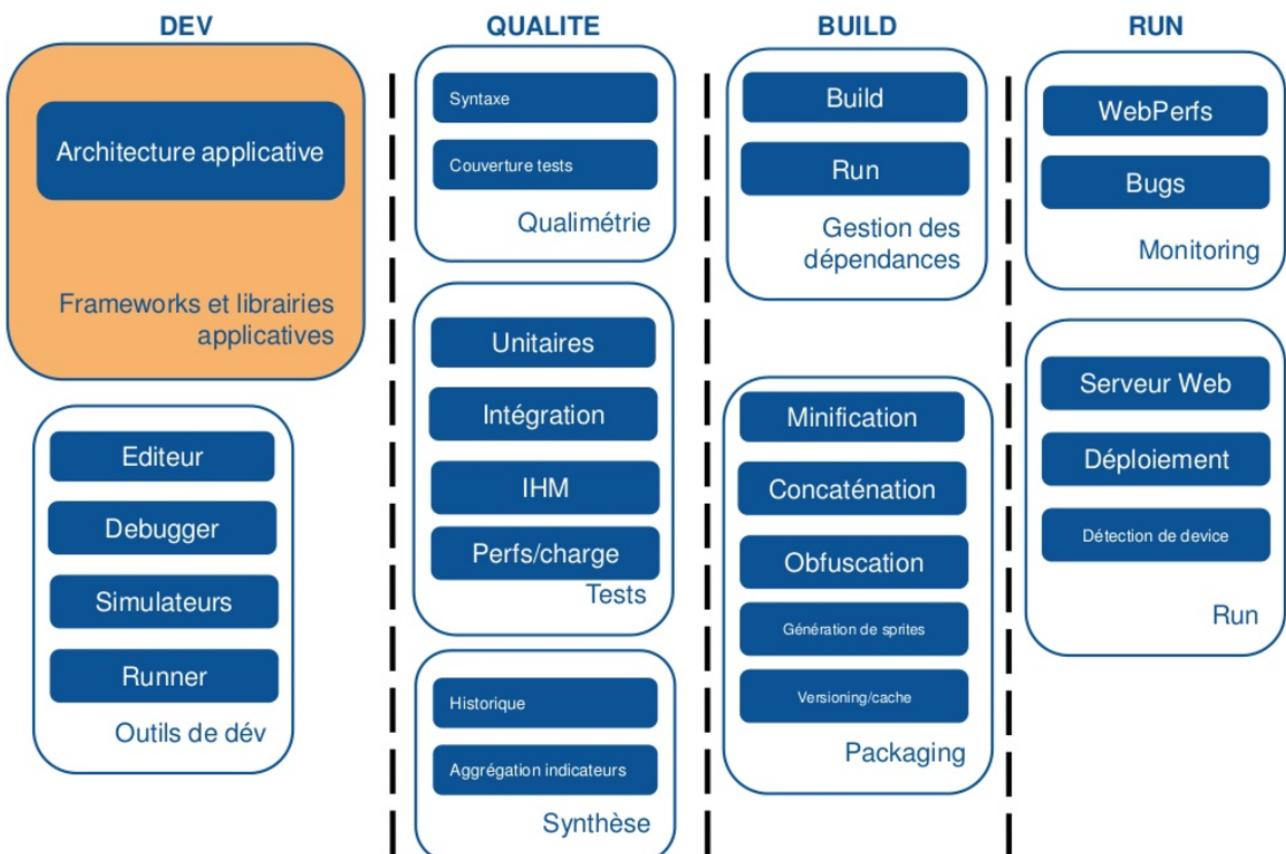


L'actuel "[marché](#)" des Frameworks UI/MVC JavaScript est très riche.

Chaque solution représente un choix d'implémentation particulier.

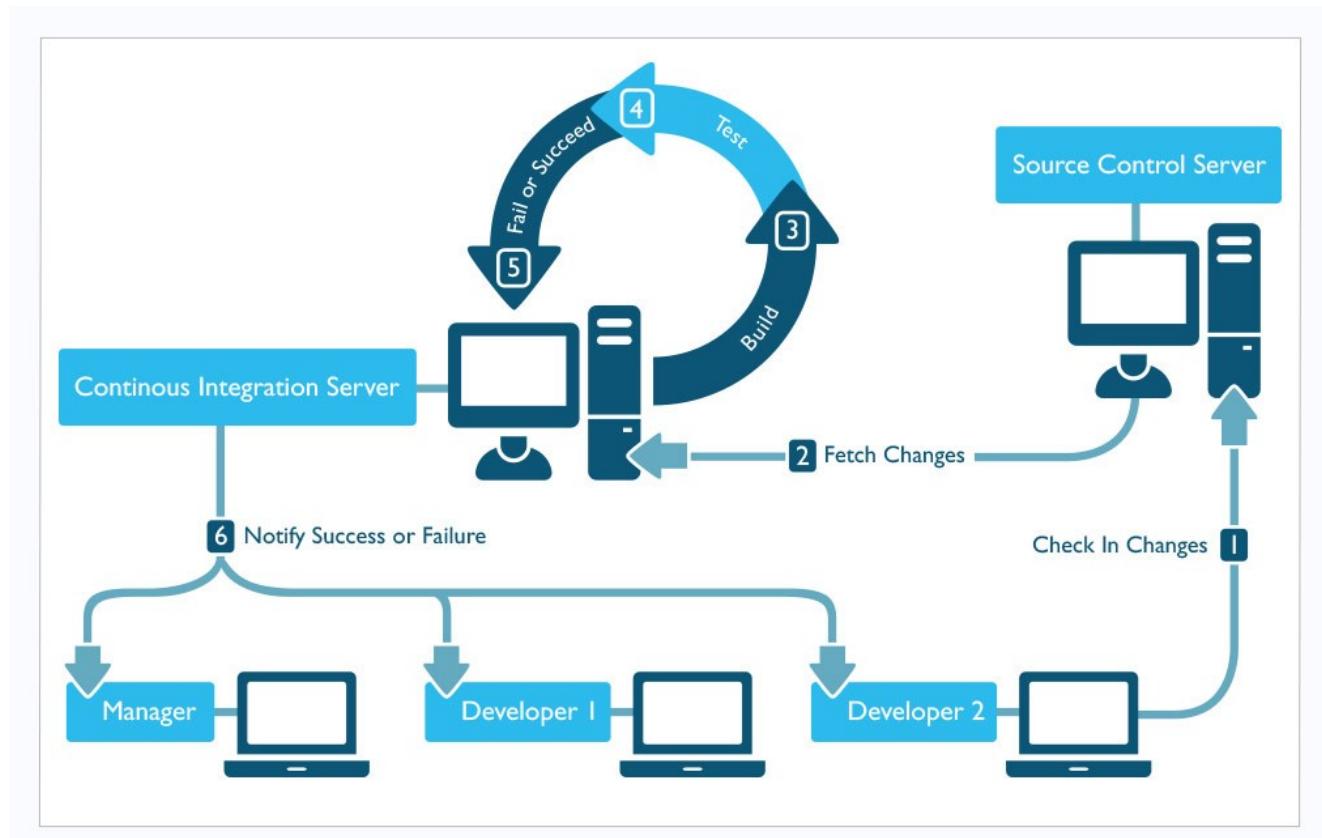


Un grand nombre d'utilitaires permettent d'automatiser les tâches découlant de l'évolution de ces pratiques de développement.



Industrialisation de la production.

Ces différents outils ont rendu possible l'industrialisation des développements par la séparation, la simplification et l'automatisation des différentes tâches.



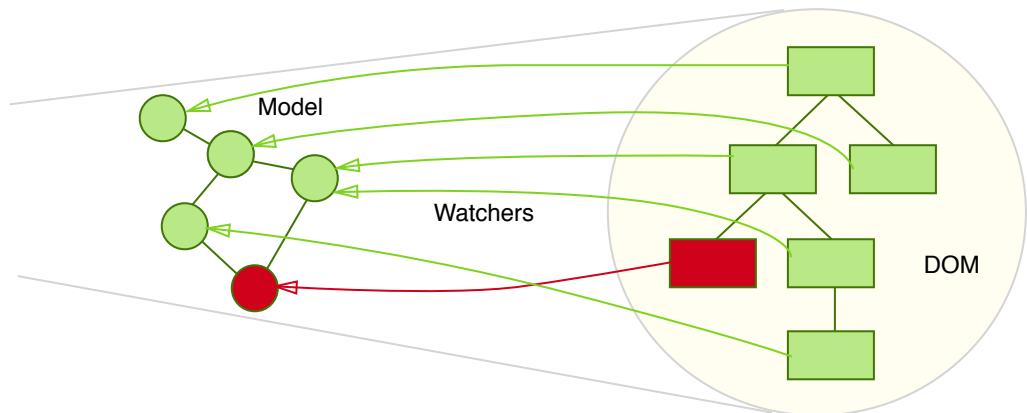


Composant : Etat, Cycle de Vie

Principes de Data-Binding : dirty-checking, observable, virtual-dom.

Dirty-checking (vérifications massives)

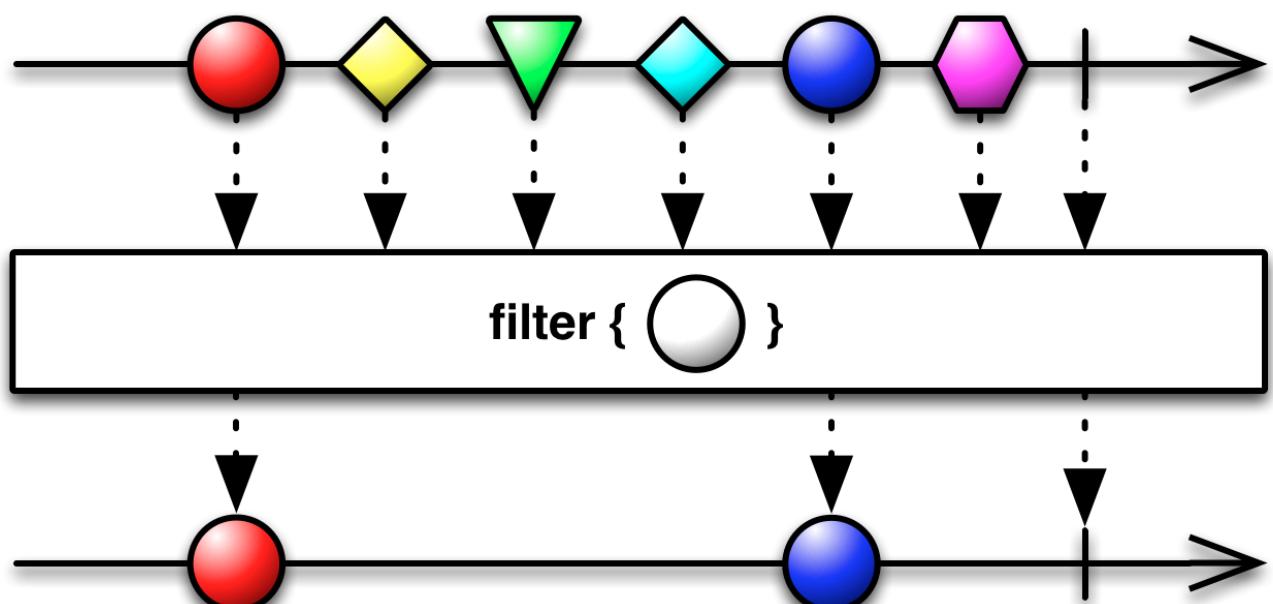
L'idée de base du dirty-checking est que dès qu'une modification des données peut avoir eu lieu, la bibliothèque va examiner l'intégralité du modèle de données pour déterminer s'il a changé, en utilisant un cycle de vie basé sur des sommes de contrôle ou sur les valeurs brutes.



Le coût de cette opération est proportionnel au nombre total d'objets observés.

Observable

Un *Observable* est un producteur de données (potentiellement asynchrone) qui peut être Observé. On le mettra sous observation avec la méthode `subscribe` et cette observation sera exécutée par un *Observer*.



Pour mieux saisir le concept il est possible d'utiliser les [graphiques interactifs](#)

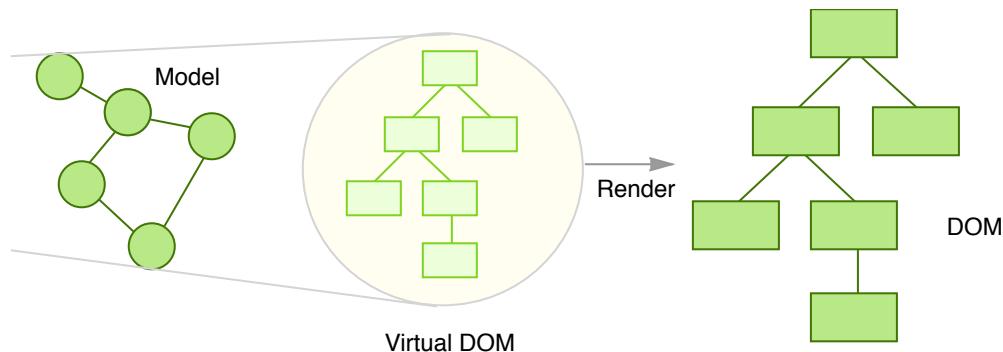
Virtual-Dom

Le DOM virtuel, est basé sur le concept de génération d'une arborescence d'objets Javascript en mémoire.

Un DOM virtuel est une représentation du DOM en Javascript, au premier affichage, le DOM virtuel est transformé en DOM réel.

Le moteur de rendu conserve au moins deux versions de l'arborescence, celle qui correspond au DOM réel et la nouvelle version qu'on veut afficher.

Puis il calcule la différence entre les deux versions et interagit lui-même avec le DOM pour trouver la façon minimale de le modifier pour obtenir la page souhaitée.



Au lieu de générer le DOM lui-même comme avec un langage de templating, le moteur se concentre sur le rendu des différences.

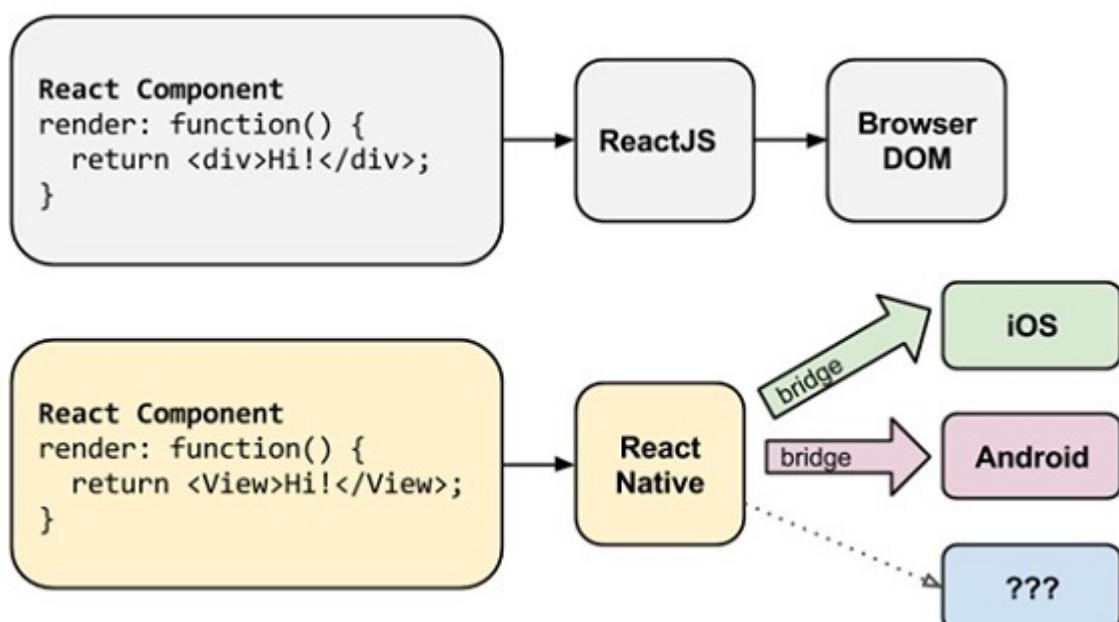
JSX, présentation. Mise en oeuvre “Transpilers”.

[JSX](#) est un subset JavaScript permettant d'écrire les templates avec une syntaxe xml incluse dans le code.

```
class Title extends Component {  
  render() {  
    return (  
      <div>  
        <img src={logo} className="App-logo" alt="logo" />  
        <h2>Welcome to React !</h2>  
      </div>  
    );  
  }  
}
```

Le fait d'utiliser une expression HTML du code permet éventuellement d'en faire l'abstraction à destination d'autres plate-formes cibles.

La syntaxe JSX induit le besoin de transformation du code.



Développer avec ReactJS

React repose sur la composition.

Composition

Un composant est une partie unitaire de l'application. Chacune de ces parties peuvent être elles-mêmes **composées d'autres composants**.

Une page web n'est alors rien d'autre qu'un **arbre de composants** dont les feuilles sont des balises html classiques.

Les composants sont isolés

Cette composition permet d'isoler des éléments de l'interface, leur comportement, leur état, leur forme et même, dans une certaine mesure, leur style.

L'isolation des composants permet d'intégrer des éléments React dans une application existante. **On peut grâce à la composition migrer une application progressivement.**

Les composants sont réutilisables, testables

Un composant isolé du reste d'un layout est plus facilement réutilisable, il suffit de le brancher aux bons endroits, avec la bonne source de données.

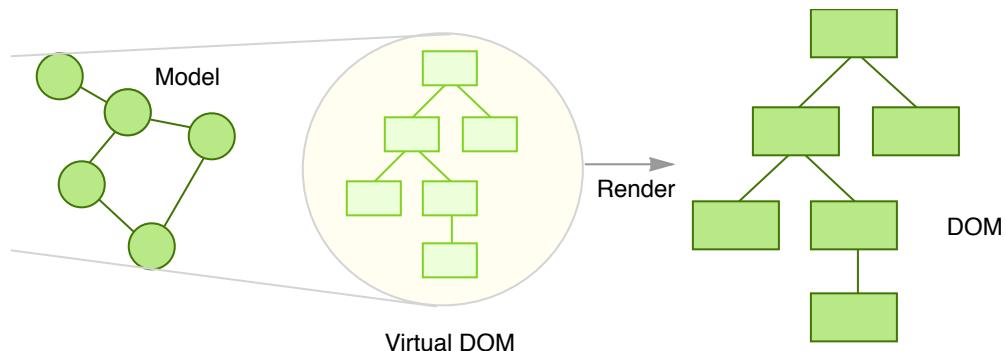
Il est plus facilement testable et vous permet d'avoir une bonne couverture de tests unitaires. **Il suffit de tester le comportement du composant en faisant abstraction de son contexte.**

Lorsque vous modifiez un composant, les effets de bord sont limités à ce composant uniquement. Votre application est plus maintenable.

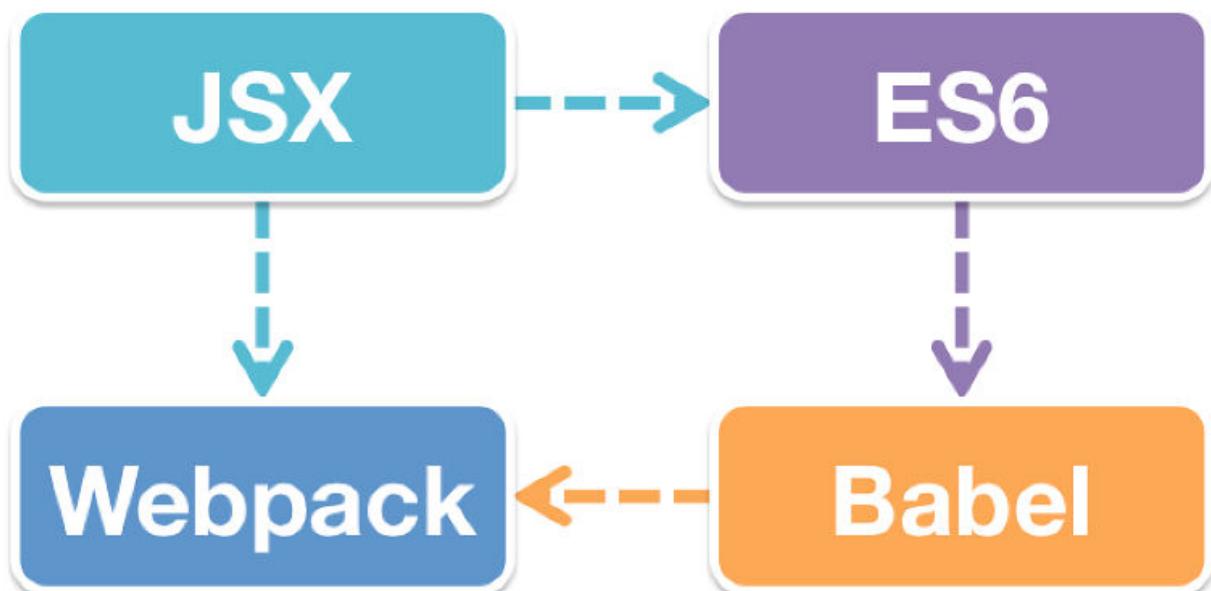
Approche : MVC et Virtual Dom, un choix de performance.

```
// Déclaration du composant
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`); // Déclaration
  }
}

ReactDOM.render(
  React.createElement(Hello, {toWhat: 'World'}, null),
  document.getElementById('root')
); // Rendu du VDOM
```



Utiliser JavaScript ou JSX.



Fondamentalement, JSX fournit juste du sucre syntaxique pour la fonction

```
React.createElement(component, props, ...children).
```

Le code JSX:

```
<MyButton color="blue" shadowSize={2}>  
  Click Me  
</MyButton>
```

Devient

```
React.createElement(  
  MyButton,  
  {color: 'blue', shadowSize: 2},  
  'Click Me'  
)
```

Comprendre JSX en détail.

Déclaration de composants.

La première partie d'un tag JSX détermine le type de l'élément React.

Les types en majuscules indiquent que la balise JSX fait référence à un composant React. Ces balises sont compilées en une référence directe à la variable nommée, donc si vous utilisez l'expression JSX `<Foo />`, Foo doit être dans la portée.

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
  // return React.createElement(CustomButton, {color: 'red'}, null);
  return <CustomButton color="red" />;
}
```

Vous pouvez également vous **référer à un composant React en utilisant la notation pointée à partir de JSX**.

Cela est pratique si vous avez un seul module qui exporte de nombreux composants React.

Par exemple, si `MyComponents.DatePicker` est un composant, vous pouvez l'utiliser directement à partir de JSX avec:

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

Lorsqu'un type d'élément commence par une lettre minuscule, il fait référence à un composant intégré comme `<div>` ou `` et produit une chaîne 'div' ou 'span' passée à `React.createElement`.

Il est recommandé de nommer des composants avec une majuscule. Si vous avez un composant qui commence par une lettre minuscule, affectez-le à une variable capitalisée avant de l'utiliser dans JSX.

```
import React from 'react';

// Wrong! This is a component and should have been capitalized:
function hello(props) { //Change to Hello
  // Correct! This use of <div> is legitimate because div is a valid HTML tag:
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Wrong! React thinks <hello /> is an HTML tag because it's not capitalized:
  return <hello toWhat="World" />; //Change to <Hello/>
}
```

Choix du type à l'exécution

Si vous voulez utiliser une expression générale pour **indiquer dynamiquement le type de l'élément**, il suffit de l'attribuer à une variable capitalisée en premier.

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Correct! JSX type can be a capitalized variable.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

Initialiser les propriétés de composants.

Il existe plusieurs façons de spécifier les propriétés en JSX.

- JavaScript Expressions
- String Literals
- Defaulted to “True”
- Spread Attributes (ES6)

Vous pouvez passer n'importe quelle expression JavaScript en tant que prop, en l'entourant de {} .

```
//JavaScript Expressions
<MyComponent foo={1 + 2 + 3 + 4} />

// String Literals
<MyComponent message="hello world" />
<MyComponent message={'hello world'} />

// Defaulted to "True"
<MyTextBox autocomplete /> //Sans valeur spécifiée la propriété vaut "true"
<MyTextBox autocomplete={true} />

// ES6 Spread Operator
<Greeting firstName="Ben" lastName="Hector" />

const props = {firstName: 'Ben', lastName: 'Hector'};
<Greeting {...props} />
```

Composants descendants (enfants).

Dans les expressions JSX qui contiennent à la fois une balise d'ouverture et une balise de fermeture, le contenu entre ces balises est transmis en tant que pilier spécial:
props.children .

Il existe plusieurs façons de transmettre des enfants:

- String Literals
- JavaScript Expressions
- JSX Children
- function

Booleans, Null, and Undefined seront ignorés.

```
// String Literals
<MyComponent>Hello world!</MyComponent>

// JavaScript Expressions
<MyComponent>{'foo'}</MyComponent>

// JSX Children
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>

// Valeurs ignorées
<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>
```

Template conditionnel basé sur les valeurs ignorées

Ce code JSX affichera uniquement un `<Header />` si `showHeader` est vrai:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

Attention: Certaines valeurs “*falsey*”, telles que le nombre 0, sont toujours rendues par React.

Rendu des éléments.

Les éléments sont les plus petits blocs d'applications React.

Un élément décrit ce que vous voulez voir à l'écran:

```
const element = <h1> Bonjour, monde </ h1>;
```

Contrairement aux éléments DOM du navigateur, les éléments React sont des objets simples et peu coûteux à créer. React DOM s'occupe de mettre à jour le DOM pour qu'il corresponde aux éléments React.

Note:

On pourrait confondre les éléments avec un concept plus largement connu de «composants». **Les «composants» sont constitués de d'éléments.**

Rendu d'un élément dans le DOM

Supposons qu'il y ait `<div>` quelque part dans votre fichier HTML:

```
<div id="root"> </div>
```

Ici l'élément est considéré **noeud DOM “racine”** car tout ce qu'il contient sera géré par React DOM.

Les applications construites uniquement avec React ont généralement un seul noeud DOM de la racine.

Pour rendre un élément React dans un nœud DOM racine, on utilise `ReactDOM.render()` :

```
const element = <h1> Bonjour, monde </ h1>;  
  
ReactDOM.render (  
  elementRef,  
  Document.getElementById ('root')  
)
```

[Essayez sur CodePen.](#)

Mise à jour de l'élément rendu

Les éléments React sont [immuables](#). Une fois que vous avez créé un élément, vous pouvez changer ses enfants ou ses attributs. Un élément est comme un cadre unique: **il représente l'interface utilisateur à un certain moment.**

Jusqu'à maintenant, la seule façon de mettre à jour l'interface utilisateur est de créer un nouvel élément et de le transmettre à `ReactDOM.render()`.

Considérez cet exemple d'horloge tic-tac:

```
function tick () {
  const element = (
    <Div>
      <H1> Bonjour, monde! </ H1>
      <H2> C'est {new Date().toLocaleTimeString ()}. </ H2>
    </ Div>
  );
  ReactDOM.render (
    element,
    Document.getElementById ('root')
  );
}

setInterval(tick, 1000);
```

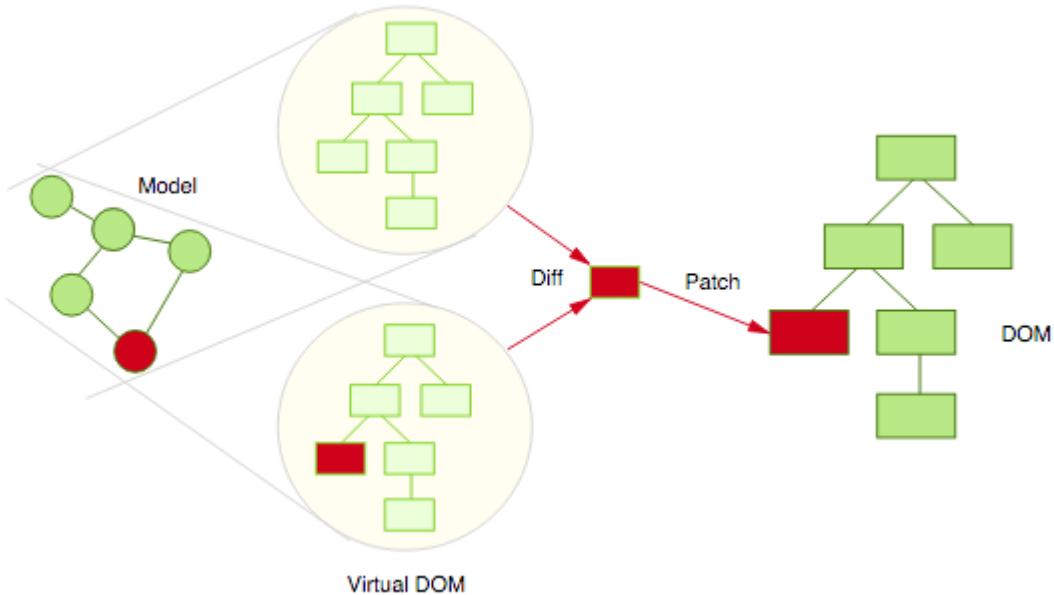
[Essayez sur CodePen.](#)

Il appelle `ReactDOM.render()` toutes les secondes à partir d'un callback [`setInterval\(\)`](#).

Note:

En pratique, la plupart des applications React n'appellent que `ReactDOM.render()` une fois.

React ne met à jour qui est nécessaire



React compare le DOM de l'élément et de ses enfants à l'ancien, et modifie seulement l'état désiré.

Vous pouvez vérifier en examinant le [dernier exemple](#) avec les outils du navigateur:

Hello, world!

It is 12:26:46 PM.

```
Console Sources Network Timeline
▼<div id="root">
  ▼<div data-reactroot>
    <h1>Hello, world!</h1>
    ▼<h2>
      <!-- react-text: 4 -->
      "It is "
      <!-- /react-text -->
      <!-- react-text: 5 -->
      "12:26:46 PM"
      <!-- /react-text -->
      <!-- react-text: 6 -->
      "."
      <!-- /react-text -->
    </h2>
  </div>
</div>
```

Même si nous créons un élément décrivant l'ensemble de l'arbre de l'interface utilisateur sur chaque *tick*, seul le texte est modifié par React DOM.

Méthodes principales de l'API.

L'API React est simple et concise.

`React` est le point d'entrée de la bibliothèque React. Si vous utilisez React comme balise de script, ces API de niveau supérieur sont disponibles sur le plan global «`React`».

Si vous utilisez ES6 avec npm, vous pouvez écrire `import React from 'react'`. Si vous utilisez ES5 avec npm, vous pouvez écrire `var React = require('react')`.

Composants

Les composants React vous permettent de diviser l'interface utilisateur en pièces indépendantes et réutilisables, et de penser à chaque pièce isolément. Les composants React peuvent être définis en sous-classant `React.Component` ou `React.PureComponent`.

Conceptuellement, les composants sont comme des fonctions JavaScript. Ils acceptent des entrées arbitraires (**appelées «props»**) et renvoient des éléments React décrivant ce qui devrait apparaître à l'écran.

Composants fonctionnels et de classe

La façon la plus **simple** de définir un composant consiste à écrire une **fonction JavaScript**:

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

Cette fonction est un composant React valide car elle accepte un seul objet “**props**” avec des données et renvoie un élément React. Nous appelons ces **composants «fonctionnels»** parce qu’ils sont littéralement des fonctions JavaScript.

Vous pouvez également utiliser une classe [ES6](#) pour définir un composant:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Les deux composants ci-dessus sont équivalents du point de vue de React. **Les classes ont des fonctionnalités supplémentaires.**

Rendu d'un composant

Auparavant, nous n'avons rencontré que des éléments React représentant des balises DOM:

```
const element = <div/>;
```

Cependant, les éléments peuvent également représenter des composants définis par l'utilisateur:

```
const element = <Welcome name="Sara" />;
```

Lorsque React voit un élément représentant un composant défini par l'utilisateur, il transmet les attributs JSX à ce composant en tant qu'objet unique. Nous appelons cet objet “**props**”.

Par exemple, ce code rend “Hello, Sara” sur la page:

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

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

[Essayez sur CodePen.](#)

Reprendons ce qui se passe dans cet exemple:

1. Nous appelons `ReactDOM.render()` avec l'élément `<Welcome name="Sara" />`.
2. Réagissez appelle le composant `Welcome` avec `{name: 'Sara'}` comme “**props**”.
3. Notre composant `Welcome` renvoie un élément `<h1> Hello, Sara </ h1>` comme résultat.
4. **React** met à jour le DOM pour correspondre `<h1> Bonjour, Sara </ h1>`.

Attention:

Prefixez toujours les noms des composants avec une majuscule.

Par exemple, `<div />` représente une balise DOM, mais `<Welcome />` représente un composant et nécessite que `Welcome` soit dans la portée.

Composants

Les composants peuvent se référer à d'autres composants. Cela permet d'utiliser la même abstraction de composant pour n'importe quel niveau de détail. *Un bouton, un formulaire, une boîte de dialogue, un écran*: dans les applications React, ce sont des composants.

Par exemple, nous pouvons créer un composant `App` qui rend `Welcome` plusieurs fois:

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

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}

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

[Essayez sur CodePen.](#)

En règle générale, les nouvelles applications React disposent d'un seul composant `App` au sommet. Cependant, si vous intégrez React à une application existante, vous pouvez démarrer de bas en haut avec un petit composant comme `Button` et progressivement vous diriger vers le haut de la hiérarchie des vues.

Attention:

Les composants doivent renvoyer un seul élément racine. C'est pourquoi nous avons ajouté un `<div>` pour contenir tous les éléments `<Welcome />`.

Extraction de composants

Diviser les composants en composants plus petits.

Par exemple, considérez ce composant `Comment` :

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}>
      />
      <div className="UserInfo-name">
        {props.author.name}>
      /</div>
    /<div>
    <div className="Comment-text">
      {props.text}>
    /<div>
    <div className="Comment-date">
      {formatDate(props.date)}>
    /<div>
  /<div>
);
}
```

[Essayez sur CodePen.](#)

Il accepte `author` (un objet), `text` (une chaîne) et `date` (date) comme `props`.

Cette composante peut être difficile à changer en raison de l'ensemble de la nidification, et il est également difficile de réutiliser certaines parties de celui-ci.

Tout d'abord, nous allons extraire `Avatar` :

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}>
  />
);
}
```

Le composant `Avatar` n'a pas besoin de savoir qu'il est rendu dans un `Comment` .

Il est recommandé de nommer des **props** du point de vue du composant plutôt que du contexte dans lequel il est utilisé.

Nous pouvons maintenant simplifier `Comment` un petit peu:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Ensuite, nous allons extraire un composant `UserInfo` qui rend un `Avatar` à côté du nom de l'utilisateur:

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

Cela nous permet de simplifier encore `Comment` :

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

[Essayez sur CodePen.](#)

C'est une bonne pratique dans une application arge d'utiliser une palette de composants.

Une bonne règle empirique est que si une partie de votre interface utilisateur est utilisée plusieurs fois (`Button` , `Panel` , `Avatar`), ou si elle est assez complexe (`App` , `FeedStory` , `Comment`), c'est un candidat pour être un composant réutilisable.

Les props sont en lecture seule.

Lorsque vous déclarez un composant, il ne doit jamais modifier ses **propriétés**. Considérez cette fonction `sum` :

```
function sum(a, b) {  
  return a + b;  
}
```

De telles fonctions sont appelées “[pure](#)” car elles n’essayent pas de modifier leurs entrées et renvoient toujours le même résultat pour les mêmes entrées.

En revanche, cette fonction est impure car elle modifie sa propre entrée:

```
function withdraw(account, amount) {  
  account.total -= amount;  
}
```

React est assez souple, mais il a une seule règle stricte:

Tous les composants React doivent agir comme des fonctions pures par rapport à leurs “props”.

Création de composant de vues. Cycle de vie.

Dans cette section, nous allons apprendre à rendre le composant `Clock` réutilisable et encapsulé. **Il mettra en place sa propre minuterie et se mettra à jour chaque seconde.**

Nous pouvons commencer par encapsuler le layout de `Clock` :

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

[Essayez sur CodePen.](#)

Cependant, il manque une exigence cruciale: le fait que l'horloge mette en place une minuterie et mette à jour l'interface utilisateur chaque seconde devrait être un détail d'implémentation de l'horloge.

Idéalement, nous voulons rendre le composant une seule fois et déclencher sa mise à jour.

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

Pour implémenter cela, nous devons ajouter “**state**” au composant `Clock`.

Le “**state**” est similaire aux “**props**”, mais il est privé et entièrement contrôlé par le composant.

Les composants définis comme classes possèdent des fonctionnalités supplémentaires. Le `local state` est exactement cela: une fonction disponible uniquement pour les classes.

Conversion d'une fonction en classe

Vous pouvez convertir un composant fonctionnel comme `Clock` en une classe en cinq étapes:

1. Créez une classe [ES6](#) avec le même nom qui étend `React.Component`.
2. Ajoutez une seule méthode vide appelée `render()`.
3. Déplacez le corps de la fonction dans la méthode `render()`.
4. Remplacez `props` par `this.props` dans le corps `render()`.
5. Supprimez la déclaration de fonction vide restante.

```
class Clock extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1>Hello, world!</h1>  
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>  
      </div>  
    );  
  }  
}
```

[Essayez sur CodePen.](#)

`Clock` est maintenant défini comme une classe plutôt qu'une fonction.

Cela nous permet d'utiliser des fonctionnalités supplémentaires telles que l'état local et les crochets du cycle de vie.

Ajout d'un état local à une classe

Nous allons déplacer la `date` des accessoires à l'état en trois étapes:

- 1) Remplacez `this.props.date` par `this.state.date` dans la méthode `render()`:

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

- 2) Ajoutez un constructeur de classe qui attribue l'état initial `this.state`:

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

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Notez comment nous passons `props` au constructeur de base:

```
constructor(props) {
  super(props);
  this.state = {date: new Date()};
}
```

Les composants de la classe doivent toujours appeler le constructeur de base avec `props`.

- 3) Supprimez le support `date` de l'élément `<Clock />`:

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

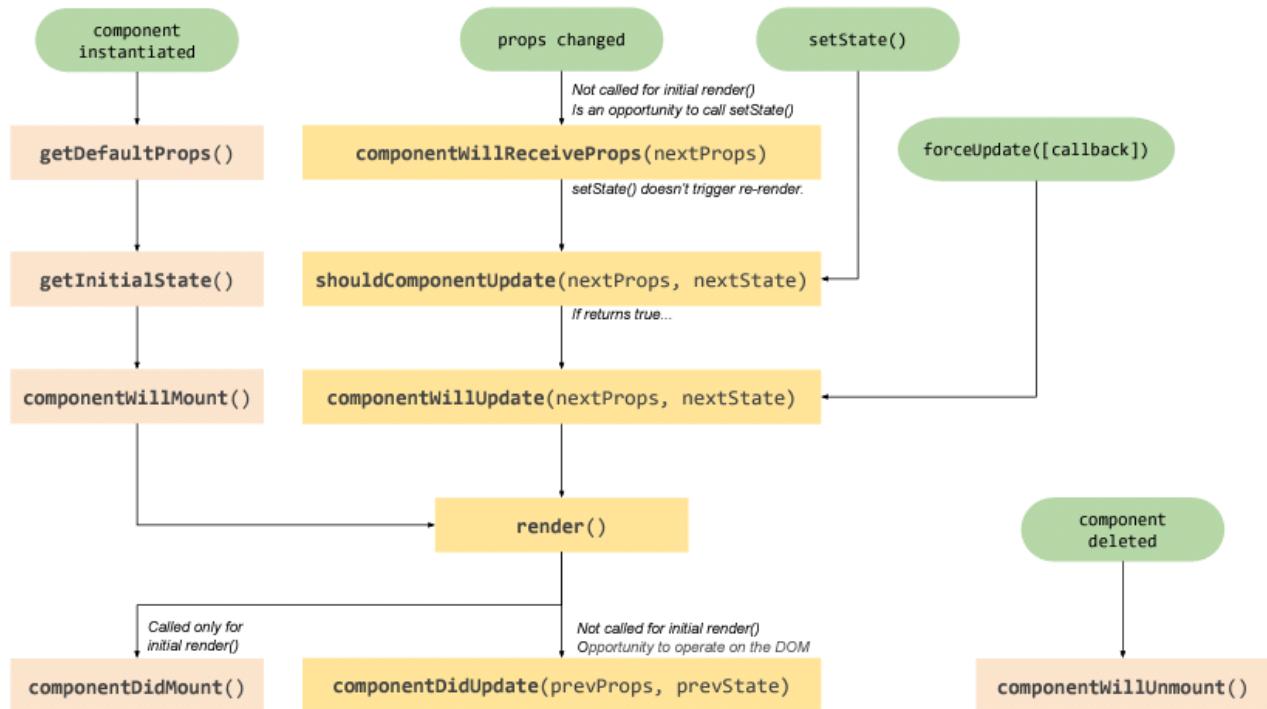
Nous ajouterons ensuite le code DU timer au composant lui-même.

Le résultat est le suivant:

```
class Clock extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {date: new Date()};  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>Hello, world!</h1>  
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>  
      </div>  
    );  
  }  
}  
  
ReactDOM.render(  
  <Clock />,  
  document.getElementById('root')  
)
```

[Essayez sur CodePen.](#)

Ajout de méthodes de cycle de vie à une classe



Dans les applications avec de nombreux composants, il est très important de libérer les ressources prises par les composants quand ils sont détruits.

Nous voulons [configurer une minuterie](#) chaque fois que `Clock` est rendu au DOM pour la première fois. C'est ce qu'on appelle le «**mounting**» dans React.

Nous voulons également [effacer cette temporisation](#) chaque fois que le DOM produit par `Clock` est supprimé. C'est ce qu'on appelle «**unmounting**» dans React.

Nous pouvons déclarer des méthodes spéciales sur la classe de composant pour exécuter du code lorsqu'un composant est monté et démonté:

```

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

  componentDidMount() {

  }

  componentWillUnmount() {

  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

```

Ces méthodes sont appelées «crochets(hooks)du cycle de vie».

Le hook `componentDidMount()` s'exécute après que le du composant soit rendu dans le DOM.

```

componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}

```

Notez comment nous sauvegardons l'ID de minuterie sur `this`.

Alors que `this.props` est configuré par React lui-même et `this.state` a une signification spéciale, **vous êtes libre d'ajouter des champs supplémentaires à la classe manuellement.**

Si vous n'utilisez pas quelque chose dans `render()`, il ne devrait pas être dans l'état.

Nous allons détruire la minuterie dans le crochet du composant `componentWillUnmount()`:

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

Enfin, nous allons mettre en œuvre la méthode `tick()` qui s'exécute toutes les secondes.

Il utilisera `this.setState()` pour planifier les mises à jour de l'état local du composant:

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

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

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

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

[Essayez sur CodePen.](#)

Récapitulif:

- 1) Lorsque `<Clock />` est passé à `ReactDOM.render()`, React appelle le constructeur du composant `Clock`. Puisque `Clock` a besoin d'afficher l'heure actuelle, il initialise `this.state` avec un objet incluant l'heure actuelle.
- 2) React appelle la méthode `render()` du composant `Clock` pour mettre à jour le DOM.
- 3) Lorsque le composant `Clock` est inséré dans le DOM, React appelle le crochet `componentDidMount()` du cycle de vie.
- 4) Chaque seconde, le navigateur appelle la méthode `tick()` qui planifie une mise à jour de l'interface utilisateur en appelant `setState()`. Grâce à l'appel `setState()`, React sait que l'état a changé et appelle la méthode `render()` pour apprendre ce qui devrait être à l'écran.
- 5) Si le composant `Clock` est supprimé du DOM, React appelle `componentWillUnmount()`.

Utilisation correcte de l'état

Il ya trois choses que vous devez savoir sur `setState ()`.

Ne pas modifier l'état directement

Par exemple, cela ne rendra pas un composant:

```
// Wrong
this.state.comment = 'Hello';
```

Au lieu de cela, utilisez `setState ()`:

```
// Correct
this.setState({comment: 'Hello'});
```

Le seul endroit où vous pouvez assigner `this.state` est le constructeur.

Les mises à jour d'état peuvent être asynchrones.

React peut regrouper plusieurs appels `setState()` en une seule mise à jour pour la performance.

Parce que `this.props` et `this.state` peuvent être mis à jour de façon asynchrone, vous ne devez pas compter sur leurs valeurs pour calculer l'état suivant.

Utilisation, utilisez une seconde forme de `setState ()` qui accepte une fonction plutôt qu'un objet.

Cette fonction recevra l'état précédent comme premier argument et les accessoires au moment où la mise à jour est appliquée comme deuxième argument.

```
// Correct
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}));
```

ou

```
// Correct
this.setState(function(prevState, props) {
  return {
    counter: prevState.counter + props.increment
  };
});
```

Mises à jour d'état sont fusionnées

Lorsque vous appelez `setState()`, React fusionne l'objet que vous fournissez dans l'état actuel.

Par exemple, votre état peut contenir plusieurs variables indépendantes:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

Vous pouvez ensuite les mettre à jour indépendamment avec des appels `setState()` distincts:

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

La fusion est peu profonde, donc `this.setState ({comments})` laisse `this.state.posts` intacte, mais remplace complètement `this.state.comments`.

Flux de données descendants.

Les composants ne devrait pas se préoccuper des états des parents ou enfants ou leur implémentation.

C'est pourquoi le `state` est souvent appelé local ou encapsulé. **Il n'est pas accessible à un composant autre que celui qui le possède et le définit.**

Un composant peut choisir de passer son état à ses composants enfants:

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

Cela fonctionne également pour les composants définis par l'utilisateur:

```
<FormattedMessage date={this.state.date} />
```

Le composant `FormattedMessage` reçoit la `date` dans ses “**props**” sans en connaître la provenance.

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

[Essayez sur CodePen.](#)

C'est ce qu'on appelle couramment un **flux de données “descendant” ou “unidirectionnel”**. Tout état est toujours détenu par un composant spécifique et toute donnée ou interface utilisateur dérivée de cet état ne peut **affecter que les composants «en dessous» dans l'arborescence.**

Si vous imaginez un arbre de composant comme une cascade d'accessoires, l'état de chaque composant est comme une source d'eau supplémentaire qui se joint à un point arbitraire, mais coule aussi vers le bas.

Pour montrer que tous les composants sont vraiment isolés, nous pouvons créer un composant `App` qui rend trois `<Clock>`s:

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}

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

[Essayez sur CodePen.](#)

Chaque «Horloge» configure sa propre minuterie et les met à jour indépendamment.



Code Applicatif

Cette section vous donne un aperçu de l'organisation du code source de React, de ses conventions et de son implémentation.

Si vous souhaitez contribuer à React, nous espérons que ce guide vous aidera à vous sentir plus à l'aise pour apporter des modifications.

Nous ne recommandons pas nécessairement ces conventions dans les applications React. Nombre d'entre elles existent pour des raisons historiques et sont susceptibles d'évoluer avec le temps.

Dossiers racines {#top-level-folders}

Après avoir cloné le [dépôt React](#), vous verrez quelques dossiers racines :

- `packages` contient des métadonnées (telles que `package.json`) et le code source (sous-répertoire `src`) de tous les paquets du dépôt React. **Si votre modification est liée au code, vous passerez le plus clair de votre temps dans le sous-répertoire `src` des différents paquets.**
- `fixtures` contient quelques petites applications React de test pour les contributeurs.
- `build` est la sortie de construction de React. Il ne figure pas dans le dépôt, mais il apparaîtra dans votre clone de React après que vous [l'aurez construit](#) pour la première fois.

La documentation est hébergée [dans un dépôt distinct de React](#).

Il existe quelques autres dossiers racines, mais ils sont principalement utilisés par l'outil et vous n'aurez probablement jamais affaire à eux lorsque vous contribuerez.

Tests colocalisés {#colocated-tests}

Nous n'avons pas de répertoire racine pour les tests unitaires. Nous les plaçons plutôt dans un répertoire appelé `__tests__` situé à côté des fichiers qu'ils testent.

Par exemple, un test pour `setInnerHTML.js` sera placé juste à côté, dans `__tests__/setInnerHTML-test.js`.

Avertissements et invariants {#warnings-and-invariants}

Le code source de React utilise `console.error` pour afficher les avertissements :

```
if (__DEV__) {
  console.error('Il y a un souci.');
}
```

Les avertissements ne sont activés que dans la phase de développement. En production, ils sont complètement retirés du code. Si vous avez besoin d'interdire l'exécution d'une partie de code, utilisez plutôt le module `invariant` :

```
var invariant = require('invariant');

invariant(
```

```
2 + 2 === 4,  
'Vous ne passerez pas !'  
);
```

L'invariant est levé lorsque la condition de invariant est false.

Le terme « invariant » signifie simplement « cette condition est toujours vraie ». Vous pouvez voir ça comme une affirmation.

Pour les invariants, il est important d'avoir un comportement similaire en développement et en production, afin qu'ils soient levés dans les deux cas. Les messages d'erreur sont automatiquement remplacés par des codes d'erreur en production afin d'éviter toute incidence négative sur la taille (en octets) du fichier.

Développement et production {#development-and-production}

Vous pouvez utiliser la variable pseudo-globale `__DEV__` dans le code source pour délimiter les blocs de code réservés au développement.

La variable est remplacée lors de la compilation et se transforme en contrôles `process.env.NODE_ENV !== 'production'` dans les *builds* CommonJS.

Pour les versions autonomes, la variable devient `true` dans la version non-minifiée du fichier produit, alors qu'elle est complètement effacée, ainsi que les blocs `if` qu'elle contrôle, dans la version minifiée.

```
if (__DEV__) {  
  // Ce code va uniquement s'appliquer pendant le développement.  
}
```

Flow {#flow}

Nous avons récemment commencé à introduire des contrôles `Flow` dans le code source. Les fichiers marqués avec l'annotation `@flow` dans le commentaire d'en-tête de licence sont soumis à vérification.

Nous acceptons les *pull requests* qui ajoutent des annotations Flow au code existant. Les annotations Flow ressemblent à ceci :

```
ReactRef.detachRefs = function(  
  instance: ReactInstance,  
  element: ReactElement | string | number | null | false,  
): void {  
  // ...  
}
```

Dans la mesure du possible, le nouveau code devrait utiliser des annotations Flow. Vous pouvez exécuter `yarn flow` localement pour vérifier votre code avec Flow.

Plusieurs paquets {#multiple-packages}

React est un [monorepo](#). Son dépôt contient plusieurs paquets distincts afin que leurs modifications puissent être coordonnées et que les problèmes puissent être signalés dans un seul et même endroit.

Le noyau de React {#react-core}

Le « noyau » de React inclut toutes les [API React de niveau racine](#), par exemple :

- `React.createElement()`
- `React.Component`
- `React.Children`

Le noyau React n'inclut que les API nécessaires à la définition des composants. Il n'inclut pas l'algorithme de [réconciliation](#) ni aucun code spécifique à une plate-forme. Il est utilisé à la fois par les composants de React DOM et de React Native.

Le code pour le noyau React se trouve dans `packages/react` au sein de l'arborescence source. Il est disponible sur npm via le module `react`. La version autonome correspondante pour l'utilisation à même le navigateur est appelée `react.js`, et exporte une variable globale appelée `React`.

Moteurs de rendu {#renderers}

React a été créé à l'origine pour le DOM, mais il a ensuite été adapté pour prendre également en charge les plates-formes natives avec [React Native](#). C'est ainsi qu'est né le concept de « moteurs de rendu » (*renderers, terme que nous utiliserons sans italiques dans la suite de ce texte, NdT*) au sein de React.

Les renderers gèrent la transformation d'une arborescence React en appels à la plate-forme sous-jacente.

Les renderers sont également situés dans `packages/` :

- [Le renderer de React DOM](#) retranscrit les composants React dans le DOM. Il implémente [les API ReactDOM racines](#) et est disponible via le module npm `react-dom`. Il peut aussi être utilisé en tant que *bundle* autonome dans le navigateur, lequel est nommé `react-dom.js` et exporte une variable globale `ReactDOM`.
- [Le renderer de React Native](#) retranscrit les composants React sous forme de vues natives. Il est utilisé en interne par React Native.
- [Le renderer de test de React](#) retranscrit les composants React sous forme d'arbres JSON. Il est utilisé par la fonctionnalité d'[instantanés \(snapshots, NdT\)](#) de [Jest](#) et est disponible via le module npm `react-test-renderer`.

Le seul autre moteur de rendu officiellement pris en charge est `react-art`. Auparavant, il se trouvait dans un dépôt GitHub séparé, mais nous l'avons déplacé dans l'arborescence source principale pour le moment.

Remarque

Techniquement, le `react-native-renderer` est une couche très mince qui apprend à React à interagir avec l'implémentation de React Native. Le véritable code spécifique à la plate-forme, qui gère les vues natives et fournit les composants, réside quant à lui dans le dépôt [React Native](#).

Réconciliateurs {#reconcilers}

Même des moteurs de rendu très différents comme React DOM et React Native doivent partager beaucoup de logique. En particulier, l'algorithme de [réconciliation](#) doit être aussi similaire que possible afin que le rendu déclaratif, les composants personnalisés, l'état local, les méthodes de cycle de vie et les refs fonctionnent de manière cohérente sur toutes les plates-formes prises en charge.

Pour résoudre ce problème, différents moteurs de rendu partagent du code entre eux. Nous appelons cette partie de React un « réconciliateur ». Lorsqu'une mise à jour telle que `setState()` est planifiée, le réconciliateur appelle `render()` sur les composants de l'arborescence et les monte, les met à jour ou les démonte.

Les réconciliateurs ne font pas l'objet de modules séparés, car ils ne disposent actuellement d'aucune API publique. Ils sont exclusivement utilisés par les moteurs de rendu tels que React DOM et React Native.

Réconciliateur Stack {#stack-reconciler}

Le réconciliateur “*stack*” est l'implémentation qui sous-tend React 15 et les versions antérieures. Nous avons depuis cessé de l'utiliser, mais il reste décrit en détail dans la [prochaine page](#).

Réconciliateur Fiber {#fiber-reconciler}

Le réconciliateur “*fiber*” représente une nouvelle tentative de résoudre les problèmes inhérents au réconciliateur “*stack*” en plus de quelques problèmes anciens. C'est le réconciliateur par défaut depuis React 16.

Ses objectifs principaux sont :

- la capacité à diviser un travail interruptible en segments ;
- la capacité à hiérarchiser, déplacer et réutiliser des travaux en cours ;
- la capacité à jongler entre parents et enfants pour exécuter une mise en page avec React ;
- la capacité à renvoyer plusieurs éléments depuis `render()` ;
- une meilleure prise en charge des périmètres d'erreur.

Vous pouvez en apprendre davantage sur l'architecture React Fiber [ici](#) et [ici](#). Bien qu'elles soient livrées avec React 16, les fonctionnalités asynchrones ne sont pas encore activées par défaut.

Son code source est situé dans [packages/react-reconciler](#).

Système d'événements {#event-system}

React implémente une abstraction par-dessus les événements natifs afin de lisser les disparités d'un navigateur à l'autre. Son code source se trouve dans [packages/react-dom/src/events](#).

Et maintenant ? {#what-next}

Lisez la [prochaine page](#) pour en apprendre davantage sur l'implémentation du réconciliateur utilisé avant React 16. Nous n'avons pas encore documenté les détails internes d'implémentation du nouveau réconciliateur.

Bundling {#bundling}

La plupart des applications React empaquetteront leur fichiers au moyen d'outils tels que [Webpack](#), [Rollup](#) ou [Browserify](#). L'empaquetage (*bundling*, NdT) consiste à suivre le graphe des importations dans les fichiers, et à les regrouper au sein d'un même fichier : un *bundle* (*terme que nous utiliserons sans italiques dans la suite de la page, NdT*). Ce bundle peut ensuite être inclus dans une page web pour charger une application entière d'un seul coup.

Exemple {#example}

Application :

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

Bundle :

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

console.log(add(16, 26)); // 42
```

Remarque

Vos bundles finiront par être très différents de ceux-là.

Si vous utilisez [Create React App](#), [Next.js](#), [Gatsby](#) ou un outil similaire, vous bénéficierez d'une configuration de Webpack prête à l'emploi pour créer le bundle de votre application.

Si ce n'est pas le cas, vous devrez configurer vous-même la génération de votre bundle. Consultez les guides [d'installation](#) et de [démarrage](#) de Webpack.

Le découpage dynamique de code {#code-splitting}

Les bundles c'est génial, mais au fur et à mesure que votre application grandit, votre bundle va grossir aussi. Surtout si vous intégrez de grosses bibliothèques tierces. Vous devez garder un œil sur le code que vous

intégrer dans votre bundle pour éviter de le rendre si lourd que le chargement de votre application prendrait beaucoup de temps.

Pour éviter de vous retrouver avec un bundle trop volumineux, il est bon d'anticiper les problèmes et de commencer à fractionner votre bundle. Le découpage dynamique de code est une fonctionnalité prise en charge par des empaqueteurs tels que [Webpack](#), [Rollup](#) ou [Browserify](#) (via [factor-bundle](#)), qui permet de créer plusieurs bundles pouvant être chargés dynamiquement au moment de l'exécution.

Fractionner votre application peut vous aider à charger à la demande (*lazy-load, NdT*) les parties qui sont nécessaires pour l'utilisateur à un moment donné, ce qui peut améliorer considérablement les performances de votre application. Bien que vous n'ayez pas réduit la quantité de code de votre application, vous évitez de charger du code dont l'utilisateur n'aura peut-être jamais besoin, et réduisez la quantité de code nécessaire au chargement initial.

import() {#import}

La meilleure façon d'introduire du découpage dynamique de code dans votre application consiste à utiliser la syntaxe d'`import()` dynamique.

Avant :

```
import { add } from './math';
console.log(add(16, 26));
```

Après :

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

Lorsque Webpack rencontre cette syntaxe, il commence automatiquement à découper le code de votre application. Si vous utilisez Create React App, c'est déjà configuré pour vous et vous pouvez [l'utiliser](#) immédiatement. C'est également pris en charge de base par [Next.js](#).

Si vous configurez Webpack vous-même, vous voudrez sans doute lire le [guide sur le découpage dynamique de code](#) de Webpack. Votre configuration Webpack devrait vaguement ressembler à ça.

Si vous utilisez [Babel](#), vous devrez vous assurer que Babel peut comprendre la syntaxe d'import dynamique mais ne la transforme pas. Pour cela, vous aurez besoin de l'extension [babel-plugin-syntax-dynamic-import](#).

React.lazy {#reactlazy}

Remarque

`React.lazy` et `Suspense` ne sont pas encore disponibles pour le rendu côté serveur. Si vous souhaitez fractionner votre code dans une application rendue côté serveur, nous vous recommandons d'utiliser

[Loadable Components](#). Il propose un [guide pratique pour fractionner le bundle avec un rendu côté serveur](#).

La fonction `React.lazy` vous permet d'afficher un composant importé dynamiquement comme n'importe quel autre composant.

Avant :

```
import OtherComponent from './OtherComponent';
```

Après :

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

Ça chargera automatiquement le bundle contenant le composant `OtherComponent` quand celui-ci sera rendu pour la première fois.

`React.lazy` prend une fonction qui doit appeler un `import()` dynamique. Ça doit renvoyer une `Promise` qui s'accomplit avec un module dont l'export par défaut contient un composant React.

Le composant importé dynamiquement devrait être exploité dans un composant `Suspense`, qui nous permet d'afficher un contenu de repli (ex. un indicateur de chargement) en attendant que ce module soit chargé.

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

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Chargement...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

La prop `fallback` accepte n'importe quel élément React que vous souhaitez afficher en attendant le chargement du composant. Vous pouvez placer le composant `Suspense` n'importe où au-dessus du composant chargé à la demande. Vous pouvez même envelopper plusieurs composants chargés à la demande avec un seul composant `Suspense`.

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

const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

```

const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Chargement...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}

```

Périmètres d'erreurs {#error-boundaries}

Si le chargement de l'autre module échoue (par exemple à cause d'une défaillance réseau), une erreur sera levée. Vous pouvez gérer ces erreurs pour assurer une expérience utilisateur agréable et retomber sur vos pieds avec [les périmètres d'erreurs](#) (*Error boundaries, NdT*). Une fois que vous avez créé votre périmètre d'erreur, vous pouvez l'utiliser n'importe où au-dessus de vos composants chargés à la demande pour afficher un état d'erreur lors d'une défaillance réseau.

```

import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Chargement...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);

```

Découpage dynamique de code basé sur les routes {#route-based-code-splitting}

Décider où introduire un découpage dynamique de code dans votre application peut s'avérer délicat. Vous voulez être sûr·e de choisir des endroits qui fractionnent les bundles de manière uniforme, sans perturber l'expérience utilisateur.

Les routes sont un bon endroit pour commencer. La plupart des gens sont habitués sur le web à ce que les transitions entre les pages mettent du temps à charger. Vous aurez également tendance à ré-afficher la page entière d'un bloc, de sorte qu'il est peu probable que vos utilisateurs interagissent avec d'autres éléments de la page pendant ce temps-là.

Voici un exemple de configuration du découpage dynamique de code basé sur les routes de votre application, qui utilise une bibliothèque comme [React Router](#) avec [React.lazy](#).

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Changement...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
);
```

Exports nommés {#named-exports}

Pour le moment, [React.lazy](#) ne prend en charge que les exports par défaut. Si le module que vous souhaitez importer utilise des exports nommés, vous pouvez créer un module intermédiaire qui réexportera le composant voulu en tant qu'export par défaut. Cela garantit que le *tree shaking* ([procédé permettant de supprimer les exports non-exploités, NdT](#)) continuera à fonctionner et que vous n'embarquerez pas de composants inutilisés.

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

React fournit un puissant modèle de composition, aussi nous recommandons d'utiliser la composition plutôt que l'héritage pour réutiliser du code entre les composants.

Dans cette section, nous examinerons quelques situations pour lesquelles les débutants en React ont tendance à opter pour l'héritage, et montrerons comment les résoudre à l'aide de la composition.

Délégation de contenu {#containment}

Certains composants ne connaissent pas leurs enfants à l'avance. C'est particulièrement courant pour des composants comme `Sidebar` ou `Dialog`, qui représentent des blocs génériques.

Pour de tels composants, nous vous conseillons d'utiliser la prop spéciale `children`, pour passer directement les éléments enfants dans votre sortie :

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

Ça permet aux autres composants de leur passer des enfants quelconques en imbriquant le JSX :

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Bienvenue
      </h1>
      <p className="Dialog-message">
        Merci de visiter notre vaisseau spatial !
      </p>
    </FancyBorder>
  );
}
```

[Essayer sur CodePen](#)

Tout ce qui se trouve dans la balise JSX `<FancyBorder>` est passé comme prop `children` au composant `FancyBorder`. Puisque `FancyBorder` utilise `{props.children}` dans une balise `<div>`, les éléments passés apparaissent dans la sortie finale.

Bien que cela soit moins courant, vous aurez parfois besoin de plusieurs « trous » dans un composant. Dans ces cas-là, vous pouvez créer votre propre convention au lieu d'utiliser `children` :

```

function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}

```

[Essayer sur CodePen](#)

Des éléments React tels que `<Contacts />` et `<Chat />` sont simplement des objets, vous pouvez les passer comme props au même titre que n'importe quelle autre donnée. Cette approche peut vous rappeler la notion de "slots" présente dans d'autres bibliothèques, mais il n'y a aucune limitation à ce que vous pouvez passer en props avec React.

Spécialisation {#specialization}

Parfois, nous voyons nos composants comme des « cas particuliers » d'autres composants. Par exemple, nous pourrions dire que `WelcomeDialog` est un cas particulier de `Dialog`.

Avec React, on réalise aussi ça avec la composition ; un composant plus « spécialisé » utilise un composant plus « générique » et le configure grâce aux props :

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function FancyBorder({color}) {
  return (
    <div style={{border: `2px solid ${color}`}}>
      <div>

```

```

    );
}

function WelcomeDialog() {
  return (
    <Dialog
      title="Bienvenue"
      message="Merci de visiter notre vaisseau spatial !"
    />;
}

```

Essayer sur CodePen

La composition fonctionne tout aussi bien pour les composants à base de classe :

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}
    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Programme d'exploration de Mars"
        message="Comment devrions-nous nous adresser à vous ?">
        <input value={this.state.login}
          onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
          Inscrivez-moi !
        </button>
      </Dialog>
    );
  }

  handleChange(e) {
    this.setState({login: e.target.value});
  }
}

```

```
}

handleSignUp() {
  alert(`Bienvenue à bord, ${this.state.login} !`);
}
}
```

[Essayer sur CodePen](#)

Qu'en est-il de l'héritage ? {#so-what-about-inheritance}

Chez Facebook, nous utilisons React pour des milliers de composants, et nous n'avons pas encore trouvé de cas où nous aurions recommandé de créer des hiérarchies d'héritage de composants.

Les props et la composition vous donnent toute la flexibilité dont vous avez besoin pour personnaliser l'apparence et le comportement d'un composant de manière explicite et sûre. Souvenez-vous qu'un composant peut accepter tout type de props, y compris des valeurs primitives, des éléments React et des fonctions.

Si vous souhaitez réutiliser des fonctionnalités sans rapport à l'interface utilisateur entre les composants, nous vous suggérons de les extraire dans un module Javascript séparé. Les composants pourront alors importer cette fonction, cet objet ou cette classe sans avoir à l'étendre.



PropTypes

Remarque

`React.PropTypes` a été déplacé dans un autre module depuis React v15.5. Merci de plutôt utiliser [le module prop-types](#).

Nous fournissons [un script codemod](#) pour automatiser cette transition.

Au fur et à mesure que votre application grandit, vous pouvez détecter un grand nombre de bugs grâce à la validation de types. Dans certains cas, vous pouvez utiliser des extensions JavaScript comme [Flow](#) ou [TypeScript](#) pour valider les types de toute votre application. Mais même si vous ne les utilisez pas, React possède ses propres fonctionnalités de validation de types. Pour lancer la validation de types des props d'un composant, vous pouvez ajouter la propriété spéciale `propTypes` :

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Bonjour, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

Dans cet exemple nous utilisons un composant à base de classe, mais ça reste vrai pour les fonctions composants et les composants créés avec `React.memo` ou `React.forwardRef`.

`PropTypes` exporte un ensemble de validateurs qui peuvent être utilisés pour s'assurer que la donnée que vous recevez est valide. Dans cet exemple, nous utilisons `PropTypes.string`. Quand une valeur non valide est fournie à une prop, un message d'avertissement apparaîtra dans la console JavaScript. Pour des raisons de performances, `propTypes` n'est vérifiée qu'en mode développement.

PropTypes {#propTypes}

Voici un exemple qui détaille les différents validateurs fournis :

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // Vous pouvez déclarer qu'une prop est d'un certain type JS. Par défaut,
  // elles sont toutes optionnelles.
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
```

```

optionalString: PropTypes.string,
optionalSymbol: PropTypes.symbol,

// Tout ce qui peut apparaître dans le rendu : des nombres, des chaînes de
// caractères, des éléments ou des tableaux (ou fragments) contenant ces types.
optionalNode: PropTypes.node,

// Un élément React.
optionalElement: PropTypes.element,

// Un type d'élément React (ex. MyComponent).
optionalElementType: PropTypes.elementType,

// Vous pouvez aussi déclarer qu'une prop est une instance d'une classe.
// On utilise pour ça l'opérateur JS instanceof.
optionalMessage: PropTypes.instanceOf(Message),

// Vous pouvez vous assurer que votre prop est limitée à certaines
// valeurs spécifiques en la traitant comme une enumération.
optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// Cette prop peut être de n'importe lequel de ces trois types
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),

// Un tableau avec des valeurs d'un certain type
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// Un objet avec des valeurs d'un certain type
optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// Un objet avec une forme spécifique
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),

// An object with warnings on extra properties
optionalObjectWithStrictShape: PropTypes.exact({
  name: PropTypes.string,
  quantity: PropTypes.number
}),

// Vous pouvez ajouter `isRequired` à la fin de n'importe lequel des validateurs
// ci-dessus pour vous assurer qu'un message d'avertissement s'affiche lorsque
// la prop n'est pas fournie.
requiredFunc: PropTypes.func.isRequired,

// Cette prop est requise et peut être de n'importe quel type
requiredAny: PropTypes.any.isRequired,

```

```

// Vous pouvez aussi spécifier un validateur personnalisé. Il devra renvoyer
// un objet Error si la validation échoue. N'utilisez pas de `console.warn`
// ou `throw`, car ça ne fonctionnera pas dans `oneOfType`.
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error(
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
},
```
// Vous pouvez aussi fournir un validateur personnalisé à `arrayOf` et
`objectOf`.

// Il faudra renvoyer un objet Error si la validation échoue. Le validateur
// sera appelé pour chaque clé du tableau ou de l'objet. Les deux premiers
// arguments du validateur sont le tableau ou l'objet lui-même, et la clé
// de la valeur actuelle.
customArrayProp: PropTypes.arrayOf(
 function(propValue, key, componentName, location, propFullName) {
 if (!/matchme/.test(propValue[key])) {
 return new Error(
 'Invalid prop `' + propFullName + '` supplied to' +
 ' `' + componentName + '`. Validation failed.'
);
 }
 }
);
```

```

Exiger un seul enfant {#requiring-single-child}

Avec `PropTypes.element`, vous pouvez spécifier qu'un seul enfant peut être passé à un composant.

```

import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  render() {
    // Ça doit être un élément unique ou un avertissement sera affiché.
    const children = this.props.children;
    return (
      <div>
        {children}
      </div>
    );
  }
}

MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};

```

Valeurs par défaut des props {#default-prop-values}

Vous pouvez définir des valeurs par défaut pour vos `props` en utilisant la propriété spéciale `defaultProps` :

```
class Greeting extends React.Component {
  render() {
    return (
      <h1>Bonjour, {this.props.name}</h1>
    );
  }
}

// Spécifie les valeurs par défaut des props :
Greeting.defaultProps = {
  name: 'bel inconnu'
};

// Affiche « Bonjour, bel inconnu » :
ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);
```

Si vous utilisez une transformation Babel telle que `transform-class-properties`, vous pouvez aussi déclarer `defaultProps` comme propriété statique dans une classe de composant React. Cependant, cette syntaxe n'a pas encore été finalisée et requiert une étape de compilation supplémentaire pour fonctionner dans un navigateur. Pour plus d'informations, voir la [proposition des aspects statiques de classe](#).

```
class Greeting extends React.Component {
  static defaultProps = {
    name: 'bel inconnu'
  }

  render() {
    return (
      <div>Bonjour, {this.props.name}</div>
    )
  }
}
```

Les `defaultProps` seront utilisées pour s'assurer que `this.props.name` aura une valeur si elle n'était pas spécifiée par le composant parent. La validation de types des `propTypes` aura lieu après que `defaultProps` est résolu, la validation de types s'applique donc également aux `defaultProps`.



Test

Vous pouvez tester vos composants React au même titre que le reste de votre code JavaScript.

Il existe plusieurs façons de tester des composants React, lesquelles se divisent au final en deux grandes catégories :

- **Effectuer le rendu d'arborescences de composants** dans un environnement de test simplifié, et vérifier la sortie.
- **Exécuter une appli complète** dans un environnement navigateur réaliste (on parle alors de tests « de bout en bout ») (*end-to-end, NdT*).

Cette partie de la documentation se concentre sur les stratégies de test pour le premier cas de figure. Bien que des tests complets de bout en bout puissent être très utiles pour éviter des régressions dans des scénarios critiques d'utilisation, ce type de test ne se préoccupe guère des composants React en particulier, et sort donc du cadre de cette documentation.

Faire des choix {#tradeoffs}

Lorsqu'on détermine quels outils de test employer, il faut réaliser certains arbitrages :

- **Vitesse d'itération vs. environnement réaliste** : certains outils fournissent une boucle de retour extrêmement rapide entre le moment où vous changez votre code et l'obtention du résultat, mais ils ne simulent pas très précisément le comportement du navigateur. D'autres pourraient utiliser un véritable environnement navigateur, mais au prix d'une vitesse d'exécution moindre, sans parler des défis que leur utilisation peut poser sur un serveur d'intégration continue.
- **Jusqu'où isoler** : avec les composants, la frontière entre un test « unitaire » et un test « d'intégration » peut être floue. Si vous testez un formulaire, le test devrait-il également tester les boutons que ce formulaire contient ? Ou le composant bouton devrait-il avoir sa propre suite de tests ? Si on change la conception du bouton, l'échec soudain du test du formulaire est-il un dommage collatéral acceptable ?

D'une équipe ou d'un produit à l'autre, les réponses valables peuvent varier.

Outils recommandés {#tools}

Jest est un harnais de test JavaScript qui vous permet d'accéder au DOM via `jsdom`. Même si `jsdom` ne simule que partiellement le fonctionnement d'un navigateur, il est souvent suffisant pour tester vos composants React. Jest combine une excellente vitesse d'itération avec de puissantes fonctionnalités telles que l'isolation des `modules` et des `horloges`, afin que vous puissiez garder un contrôle fin sur la façon dont votre code s'exécute.

React Testing Library fournit un ensemble de fonctions utilitaires pour tester des composants React sans dépendre de leurs détails d'implémentation. Cette approche facilite le changement de conception interne et vous aiguille vers de meilleures pratiques en termes d'accessibilité. Même s'il ne fournit pas de moyen pour réaliser le rendu « superficiel » d'un composant (sans ses enfants), on peut y arriver avec un harnais tel que Jest et ses mécanismes `d'isolation`.

Pour en savoir plus {#learn-more}

Cette partie de la documentation comprend deux (autres) pages :

Voici quelques approches courantes pour tester des composants React.

Remarque

Cette page suppose que vous utilisez [Jest](#) comme harnais de test. Si vous utilisez un harnais différent, vous aurez peut-être besoin d'ajuster l'API, mais l'aspect général de la solution restera probablement inchangé. Pour en apprendre davantage sur la mise en place d'un environnement de test, consultez la page [Environnements de test](#).

Dans cette page, nous utiliserons principalement des fonctions composants. Ceci dit, ces stratégies de test sont découplées de ce genre de détail d'implémentation, et fonctionneront tout aussi bien pour des composants définis à base de classes.

- [Mise en place / nettoyage](#)
- [act\(\)](#)
- [Rendu](#)
- [Chargement de données](#)
- [Simuler des modules](#)
- [Événements](#)
- [Horloges](#)
- [Capture d'instantanés](#)
- [Moteurs de rendu multiples](#)
- [Pas trouvé votre bonheur ?](#)

Mise en place / nettoyage {#setup--teardown}

Pour chaque test, nous voulons habituellement réaliser le rendu d'un arbre React au sein d'un élément DOM attaché à [document](#). Ce dernier point est nécessaire pour que le composant puisse recevoir les événements du DOM. Et lorsque le test se termine, nous voulons « nettoyer » et démonter l'arbre présent dans [document](#).

Une façon courante de faire ça consiste à associer les blocks [beforeEach](#) et [afterEach](#) afin qu'il s'exécutent systématiquement autour de chaque test, ce qui permet d'en isoler les effets :

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

Vous utilisez peut-être une autre approche, mais gardez à l'esprit que vous voulez exécuter le nettoyage même si un test échoue. Sinon, vos tests peuvent commencer à « fuire », et un test pourrait altérer par inadvertance le comportement d'un autre, ce qui complexifie beaucoup le débogage.

act() {#act}

Lorsqu'on écrit des tests UI, des tâches comme le rendu lui-même, les événements utilisateurs ou encore le chargement de données peuvent être considérées comme autant « d'unités » d'interaction avec l'interface utilisateur. [react-dom/rest-utils](#) fournit une fonction utilitaire appelée `act()` qui s'assure que toutes les mises à jour relatives à ces « unités » ont bien été traitées et appliquées au DOM avant que nous ne commençons à exprimer nos assertions :

```
act(() => {
  // rendu des composants
});
// exécution des assertions
```

Ça nous aide à rapprocher nos tests du comportement que de véritables utilisateurs constateraient en utilisant notre application. La suite de ces exemples utilise `act()` pour bénéficier de ces garanties.

Vous trouverez peut-être que le recours manuel à `act()` est rapidement un tantinet verbeux. Pour vous épargner une bonne partie du code générique associé, vous pouvez opter pour une bibliothèque telle que [React Testing Library](#), dont les utilitaires sont basés sur `act()`.

Remarque

Le terme `act` vient de l'approche [Arrange-Act-Assert](#).

Rendu {#rendering}

Vous voudrez fréquemment vérifier que le rendu d'un composant est correct pour un jeu de props donné. Prenons un composant simple qui affiche un message basé sur une prop :

```
// hello.js

import React from "react";

export default function Hello(props) {
  if (props.name) {
    return <h1>Bonjour, {props.name} !</h1>;
  } else {
    return <span>Salut, étranger</span>;
  }
}
```

Nous pouvons écrire un test pour ce composant :

```
// hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("s'affiche avec ou sans nom", () => {
  act(() => {
    render(<Hello />, container);
  });
  expect(container.textContent).toBe("Salut, étranger");

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("Bonjour, Jenny !");

  act(() => {
    render(<Hello name="Margaret" />, container);
  });
  expect(container.textContent).toBe("Bonjour, Margaret !");
});
```

Chargement de données {#data-fetching}

Au lieu d'appeler de véritables API dans tous vos tests, vous pouvez simuler les requêtes et renvoyer des données factices. Simuler le chargement de données avec de « fausses » données évite de fragiliser les tests lors d'un back-end indisponible, et les accélère en prime. Remarquez que vous voudrez peut-être qu'une petite partie de vos tests utilisent un framework « **de bout en bout** » pour vérifier que l'appli dans son ensemble fonctionne bien.

```
// user.js

import React, { useState, useEffect } from "react";

export default function User(props) {
  const [user, setUser] = useState(null);

  async function fetchUserData(id) {
    const response = await fetch="/" + id);
    setUser(await response.json());
  }

  useEffect(() => {
    fetchUserData(props.id);
  }, [props.id]);

  if (!user) {
    return "Chargement...";
  }

  return (
    <details>
      <summary>{user.name}</summary>
      <strong>{user.age}</strong> ans
      <br />
      vit à {user.address}
    </details>
  );
}
}
```

Nous pouvons écrire les tests associés :

```
// user.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import User from "./user";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

```

});
```

```

it("affiche les données utilisateur", async () => {
  const fakeUser = {
    name: "Joni Baez",
    age: "32",
    address: "123, Charming Avenue"
  };

  jest.spyOn(global, "fetch").mockImplementation(() =>
    Promise.resolve({
      json: () => Promise.resolve(fakeUser)
    })
  );

  // Utilise la version asynchrone de `act` pour appliquer les promesses
  // accomplies
  await act(async () => {
    render(<User id="123" />, container);
  });

  expect(container.querySelector("summary").textContent).toBe(fakeUser.name);
  expect(container.querySelector("strong").textContent).toBe(fakeUser.age);
  expect(container.textContent).toContain(fakeUser.address);

  // retire la simulation pour assurer une bonne isolation des tests
  global.fetch.mockRestore();
});

```

Simuler des modules {#mocking-modules}

Certains modules ne fonctionneront peut-être pas bien dans un environnement de test, ou ne seront pas essentiels au test en lui-même. En simulant ces modules pour les remplacer par des versions factices, nous pouvons faciliter l'écriture des tests pour notre propre code.

Prenons un composant **Contact** qui intègre un composant tiers **GoogleMap** :

```

// map.js

import React from "react";

import { LoadScript, GoogleMap } from "react-google-maps";
export default function Map(props) {
  return (
    <LoadScript id="script-loader" googleMapsApiKey="VOTRE_CLÉ_API">
      <GoogleMap id="example-map" center={props.center} />
    </LoadScript>
  );
}

```

```
// contact.js

import React from "react";
import Map from "./map";

export default function Contact(props) {
  return (
    <div>
      <address>
        Contacter {props.name} par{" "}
        <a data-testid="email" href={"mailto:" + props.email}>
          e-mail
        </a>
        ou sur son <a data-testid="site" href={props.site}>
          site web
        </a>.
      </address>
      <Map center={props.center} />
    </div>
  );
}
```

Si nous ne voulons pas charger ce composant tiers lors de nos tests, nous pouvons simuler la dépendance elle-même pour renvoyer un composant factice, et exécuter nos tests :

```
// contact.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Contact from "./contact";
import MockedMap from "./map";

jest.mock("./map", () => {
  return function DummyMap(props) {
    return (
      <div data-testid="map">
        {props.center.lat}:{props.center.long}
      </div>
    );
  };
});

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});
```

```

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("devrait afficher les infos de contact", () => {
  const center = { lat: 0, long: 0 };
  act(() => {
    render(
      <Contact
        name="Joni Baez"
        email="test@example.com"
        site="http://test.com"
        center={center}
      />,
      container
    );
  });

  expect(
    container.querySelector("[data-testid='email']").getAttribute("href")
  ).toEqual("mailto:test@example.com");

  expect(
    container.querySelector('[data-testid="site"]').getAttribute("href")
  ).toEqual("http://test.com");

  expect(container.querySelector('[data-testid="map"]').textContent).toEqual(
    "0:0"
  );
});

```

Événements {#events}

Nous vous conseillons de déclencher de véritables événements DOM sur des éléments DOM, et de vérifier le résultat. Prenez ce composant **Toggle** :

```

// toggle.js

import React, { useState } from "react";

export default function Toggle(props) {
  const [state, setState] = useState(false);
  return (
    <button
      onClick={() => {
        setState(previousState => !previousState);
        props.onChange(!state);
      }}
    >{state ? "OFF" : "ON"}</button>
  );
}

```

```

        }
        data-testid="toggle"
      >
      {state === true ? "Éteindre" : "Allumer"}
    </button>
  );
}

```

Nous pourrions le tester comme ceci :

```

// toggle.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Toggle from "./toggle";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  // `container` *doit* être attaché à `document` pour que les événements
  // fonctionnent correctement.
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("change de valeur suite au clic", () => {
  const onChange = jest.fn();
  act(() => {
    render(<Toggle onChange={onChange} />, container);
  });

  // récupère l'élément bouton et déclenche quelques clics dessus
  const button = document.querySelector("[data-testid=toggle]");
  expect(button.innerHTML).toBe("Allumer");

  act(() => {
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onChange).toHaveBeenCalledTimes(1);
  expect(button.innerHTML).toBe("Éteindre");

  act(() => {

```

```

    for (let i = 0; i < 5; i++) {
      button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
    }
  });

expect(onChange).toHaveBeenCalledTimes(6);
expect(button.innerHTML).toBe("Allumer");
});

```

Les événements DOM disponibles et leurs propriétés sont décrits dans le [MDN](#). Remarquez que vous devez passer `{ bubbles: true }` pour chaque événement créé afin qu'ils puissent atteindre l'écouteur de React, car React délègue automatiquement les événements au niveau racine du document.

Remarque

React Testing Library propose [une façon plus concise](#) de déclencher des événements.

Horloges #timers

Votre code dépend peut-être de fonctions calées sur le temps telles que `setTimeout`, afin de planifier davantage de travail à l'avenir. Dans l'exemple ci-après, un panneau de choix multiples attend une sélection puis avance, avec un timeout si la sélection ne survient pas dans les 5 secondes :

```

// card.js

import React, { useEffect } from "react";

export default function Card(props) {
  useEffect(() => {
    const timeoutID = setTimeout(() => {
      props.onSelect(null);
    }, 5000);
    return () => {
      clearTimeout(timeoutID);
    };
  }, [props.onSelect]);

  return [1, 2, 3, 4].map(choice => (
    <button
      key={choice}
      data-testid={choice}
      onClick={() => props.onSelect(choice)}
    >
      {choice}
    </button>
  ));
}

```

Nous pouvons écrire les tests de ce composant en tirant parti de la [simulation d'horloges de Jest](#) et en testant les différents états possibles.

```
// card.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Card from "./card";

jest.useFakeTimers();

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("devrait sélectionner null à expiration", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  // avance dans le temps de 100ms
  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // puis avance de 5 secondes
  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).toHaveBeenCalledWith(null);
});

it("devrait nettoyer derrière lui lorsqu'il est retiré", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });
})
```

```

act(() => {
  jest.advanceTimersByTime(100);
});
expect(onSelect).not.toHaveBeenCalled();

// démonte l'appli
act(() => {
  render(null, container);
});

act(() => {
  jest.advanceTimersByTime(5000);
});
expect(onSelect).not.toHaveBeenCalled();
});

it("devrait accepter des sélections", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    container
      .querySelector("[data-testid='2']")
      .dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onSelect).toHaveBeenCalledWith(2);
});

```

Vous pouvez ne recourir à de fausses horloges que pour certains tests. Ci-avant nous les avons activées en appelant `jest.useFakeTimers()`. Le principal avantage réside dans le fait que votre test n'a pas besoin d'attendre effectivement cinq secondes pour s'exécuter, et vous n'avez pas eu besoin de complexifier le code de votre composant uniquement pour permettre ses tests.

Capture d'instantanés {#snapshot-testing}

Les frameworks tels que Jest vous permettent aussi de sauvegarder des « instantanés » de données grâce à `toMatchSnapshot` / `toMatchInlineSnapshot`. Avec elles, vous pouvez « sauver » la sortie de rendu d'un composant et vous assurer que toute modification qui lui sera apportée devra être explicitement confirmée en tant qu'évolution de l'instantané.

Dans l'exemple qui suit, nous affichons un composant et formatons le HTML obtenu grâce au module `pretty`, pour enfin le sauvegarder comme instantané en ligne :

```

// hello.test.js, à nouveau

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";

```

```

import { act } from "react-dom/test-utils";
import pretty from "pretty";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("devrait afficher une salutation", () => {
  act(() => {
    render(<Hello />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchSnapshot(); /* ...rempli automatiquement par Jest... */

  act(() => {
    render(<Hello name="Jenny" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchSnapshot(); /* ...rempli automatiquement par Jest... */

  act(() => {
    render(<Hello name="Margaret" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchSnapshot(); /* ...rempli automatiquement par Jest... */
});

```

Il est généralement préférable de recourir à des assertions spécifiques plutôt qu'à des instantanés. Ce type de tests inclut des détails d'implémentation qui les rendent particulièrement fragiles, entraînant une sorte d'anesthésie des équipes vis-à-vis des échecs de tests dus aux instantanés. Une [simulation ciblée de composants enfants](#) peut vous aider à réduire la taille de vos instantanés et à les garder lisibles pour vos revues de code.

Moteurs de rendu multiples {#multiple-renderers}

Dans de rares cas, vous pourrez vous retrouver à exécuter un test pour un composant qui, lui, recourt à plusieurs moteurs de rendu. Par exemple, peut-être exécutez-vous des tests à base d'instantanés sur un composant en utilisant `react-test-renderer`, alors que sous le capot le composant utilise `ReactDOM.render` pour obtenir le contenu d'un composant enfant. Dans un tel scénario vous pouvez enrober les mises à jour avec les appels aux fonctions `act()` des moteurs appropriés.

```
import { act as domAct } from "react-dom/test-utils";
import { act as testAct, create } from "react-test-renderer";
// ...
let root;
domAct(() => {
  testAct(() => {
    root = create(<App />);
  });
});
expect(root).toMatchSnapshot();
```



Strict Mode

`StrictMode` est un outil pour détecter les problèmes potentiels d'une application. Tout comme `Fragment`, `StrictMode` n'affiche rien. Il active des vérifications et avertissements supplémentaires pour ses descendants.

Remarque

Les vérifications du mode strict sont effectuées uniquement durant le développement. *Elles n'impactent pas la version utilisée en production.*

Vous pouvez activer le mode strict pour n'importe quelle partie du code de votre application. Par exemple : `embed:strict-mode/enabling-strict-mode.js`

Dans l'exemple ci-dessus, les vérifications du mode strict ne seront *pas* appliquées pour les composants `Header` et `Footer`. En revanche, les composants `ComponentOne` et `ComponentTwo`, ainsi que tous leurs descendants, seront vérifiés.

Actuellement, `StrictMode` est utilisé pour :

- Identifier les composants utilisant des méthodes de cycle de vie dépréciées
- Signaler l'utilisation dépréciée de l'API `ref` à base de chaîne de caractères
- Signaler l'utilisation dépréciée de `findDOMNode`
- Déetecter des effets de bord inattendus
- Déetecter l'API dépréciée de Contexte (ex. `childContextTypes`)

D'autres fonctionnalités seront ajoutées dans les futures versions de React.

Identifier les méthodes de cycle de vie dépréciées {#identifying-unsafe-lifecycles}

Comme l'explique [cet article de blog](#), certaines méthodes dépréciées de cycle de vie comportent des risques lorsqu'elles sont utilisées dans des applications React asynchrones. Qui plus est, si votre application utilise des bibliothèques tierces, il devient difficile de s'assurer que ces méthodes ne sont pas utilisées. Heureusement, le mode strict peut nous aider à les identifier !

Lorsque le mode strict est actif, React constitue une liste de tous les composants à base de classe utilisant les méthodes de cycle de vie à risque, et affiche dans la console un message d'avertissement avec des informations à propos de ces composants, comme ceci :

```
▶ Warning: Unsafe lifecycle methods were found within a strict-mode tree:  
  in div (created by ExampleApplication)  
  in ExampleApplication  
  
componentWillMount: Please update the following components to use componentDidMount instead: ThirdPartyComponent  
  
Learn more about this warning here:  
https://fb.me/react-strict-mode-warnings
```

En résolvant les problèmes identifiés par le mode strict *aujourd'hui*, vous pourrez plus facilement tirer parti du rendu concurrent qui arrivera dans les futures versions de React.

Signaler l'utilisation dépréciée de l'API `ref` à base de chaîne de caractères {#warning-about-legacy-string-ref-api-usage}

Auparavant, React fournissait deux manières de gérer les refs : l'API dépréciée à base de chaîne de caractères et l'API à base de fonction de rappel. Bien que la première ait été la plus pratique des deux, elle avait [plusieurs](#)

inconvénients. Du coup, nous recommandions officiellement d'utiliser plutôt la forme à base de fonction de rappel.

React 16.3 a ajouté une troisième option qui offre le confort de la première approche, mais sans ses inconvénients : [embed:16-3-release-blog-post/create-ref-example.js](#)

Dans la mesure où les refs à base d'objets sont largement utilisées comme substitut des refs à base de chaînes de caractères, le mode strict nous avertit désormais lors de l'utilisation de ces dernières.

Remarque

Les refs à base de fonction de rappel continueront d'être prises en charge en plus de la nouvelle API [createRef](#).

Vous n'avez pas besoin de remplacer les refs à base de fonctions de rappel dans vos composants. Elles sont un peu plus flexibles et resteront donc prises en charge, à titre de fonctionnalité avancée.

[Vous pouvez en apprendre davantage sur l'API createRef ici.](#)

Signaler l'utilisation dépréciée de [findDOMNode](#) {#warning-about-deprecated-finddomnode-usage}

React proposait autrefois [findDOMNode](#) pour rechercher dans l'arborescence le nœud DOM associé à une instance de classe. Normalement, vous n'avez pas besoin de ça car vous pouvez [attacher directement une ref à un nœud du DOM](#).

[findDOMNode](#) pouvait aussi être utilisée sur des composants à base de classe, mais ça cassait l'encapsulation en permettant à un parent d'exiger que certains enfants soient présents dans le rendu. Cette technique gênait les refactorisations car un composant ne pouvait plus changer ses détails d'implémentation en confiance, dans la mesure où des parents étaient susceptibles d'obtenir un accès direct à son nœud DOM. [findDOMNode](#) ne renvoie par ailleurs que le premier enfant, alors qu'avec les Fragments un composant peut renvoyer plusieurs nœuds DOM. [findDOMNode](#) est aussi une API temporalisée : sa valeur renvoyée n'est pas mise à jour automatiquement, de sorte que si un composant enfant se rafraîchit avec un autre nœud DOM, l'API ne vous en informe pas. En d'autres termes, [findDOMNode](#) ne fonctionnait que pour les composants renvoyant un unique nœud DOM qui ne changeait jamais.

Préférez une approche explicite en passant une ref à votre composant personnalisé et en la transférant au DOM grâce au [transfert de ref](#).

Vous pouvez également ajouter un nœud DOM d'enrobage dans votre composant et lui associer une ref directement.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.wrapper = React.createRef();
  }
  render() {
    return <div ref={this.wrapper}>{this.props.children}</div>;
  }
}
```

Remarque

En CSS, la propriété `display: contents` peut être utilisée si vous ne voulez pas que le nœud fasse partie de la mise en page.

DéTECTER les effets de bord inattendus {#detecting-unexpected-side-effects}

Conceptuellement, React fonctionne en deux étapes :

- La phase de **rendu** détermine les modifications qui doivent être retranscrites, par exemple dans le DOM. Lors de cette phase, React appelle `render` puis compare le résultat au rendu précédent.
- La phase de **commit** est celle de l'application des modifications. (Dans le cas de React DOM, c'est durant cette phase que React insère, modifie, et supprime des nœuds du DOM.) C'est également durant cette phase que React appelle des méthodes de cycle de vie comme `componentDidMount` et `componentDidUpdate`.

La phase de commit est le plus souvent très rapide, mais le rendu peut être lent. C'est pourquoi le mode concurrent à venir (qui n'est pas encore activé par défaut) découpe le travail de rendu en morceaux, suspendant et reprenant le travail pour éviter de bloquer le navigateur. Ça signifie que React peut invoquer les méthodes de cycle de vie de la phase de rendu plus d'une fois avant le commit, ou les invoquer sans phase de commit du tout (à cause d'une erreur ou d'une interruption de plus haute priorité).

Pour les composants à base de classes, les méthodes de cycle de vie de la phase de rendu sont les suivantes :

- `constructor`
- `componentWillMount` (or `UNSAFE_componentWillMount`)
- `componentWillReceiveProps` (or `UNSAFE_componentWillReceiveProps`)
- `componentWillUpdate` (or `UNSAFE_componentWillUpdate`)
- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- les fonctions de modifications passées à `setState` (son premier argument)

Vu que les méthodes ci-dessus peuvent être appelées plus d'une fois, il est impératif qu'elles ne contiennent pas d'effets de bord. Ignorer cette règle peut entraîner divers problèmes, dont des fuites de mémoire et un état applicatif invalide. Malheureusement, il peut être difficile de détecter ces problèmes car ils sont souvent **non-déterministes**.

Le mode strict ne détecte pas automatiquement ces effets de bord, mais il peut vous aider à les repérer en les rendant un peu plus déterministes. Il y parvient en invoquant volontairement deux fois les fonctions suivantes :

- Les méthodes `constructor`, `render` et `shouldComponentUpdate` des composants à base de classe
- La méthode statique `getDerivedStateFromProps` des composants à base de classe
- Le corps des fonctions composants
- Les fonctions de mise à jour d'état (le premier argument passé à `setState`)
- Les fonctions passées à `useState`, `useMemo` ou `useReducer`

Remarque

Cette fonctionnalité s'applique uniquement en mode développement. *Les méthodes de cycle de vie ne seront pas invoquées deux fois en mode production.*

Par exemple, examinez le code suivant : [embed:strict-mode/side-effects-in-constructor.js](#)

Au premier abord, ce code ne semble pas problématique. Cependant, si

`SharedApplicationState.recordEvent` n'est pas `idempotent`, alors l'instanciation multiple de ce composant pourrait corrompre l'état applicatif. Ce genre de bug subtil peut ne pas se manifester durant le développement, ou s'avérer tellement erratique qu'il est négligé.

En invoquant volontairement deux fois les méthodes comme le constructeur d'un composant, le mode strict facilite la détection de ces schémas.

Déetecter l'API dépréciée de Contexte {#detecting-legacy-context-api}

L'API dépréciée de Contexte est source d'erreur, et sera retirée dans une future version majeure de React. Elle fonctionne toujours dans les versions 16.x mais le mode strict affichera ce message d'avertissement :

```
✖ ►Warning: Legacy context API has been detected within a strict-mode tree:  
  in div (at App.js:32)  
  in App (at index.js:7)  
  
Please update the following components: LegacyContextConsumer, LegacyContextProvider  
  
Learn more about this warning here:  
https://fb.me/react-strict-mode-warnings
```

Lisez la [documentation de la nouvelle API de Contexte](#) pour faciliter la migration vers cette nouvelle version.



Optimisation

Le **Profiler** mesure à quelle fréquence une application React réalise son rendu, et détermine le « coût » de ces rendus. L'objectif est de vous aider à identifier les parties d'une application qui sont lentes et pourraient bénéficier [d'optimisations telles que la mémoïsation](#).

Remarque

Le profilage pénalise légèrement les performances effectives, il est donc **désactivé dans le build de production**.

Pour activer le profilage en production, React fournit un build de production spécifique avec le profilage activé. Vous pouvez apprendre comment l'utiliser sur fb.me/react-profiling.

Utilisation {#usage}

Un **Profiler** peut être ajouté n'importe où dans l'arborescence React pour mesurer le coût de rendu de la partie de l'arbre qu'il entoure. Il nécessite deux props : un **id** (chaîne de caractères) et une fonction de rappel **onRender** que React appellera dès qu'un composant au sein de l'arborescence enrobée « finalise » (*"commits"*, *NdT*) une mise à jour.

Par exemple, pour profiler le composant **Navigation** et ses descendants, on ferait ceci :

```
render(  
  <App>  
    <Profiler id="Navigation" onRender={callback}>  
      <Navigation {...props} />  
    </Profiler>  
    <Main {...props} />  
  </App>  
)
```

Vous pouvez utiliser plusieurs composants **Profiler** pour mesurer différentes parties d'une même application :

```
render(  
  <App>  
    <Profiler id="Navigation" onRender={callback}>  
      <Navigation {...props} />  
    </Profiler>  
    <Profiler id="Main" onRender={callback}>  
      <Main {...props} />  
    </Profiler>  
  </App>  
)
```

Les composants **Profiler** peuvent par ailleurs être imbriqués pour mesurer différents périmètres dans une même partie de l'arborescence :

```

render(
  <App>
    <Profiler id="Panel" onRender={callback}>
      <Panel {...props}>
        <Profiler id="Content" onRender={callback}>
          <Content {...props} />
        </Profiler>
        <Profiler id="PreviewPane" onRender={callback}>
          <PreviewPane {...props} />
        </Profiler>
      </Panel>
    </Profiler>
  </App>
);

```

Remarque

Même si **Profiler** est un composant léger, il ne devrait être utilisé que lorsqu'il est nécessaire, car chaque utilisation entraîne une pénalité en termes de processeur et de mémoire pour l'application.

Fonction de rappel **onRender** {#onrender-callback}

Le **Profiler** nécessite une fonction **onRender** dans ses props. React appelle cette fonction dès qu'un composant dans l'arbre profilé « finalise » (*"commits"*, NdT) une mise à jour. La fonction reçoit des paramètres dérivant ce qui vient de faire l'objet d'un rendu, et le temps que ça a pris.

```

function onRenderCallback(
  id, // la prop "id" du Profiler dont l'arborescence vient d'être mise à jour
  phase, // soit "mount" (si on est au montage) soit "update" (pour une mise à jour)
  actualDuration, // temps passé à faire le rendu de la mise à jour finalisée
  baseDuration, // temps estimé du rendu pour l'ensemble du sous-arbre sans
  memoisation
  startTime, // horodatage du début de rendu de cette mise à jour par React
  commitTime, // horodatage de la finalisation de cette mise à jour par React
  interactions // Un Set des interactions qui constituent cette mise à jour
) {
  // Agrège ou logue les mesures de rendu...
}

```

Examinons chaque argument d'un peu plus près...

- **id: string** - la prop **id** du **Profiler** dont l'arbre vient d'être finalisé. On s'en sert pour identifier la partie de l'arbre qui vient d'être finalisée si on utilise plusieurs profileurs.
- **phase: "mount" | "update"** - nous permet de savoir si l'arbre vient d'être monté (premier rendu) ou si c'est un nouveau rendu suite à une modification des props, de l'état local ou de hooks.
- **actualDuration: number** - temps passé à faire le rendu du **Profiler** et de ses descendants lors de la mise à jour courante. Nous permet de voir dans quelle mesure le sous-arbre tire parti de la

mémoïsation (ex. `React.memo`, `useMemo`, `shouldComponentUpdate`). Dans l'idéal cette valeur devrait décroître significativement après le montage initial, car de nombreux descendants ne devraient nécessiter un nouveau rendu que si leurs propres props changent.

- **baseDuration: number** - durée du `render` le plus récent qui revisitait l'ensemble des composants dans l'arbre enrobé par le `Profiler`. Cette valeur nous permet d'estimer un scénario du pire (ex. le montage initial ou un arbre sans mémoïsation).
- **startTime: number** - horodatage du début du rendu de la mise à jour courante par React.
- **commitTime: number** - horodatage de la finalisation de la mise à jour courante par React. Cette valeur est partagée entre tous les profileurs d'une même finalisation, ce qui permet de les regrouper si on le souhaite.
- **interactions: Set** - un `Set` des « `interactions` » qui ont été pistées lors de la planification de la mise à jour (ex. lors des appels à `render` ou `setState`).

Remarque

Vous pouvez utiliser les interactions pour identifier la cause d'une mise à jour, même si l'API pour le pistage des interactions est encore expérimentale.

Vous pouvez en apprendre davantage sur fb.me/react-interaction-tracing.



Performance

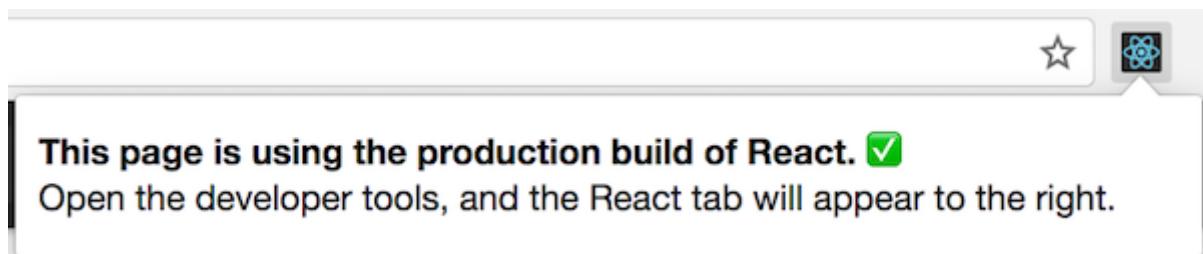
En interne, React fait appel à différentes techniques intelligentes pour minimiser le nombre d'opérations coûteuses sur le DOM nécessaires à la mise à jour de l'interface utilisateur (UI). Pour de nombreuses applications, utiliser React offrira une UI rapide sans avoir à fournir beaucoup de travail pour optimiser les performances. Néanmoins, il existe plusieurs façons d'accélérer votre application React.

Utiliser la version de production {#use-the-production-build}

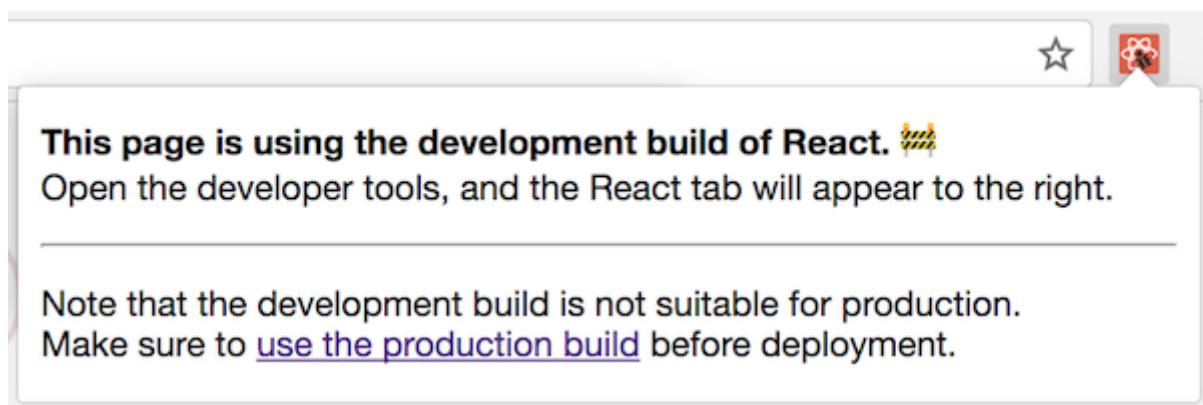
Si vous mesurez ou rencontrez des problèmes de performances dans vos applications React, assurez-vous que vous testez bien la version minifiée de production.

Par défaut, React intègre de nombreux avertissements pratiques. Ces avertissements sont très utiles lors du développement. Toutefois, ils rendent React plus gros et plus lent, vous devez donc vous assurer que vous utilisez bien une version de production lorsque vous déployez l'application.

Si vous n'êtes pas sûr·e que votre processus de construction est correctement configuré, vous pouvez le vérifier en installant [l'extension React Developer Tools pour Chrome](#). Si vous visitez un site avec React en production, l'icône aura un fond sombre :



Si vous visitez un site avec React dans sa version de développement, l'icône aura un fond rouge :



L'idée, c'est que vous utilisez le mode développement lorsque vous travaillez sur votre application, et le mode production lorsque vous la déployez pour vos utilisateurs.

Vous trouverez ci-dessous les instructions pour procéder à la construction de votre application pour la production.

Create React App {#create-react-app}

Si votre projet est construit avec [Create React App](#), exécutez :

```
npm run build
```

Cela générera la version de production de votre application dans le répertoire `build/` de votre projet.

Rappelez-vous que cela n'est nécessaire qu'avant le déploiement en production. Lors du développement, utilisez `npm start`.

Versions de production officielles {#single-file-builds}

Nous mettons à disposition des versions de React et de React DOM prêtes pour la production sous la forme de fichiers uniques :

```
<script src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js">
</script>
```

Rappelez-vous que seuls les fichiers React finissant par `.production.min.js` sont adaptés à la production.

Brunch {#brunch}

Pour obtenir la version de production la plus efficace avec Brunch, installez l'extension `terser-brunch` :

```
# Si vous utilisez npm :
npm install --save-dev terser-brunch

# Si vous utilisez Yarn :
yarn add --dev terser-brunch
```

Ensuite, pour créer la version de production, ajoutez l'option `-p` à la commande `build` :

```
brunch build -p
```

N'oubliez pas que cela n'est nécessaire que pour générer votre version de production. Vous ne devez pas utiliser l'argument `-p` ni l'extension lors des phases de développement, car ça masquerait les avertissements utiles de React et ralentirait notablement la construction de l'application.

Browserify {#browserify}

Pour obtenir la version de production la plus efficace avec Browserify, installez quelques extensions :

```
# Si vous utilisez npm :
npm install --save-dev envify terser uglifyify

# Si vous utilisez Yarn :
yarn add --dev envify terser uglifyify
```

Pour créer la version de production, assurez-vous d'ajouter ces transformations (**l'ordre a son importance**) :

- La transformation `envify` s'assure que l'environnement est correctement défini. Définissez-la globalement (`-g`).
- La transformation `uglifyify` supprime les imports de développement. Définissez-la également au niveau global (`-g`).
- Enfin, le *bundle* qui en résulte est transmis à `terser` pour être obfusqué ([les raisons sont détaillées ici](#)).

Par exemple :

```
browserify ./index.js \
  -g [ envify --NODE_ENV production ] \
  -g uglifyify \
  | terser --compress --mangle > ./bundle.js
```

Rappelez-vous que vous n'avez à faire cela que pour la version de production. Vous ne devez pas appliquer ces extensions en développement, car cela masquerait des avertissements utiles de React et ralentirait la construction.

Rollup {#rollup}

Pour obtenir la version de production la plus efficace avec Rollup, installez quelques extensions :

```
# Si vous utilisez npm :
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser

# Si vous utilisez Yarn :
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser
```

Pour créer la version de production, assurez-vous d'ajouter ces transformations (**l'ordre a son importance**) :

- L'extension `replace` s'assure que l'environnement est correctement configuré.
- L'extension `commonjs` prend en charge CommonJS au sein de Rollup.
- L'extension `terser` réalise la compression et obfusque le bundle final.

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-terser')(),
  // ...
]
```

Pour une configuration complète, [vous pouvez consulter ce gist](#).

Rappelez-vous que vous n'avez à faire cela que pour la version de production. Vous ne devez pas utiliser les extensions `terser` ou `replace` avec une valeur '`production`' en développement, car cela masquerait des avertissements utiles de React et ralentirait la construction.

webpack {#webpack}

Remarque

Si vous utilisez Create React App, merci de suivre [les instructions ci-dessus](#).

Cette section n'est utile que si vous configurez webpack vous-même.

Webpack v4+ minifera automatiquement votre code en mode production.

```
const TerserPlugin = require('terser-webpack-plugin');

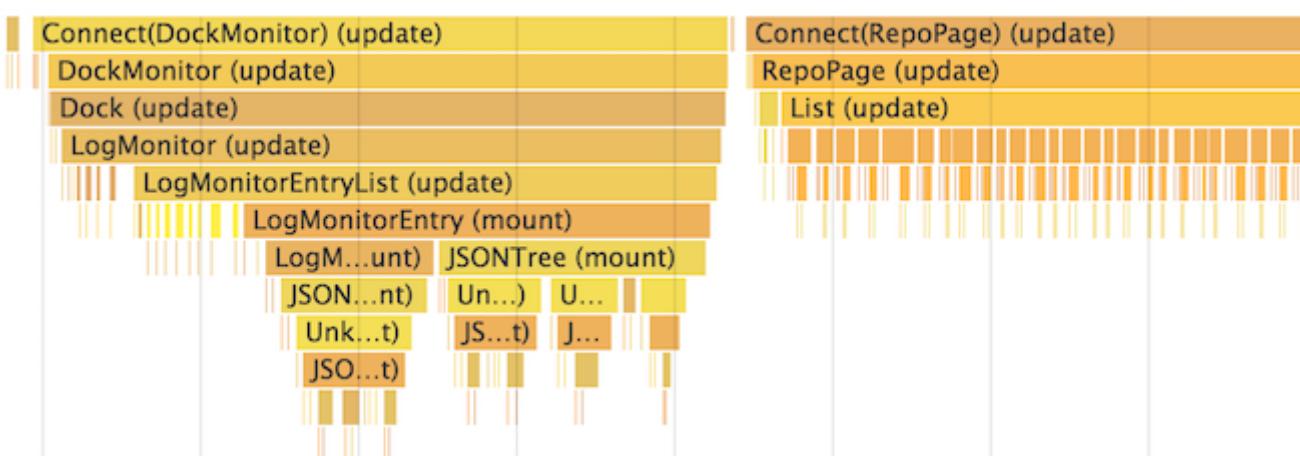
module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

Vous pouvez en apprendre davantage sur le sujet en consultant la [documentation webpack](#).

Rappelez-vous que vous n'avez à faire cela que pour la version de production. Vous ne devez pas utiliser `TerserPlugin` en développement, car cela masquerait des avertissements utiles de React et ralentirait la construction.

Profilage des composants avec l'onglet Performance de Chrome {#profiling-components-with-the-chrome-performance-tab}

En mode de **développement**, vous pouvez voir comment les composants sont montés, mis à jour et démontés en utilisant les outils de performances dans les navigateurs qui les prennent en charge. Par exemple :



Pour faire ça avec Chrome :

1. **Désactivez temporairement toutes les extensions de Chrome, en particulier React DevTools.** Elles peuvent considérablement impacter les résultats !
2. Assurez-vous d'utiliser l'application en mode de développement.
3. Ouvrez l'onglet **Performances** dans les DevTools de Chrome et appuyez sur **Record**.
4. Effectuez les opérations que vous voulez analyser. N'enregistrez pas plus de 20 secondes, car Chrome pourrait se bloquer.
5. Arrêtez l'enregistrement.
6. Les événements React seront regroupés sous l'étiquette **User Timing**.

Pour une présentation plus détaillée, consultez [cet article de Ben Schwarz](#).

Veuillez noter que **ces résultats sont relatifs et que les composants seront rendus plus rapidement en production**. Néanmoins, ça devrait vous aider à comprendre quand des éléments d'interface sont mis à jour par erreur, ainsi que la profondeur et la fréquence des mises à jour de l'UI.

Pour le moment, Chrome, Edge et IE sont les seuls navigateurs prenant en charge cette fonctionnalité, mais comme nous utilisons [l'API standard User Timing](#), nous nous attendons à ce que d'autres navigateurs la prennent en charge.

Profilage des composants avec le DevTools Profiler {#profiling-components-with-the-devtools-profiler}

`react-dom` 16.5+ et `react-native` 0.57+ offrent des capacités de profilage avancées en mode de développement avec le Profiler de l'extension React DevTools. Vous trouverez un aperçu du profileur sur le billet de blog « [Découvrez le profileur React](#) ». Une présentation vidéo du profileur est également [disponible sur YouTube](#).

Si vous n'avez pas encore installé l'extension React DevTools, vous pourrez la trouver ici :

- [L'extension pour le navigateur Chrome](#).
- [L'extension pour le navigateur Firefox](#).
- [Le module pour Node.js](#).

Remarque

Un module de profilage pour la production de `react-dom` existe aussi dans [react-dom/profiling](#). Pour en savoir plus sur l'utilisation de ce module, rendez-vous à l'adresse [fb.me/react-profiling](#).

Virtualiser les listes longues {#virtualize-long-lists}

Si votre application génère d'importantes listes de données (des centaines ou des milliers de lignes), nous vous conseillons d'utiliser la technique de « fenêtrage » (*windowing, NdT*). Cette technique consiste à n'afficher à tout instant qu'un petit sous-ensemble des lignes, ce qui permet de diminuer considérablement le temps nécessaire au rendu des composants ainsi que le nombre de noeuds DOM créés.

[react-window](#) et [react-virtualized](#) sont des bibliothèques populaires de gestion du fenêtrage. Elles fournissent différents composants réutilisables pour afficher des listes, des grilles et des données tabulaires. Vous pouvez également créer votre propre composant, comme [l'a fait Twitter](#), si vous voulez quelque chose de plus adapté à vos cas d'usage spécifiques.

Éviter la réconciliation {#avoid-reconciliation}

React construit et maintient une représentation interne de l'UI produite, représentation qui inclut les éléments React renvoyés par vos composants. Elle permet à React d'éviter la création de nœuds DOM superflus et l'accès excessif aux nœuds existants, dans la mesure où ces opérations sont plus lentes que sur des objets JavaScript. On y fait parfois référence en parlant de « DOM virtuel », mais ça fonctionne de la même façon avec React Native.

Quand les props ou l'état local d'un composant changent, React décide si une mise à jour du DOM est nécessaire en comparant l'élément renvoyé avec l'élément du rendu précédent. Quand ils ne sont pas égaux, React met à jour le DOM.

Même si React ne met à jour que les nœuds DOM modifiés, refaire un rendu prend un certain temps. Dans la plupart des cas ce n'est pas un problème, mais si le ralentissement est perceptible, vous pouvez accélérer le processus en surchargeant la méthode `shouldComponentUpdate` du cycle de vie, qui est déclenchée avant le démarrage du processus de rafraîchissement. L'implémentation par défaut de cette méthode renvoie `true`, laissant ainsi React faire la mise à jour :

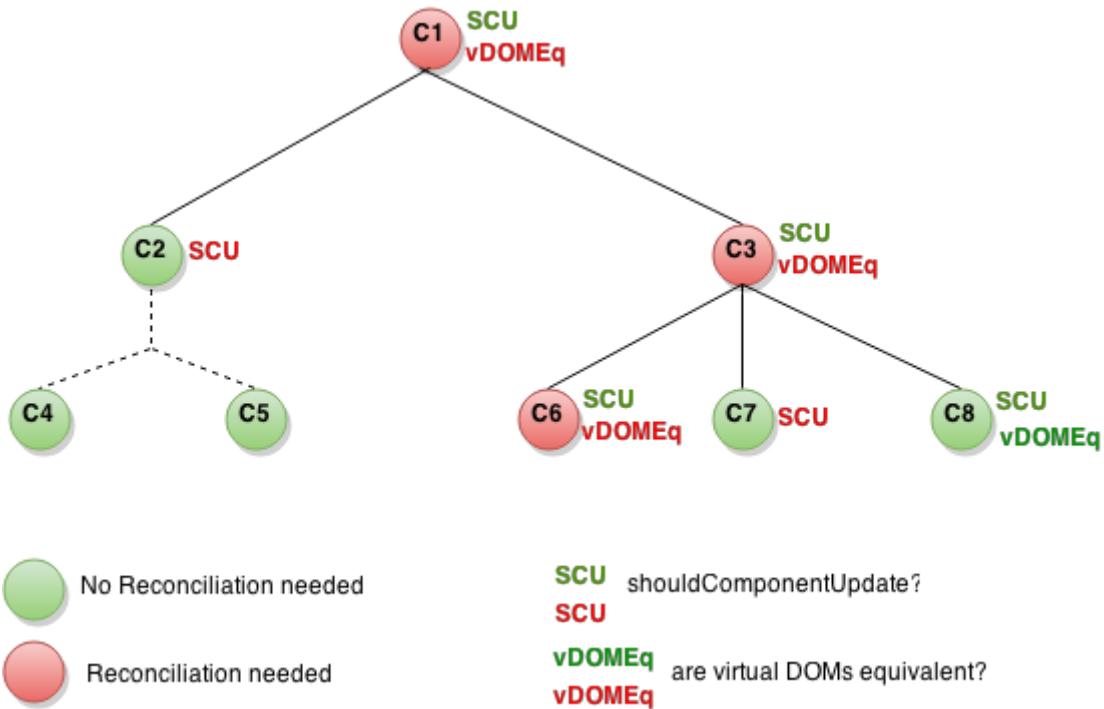
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

Si vous savez que dans certaines situations votre composant n'a pas besoin d'être mis à jour, vous pouvez plutôt renvoyer `false` depuis `shouldComponentUpdate` afin de sauter le rendu, et donc l'appel à la méthode `render()` sur ce composant et ses enfants.

Le plus souvent, plutôt que d'écrire manuellement `shouldComponentUpdate()`, vous pouvez plutôt choisir d'étendre `React.PureComponent`. Ça revient à implémenter `shouldComponentUpdate()` avec une comparaison superficielle des propriétés et état actuels et précédents.

shouldComponentUpdate en action {#shouldcomponentupdate-in-action}

Voici un sous-arbre de composants. Pour chacun, `SCU` indique ce que `shouldComponentUpdate` renvoie, et `vDOMEq` indique si les éléments renvoyés étaient équivalents. Enfin, la couleur du cercle indique si le composant doit être réconcilié ou non.



Puisque `shouldComponentUpdate` a renvoyé `false` pour le sous-arbre d'origine C2, React n'a pas tenté de faire le rendu de C2, et n'a pas invoqué non plus `shouldComponentUpdate` sur C4 et C5.

Pour C1 et C3, `shouldComponentUpdate` a renvoyé `true`, React a donc dû descendre dans les feuilles de l'arbre et les vérifier. Pour C6, `shouldComponentUpdate` a renvoyé `true`, et puisque les éléments renvoyés n'étaient pas équivalents, React a dû mettre à jour le DOM.

Le dernier cas intéressant concerne C8. React a dû faire le rendu de ce composant, mais puisque les éléments React renvoyés étaient équivalents à ceux du rendu précédent, il n'était pas nécessaire de mettre à jour le DOM.

Remarquez que React n'a dû modifier le DOM que pour C6, ce qui était inévitable. Pour C8, il s'en est dispensé suite à la comparaison des éléments React renvoyés, et pour le sous-arbre de C2 ainsi que pour C7, il n'a même pas eu à comparer les éléments car nous avons abandonné au niveau de `shouldComponentUpdate`, et `render` n'a pas été appelée.

Exemples {#examples}

Si la seule façon de changer pour votre composant provient d'une modification de `props.color` ou `state.count`, alors vous devez vérifier ces valeurs dans `shouldComponentUpdate` :

```

class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
  }
}

```

```

    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
}

render() {
  return (
    <button
      color={this.props.color}
      onClick={() => this.setState(state => ({count: state.count + 1}))}>
      Compteur : {this.state.count}
    </button>
  );
}
}

```

Dans ce code, `shouldComponentUpdate` vérifie simplement si `props.color` ou `state.count` ont changé. Dans le cas contraire, le composant n'est pas mis à jour. Si votre composant devient plus complexe, vous pourriez utiliser une approche similaire en procédant à une « comparaison superficielle » (*shallow comparison, NdT*) de tous les champs de `props` et `state` afin de déterminer si le composant doit être mis à jour. Ce modèle est suffisamment fréquent pour que React nous y aide : on hérite simplement de `React.PureComponent`. Ce code est donc une façon plus simple de réaliser la même chose :

```

class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Compteur : {this.state.count}
      </button>
    );
  }
}

```

La plupart du temps, vous pouvez utiliser `React.PureComponent` au lieu de redéfinir `shouldComponentUpdate` vous-même. Il ne réalise qu'une comparaison superficielle, vous ne pouvez donc pas l'utiliser si les propriétés ou l'état sont modifiés d'une façon qui échapperait à ce type de comparaison.

Ça peut devenir un problème avec des structures de données plus complexes. Supposons, par exemple, que vous voulez qu'un composant `ListofWords` affiche une liste de mots séparés par des virgules, avec un composant parent `WordAdder` qui vous permet d'ajouter un mot à la liste d'un simple clic. Ce code *ne fonctionnera pas* correctement :

```

class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // Cette section comporte une mauvaise pratique qui entraînera un bug.
    const words = this.state.words;
    words.push('marklar');
    this.setState({words: words});
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick} />
        <ListOfWords words={this.state.words} />
      </div>
    );
  }
}

```

Le problème est que `PureComponent` va faire une comparaison référentielle entre l'ancienne et la nouvelle valeur de `this.props.words`. Dans la mesure où ce code modifie directement le tableau `words` dans la méthode `handleClick` de `WordAdder`, l'ancienne et la nouvelle valeurs de `this.props.words` sont considérées comme équivalentes (même objet en mémoire), bien que les mots dans le tableau aient été modifiés. Le composant `ListOfWords` ne sera pas mis à jour, même s'il devrait afficher de nouveaux mots.

La puissance des données immuables {#the-power-of-not-mutating-data}

La façon la plus simple d'éviter ce problème consiste à éviter de modifier directement les valeurs que vous utilisez dans les props ou l'état local. Par exemple, la méthode `handleClick` au-dessus pourrait être réécrite en utilisant `concat` comme suit :

```

handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  })
}

```

```
});  
}
```

ES6 offre la [syntaxe de décomposition](#) (*spread operator, NdT*) pour les tableaux, ce qui facilite ce type d'opération. Si vous utilisez Create React App, cette syntaxe est disponible par défaut.

```
handleClick() {  
  this.setState(state => ({  
    words: [...state.words, 'marklar'],  
  }));  
};
```

D'une manière similaire, vous pouvez réécrire du code qui modifie des objets en évitant la mutation. Par exemple, supposons que nous ayons un objet nommé `colormap` et que nous voulions écrire une fonction qui change la valeur de `colormap.right` en `'blue'`. Nous pourrions l'écrire ainsi :

```
function updateColorMap(colormap) {  
  colormap.right = 'blue';  
}
```

Pour écrire cela en évitant de modifier l'objet original, nous pouvons utiliser la méthode [Object.assign](#) :

```
function updateColorMap(colormap) {  
  return Object.assign({}, colormap, {right: 'blue'});  
}
```

`updateColorMap` renvoie désormais un nouvel objet, plutôt que de modifier l'ancien. [Object.assign](#) fait partie d'ES6 et nécessite un polyfill.

[La syntaxe de décomposition des objets](#) facilite la mise à jour d'objets sans pour autant les modifier :

```
function updateColorMap(colormap) {  
  return {...colormap, right: 'blue'};  
}
```

Cette fonctionnalité est apparue dans JavaScript avec ES2018.

Si vous utilisez Create React App, la méthode `Object.assign` et la syntaxe de décomposition d'objets sont toutes deux disponibles par défaut.

Lorsque vous faites face à des objets profondément imbriqués, les mettre à jour de manière immuable peut se révéler compliqué. Si vous faites face à ce problème, tournez-vous vers [Immer](#) ou [immutability-helper](#). Ces librairies vous permettent d'écrire du code très lisible sans perdre les bénéfices de l'immuabilité.



Portal

Les portails fournissent une excellente solution pour afficher des composants enfants dans un nœud DOM qui existe en dehors de la hiérarchie DOM du composant parent.

```
ReactDOM.createPortal(child, container)
```

Le premier argument (`child`) peut être n'importe quel [enfant affichable par React](#), comme un élément, une chaîne de caractères ou un fragment. Le second argument (`container`) est un élément du DOM.

Utilisation {#usage}

D'habitude, lorsque vous renvoyez un élément depuis le rendu d'un composant, cet élément est monté dans le DOM en tant qu'enfant du plus proche parent :

```
render() {
  // React monte une nouvelle div et affiche les enfants à l'intérieur de celle-ci
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

Cependant il est parfois utile d'insérer un enfant à un autre emplacement du DOM :

```
render() {
  // React *ne crée pas* une nouvelle div, mais affiche les enfants dans
  `domNode`.
  // `domNode` peut être n'importe quel élément valide du DOM, peu importe sa
  position.
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

Un cas typique d'utilisation des portails survient lorsqu'un composant parent possède un style `overflow: hidden` ou `z-index` et que l'enfant a besoin de « sortir de son conteneur » visuellement. C'est par exemple le cas des boîtes de dialogues, des pop-ups ou encore des infobulles.

Remarque

Lorsque vous travaillez avec les portails, gardez en tête que la [gestion du focus du clavier](#) devient très importante.

Pour les fenêtres modales, assurez-vous que tout le monde puisse interagir avec celles-ci en suivant les règles [WAI-ARIA Modal Authoring Practices du W3C](#) (en anglais).

Essayer dans CodePen

La propagation des événements dans les portails {#event-bubbling-through-portals}

Même si un portail peut être placé n'importe où dans l'arborescence DOM, il se comporte comme un enfant React normal à tous les autres points de vue. Les fonctionnalités comme le contexte se comportent exactement de la même façon, indépendamment du fait que l'enfant soit un portail, car le portail existe toujours dans *l'arborescence React*, indépendamment de sa position dans *l'arborescence DOM*.

Ça concerne aussi la propagation montante des événements. Un événement déclenché à l'intérieur d'un portail sera propagé aux ancêtres dans *l'arborescence React*, même si ces éléments ne sont pas ses ancêtres dans *l'arborescence DOM*. Prenons par exemple le code HTML suivant :

```
<html>
  <body>
    <div id="app-root"></div>
    <div id="modal-root"></div>
  </body>
</html>
```

Un composant **Parent** dans `#app-root` pourrait attraper un événement montant non-intercepté provenant du nœud frère `#modal-root`.

```
// Ces deux conteneurs sont frères dans le DOM
const appRoot = document.getElementById('app-root');
const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    // L'élément portail est inséré dans l'arborescence DOM une fois
    // que les enfants du Modal sont montés, ce qui signifie que
    // les enfants seront montés sur un nœud DOM détaché.
    // Si un composant enfant nécessite d'être attaché au DOM
    // dès le montage, par exemple pour mesurer un nœud DOM ou
    // utiliser 'autoFocus' dans un nœud descendant, ajoutez un état
    // à la modale et affichez uniquement les enfants une fois la
    // modale insérée dans le DOM.
    modalRoot.appendChild(this.el);
  }
}
```

```

componentWillUnmount() {
  modalRoot.removeChild(this.el);
}

render() {
  return ReactDOM.createPortal(
    this.props.children,
    this.el
  );
}
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {clicks: 0};
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // La fonction se déclenchera lorsque le bouton dans l'enfant sera cliqué,
    // permettant la mise à jour de l'état du parent, même si le bouton
    // n'en est pas un descendant direct dans le DOM.
    this.setState(state => ({
      clicks: state.clicks + 1
    }));
  }

  render() {
    return (
      <div onClick={this.handleClick}>
        <p>Nombre de clics : {this.state.clicks}</p>
        <p>
          Ouvrez les outils de développement de votre navigateur
          pour observer que ce bouton n'est pas un enfant de la div
          qui écoute les événements de clic.
        </p>
        <Modal>
          <Child />
        </Modal>
      </div>
    );
  }
}

function Child() {
  // Lors de clics sur ce bouton, l'événement sera propagé au parent
  // car il n'y a pas d'attribut 'onClick' défini ici.
  return (
    <div className="modal">
      <button>Cliquez ici</button>
    </div>
  );
}

```

```
ReactDOM.render(<Parent />, appRoot);
```

[Essayer dans CodePen](#)

Attraper un événement en cours de propagation depuis un portail dans un composant parent autorise le développement d'abstractions plus flexibles qui ne sont pas forcément liées aux portails. Par exemple, si vous affichez un composant `<Modal />`, le parent peut capturer ses événements, que le parent soit implémenté à base de portails ou non.



Hooks

Les *Hooks* sont arrivés avec React 16.8. Ils vous permettent de bénéficier d'un état local et d'autres fonctionnalités de React sans avoir à écrire une classe.

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

function Example() {
  // Déclare une nouvelle variable d'état, qu'on va appeler « count »
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Vous avez cliqué {count} fois</p>
      <button onClick={() => setCount(count + 1)}>
        Cliquez ici
      </button>
    </div>
  );
}
```

Cette nouvelle fonction `useState` est le premier « Hook » que nous allons explorer, mais cet exemple est juste un petit aperçu. Ne vous inquiétez pas si vous n'y comprenez rien pour le moment, ça va venir vite !

Vous pouvez commencer à apprendre les Hooks à la page suivante. Dans celle-ci, nous vous expliquons pourquoi nous avons ajouté les Hooks à React et comment ils vous aideront à écrire des applications géniales.

Remarque

Les Hooks sont apparus dans React 16.8.0. Lors de la mise à jour de React, n'oubliez pas de mettre à jour tous les modules, dont React DOM.

React Native prend en charge les Hooks depuis [sa version 0.59](#).

Pas de rupture de compatibilité {#no-breaking-changes}

Avant de continuer, remarquez bien que les hooks sont :

Complètement optionnels. Vous pouvez essayer les Hooks dans quelques composants sans réécrire le code existant. Mais vous n'avez pas à apprendre et utiliser les Hooks dès maintenant si vous ne le souhaitez pas.

100% rétro-compatibles. Les Hooks préservent la compatibilité ascendante.

Disponibles maintenant. Les Hooks sont disponibles depuis la version 16.8.0.

Les classes en React ne sont pas menacées. Vous pouvez en apprendre davantage sur la stratégie d'adoption progressive des Hooks [en bas de cette page](#).

Les Hooks n invalident pas vos connaissances des concepts de React. Les Hooks fournissent plutôt une API plus directe pour les concepts React que vous connaissez déjà : props, état local, contexte, refs et cycle de vie. Comme nous le verrons plus tard, les Hooks offrent également un nouveau moyen puissant de les combiner.

Si vous voulez juste commencer à apprendre les Hooks, n'hésitez pas à aller directement à la page suivante ! Vous pouvez également continuer à lire cette page pour en apprendre davantage sur les raisons pour lesquelles nous ajoutons les Hooks et sur la façon dont nous allons commencer à les utiliser sans réécrire nos applications.

Raisons {#motivation}

Les Hooks résolvent une grande variété de problèmes apparemment sans rapports en React, que nous avons rencontrés pendant cinq ans d'écriture et de maintenance de dizaines de milliers de composants. Que vous appreniez React, l'utilisiez quotidiennement ou préfériez une bibliothèque différente avec un modèle de composants similaire, vous pourriez reconnaître certains de ces problèmes.

Il est difficile de réutiliser la logique à état entre les composants {#its-hard-to-reuse-stateful-logic-between-components}

React n'offre aucun moyen « d'attacher » un comportement réutilisable à un composant (par exemple, le connecter à un état applicatif). Si vous utilisez déjà React depuis quelques temps, vous avez peut-être déjà rencontré les [props de rendu](#) et les [composants d'ordre supérieur](#) qui tentent d'apporter une solution. Mais ces approches exigent la restructuration de vos composants lorsque vous les utilisez, ce qui rend le code plus lourd et difficile à maintenir.

Si vous examinez une application React typique dans les React DevTools, vous verrez un véritable « enfer d'enrobage » de composants perdus dans des couches de fournisseurs et consommateurs de contexte, composants d'ordre supérieur, props de rendu et autres abstractions. On pourrait [les filtrer dans les DevTools](#), mais ils sont symptomatiques d'un problème plus profond : React a besoin d'une meilleure primitive pour la réutilisation des logiques à état.

Avec les Hooks, vous pouvez extraire la logique à état d'un composant pour la réutiliser et la tester de façon indépendante. **Les Hooks vous permettent de réutiliser de la logique à état sans modifier la hiérarchie de vos composants.** Ça facilite le partage des Hooks entre plusieurs composants, voire avec la communauté.

Nous parlerons de ça plus en détail dans [Construire vos propres Hooks](#).

Les composants complexes deviennent difficiles à comprendre {#complex-components-become-hard-to-understand}

Nous avons souvent dû maintenir des composants qui étaient simples au départ, pour devenir d'ingérables ramassis de logiques à état et d'effets de bord. Chaque méthode de cycle de vie contient un mélange de logiques sans aucun rapport. Par exemple, des composants peuvent charger des données dans les méthodes `componentDidMount` et `componentDidUpdate`. Toutefois, cette même méthode `componentDidMount` pourrait contenir d'autres logiques dédiées à la configuration d'écouteurs d'événements, qui seront à leur tour nettoyés dans la méthode `componentWillUnmount`.

Le code mutuellement lié dont les évolutions doivent rester cohérentes est divisé en plusieurs parties, alors que du code sans rapport finit par être combiné en une seule méthode. Ça ouvre grand la porte à l'introduction de bugs et incohérences.

Il est fréquemment impossible de découper ces composants en d'autres plus petits, car la logique à état est éparpillée. Ils sont également difficiles à tester. C'est la raison pour laquelle de nombreux utilisateurs de React préfèrent l'associer à une bibliothèque externe de gestion d'état applicatif. Toutefois, ça rajoute souvent encore davantage d'abstraction et vous oblige à jouer entre les fichiers, ce qui complexifie la réutilisation de ces composants.

Pour mettre un terme à ces soucis, **les Hooks vous permettent de découper un composant en petites fonctions basées sur les parties qui sont intrinsèquement liées (comme la configuration d'un abonnement ou le chargement de données)**, plutôt que de forcer leur découpe sur base des méthodes de cycle de vie. Il est aussi possible de gérer l'état local d'un composant avec un réducteur, pour le rendre plus prévisible.

Les classes sont déroutantes pour les gens comme pour les machines {#classes-confuse-both-people-and-machines}

En plus de rendre plus difficiles la réutilisation et l'organisation du code, nous avons remarqué que les classes peuvent constituer une barrière significative à l'apprentissage de React. Vous devez comprendre comment `this` fonctionne en JavaScript, d'une façon très différente de la plupart des langages. Vous devez vous souvenir de lier les gestionnaires d'événements. Sans certaines [propositions de syntaxes](#) encore instables, le code est très verbeux. Les gens peuvent parfaitement comprendre les props, l'état local, et le flux de données descendant mais lutter néanmoins avec les classes. La distinction entre fonctions composants et composants à base de classes, ainsi que les situations où leur usage respectif est approprié, conduisent à des désaccords même entre développeurs React expérimentés.

En outre, React est sorti il y a environ cinq ans, et nous voulons nous assurer qu'il reste pertinent pour les cinq prochaines années. Comme [Svelte](#), [Angular](#), [Glimmer](#), et d'autres l'ont montré, la [compilation anticipée](#) de composants recèle un fort potentiel, surtout si elle ne se limite pas aux gabarits. Récemment, nous avons expérimenté autour du [component folding](#) en utilisant [Prepack](#), et les premiers résultats sont encourageants. Toutefois, nous avons constaté que les composants à base de classes peuvent encourager des approches involontaires qui empêchent de telles optimisations. Les classes présentent aussi des problèmes pour l'outil actuel. Par exemple, les classes ne sont pas efficacement minifiées, et elles rendent le chargement à chaud peu fiable. Nous voulons présenter une API qui permet au code de rester plus aisément optimisable.

Pour résoudre ces problèmes, **les Hooks nous permettent d'utiliser davantage de fonctionnalités de React sans recourir aux classes.** Conceptuellement, les composants React ont toujours été proches des fonctions. Les Hooks tirent pleinement parti des fonctions, sans sacrifier l'esprit pratique de React. Les Hooks donnent accès à des échappatoires impératives et ne vous obligent pas à apprendre des techniques complexes de programmation fonctionnelle ou réactive..

Exemples

[L'aperçu des Hooks](#) est un bon moyen de commencer à apprendre les Hooks.

Stratégie d'adoption progressive {#gradual-adoption-strategy}

TL;DR : nous n'avons aucune intention de retirer les classes de React.

Nous savons que les développeurs React se concentrent sur la sortie de leurs produits et n'ont pas le temps d'explorer chaque nouvelle API qui sort. Les Hooks sont tout nouveaux, et il serait peut-être sage d'attendre que davantage d'exemples et de tutoriels soient disponibles avant d'envisager de les apprendre ou de les adopter.

Nous comprenons aussi que la barre pour ajouter une nouvelle primitive à React est extrêmement haute. Pour les lecteurs curieux, nous avons préparé une [RFC détaillée](#) qui explore plus en détail les raisons derrière les Hooks, et fournit une perspective supplémentaire sur certaines décisions de conception et sur des sources d'inspiration.

Point très important : les Hooks fonctionnent côté à côté avec du code existant, vous pouvez donc les adopter progressivement. Il n'y a aucune raison pressante de migrer vers les Hooks. Nous conseillons d'éviter les « réécritures intégrales », en particulier pour les composants existants complexes à base de classes. Il faut ajuster un peu son modèle mental pour commencer à « penser en Hooks ». À en croire notre

expérience, il vaut mieux s'habituer aux Hooks dans de nouveaux composants non-critiques, et s'assurer que toutes les personnes de l'équipe sont à l'aise avec. Après avoir essayé les Hooks, n'hésitez pas à [nous faire vos retours](#), qu'ils soient positifs ou négatifs.

Nos voulons que les Hooks couvrent tout les cas d'usages des classes, mais **nous continuerons à prendre en charge les composants à base de classes jusqu'à nouvel ordre**. Chez Facebook, nous avons des dizaines de milliers de composants écrit en tant que classes, et nous n'avons absolument pas l'intention de les réécrire. Au lieu de ça, nous avons commencé à utiliser les Hooks dans le nouveau code, côte à côte avec les classes.

Questions fréquemment posées {#frequently-asked-questions}

Nous avons préparé une [FAQ des Hooks](#) qui répond aux questions les plus couramment posées sur les Hooks.

Prochaines étapes {#next-steps}

Arrivé·e sur cette fin de page, vous devriez avoir une idée plus claire des problèmes résolus par les Hooks, mais de nombreux détails restent sans doute obscurs. Ne vous en faites pas ! **En route pour la page suivante, dans laquelle nous commencerons à apprendre les Hooks par l'exemple.**

Les *Hooks* sont une nouveauté de React 16.8. Ils permettent de bénéficier d'un état local et d'autres fonctionnalités de React sans avoir à écrire de classes.

La [page d'introduction](#) présentait les Hooks avec cet exemple :

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

function Example() {
  // Déclare une nouvelle variable d'état, que l'on va appeler « count »
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Vous avez cliqué {count} fois</p>
      <button onClick={() => setCount(count + 1)}>
        Cliquez ici
      </button>
    </div>
  );
}
```

Pour commencer à comprendre comment fonctionnent les Hooks, comparons ce code avec un exemple équivalent à base de classe.

Exemple équivalent avec une classe {#equivalent-class-example}

Si vous avez déjà utilisé les classes en React, ce code devrait vous parler :

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>Vous avez cliqué {this.state.count} fois</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Cliquez ici
        </button>
      </div>
    );
  }
}
```

L'état démarre à `{ count: 0 }`, et nous incrémentons `state.count` en appelant `this.setState()` lorsque l'utilisateur clique sur le bouton. Nous utiliserons des extraits de cette classe dans tout le reste de cette page.

Remarque

Vous vous demandez peut-être pourquoi nous utilisons un compteur plutôt qu'un exemple plus réaliste. Ça nous permet tout simplement de nous concentrer sur l'API pendant que nous faisons nos premiers pas avec les Hooks.

Hooks et fonctions composants {#hooks-and-function-components}

Pour rappel, les fonctions composants en React ressemblent à ceci :

```
const Example = (props) => {
  // Vous pouvez utiliser des Hooks ici !
  return <div />;
}
```

ou à ça :

```
function Example(props) {
  // Vous pouvez utiliser des Hooks ici !
  return <div />;
}
```

Vous les connaissiez peut-être sous le nom de « composants sans état » (*Stateless (Functional) Components ou SFC, NdT*). Comme nous avons maintenant la possibilité d'utiliser l'état local React dans ces composants, nous préférerons le terme « fonctions composants ».

Les Hooks **ne fonctionnent pas** dans les classes. Mais vous pouvez les utiliser pour éviter d'écrire des classes.

Un Hook, qu'est-ce que c'est ? {#whats-a-hook}

Pour notre nouvel exemple, commençons par importer le Hook `useState` de React :

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

function Example() {
  // ...
}
```

Qu'est-ce qu'un Hook ? Un Hook est une fonction qui permet de « se brancher » sur des fonctionnalités React. Par exemple, `useState` est un Hook qui permet d'ajouter l'état local React à des fonctions composants. Nous en apprendrons plus sur les Hooks par la suite.

Quand utiliser un Hook ? Auparavant, si vous écriviez une fonction composant et que vous réalisiez que vous aviez besoin d'un état local à l'intérieur, vous deviez la convertir en classe. Désormais vous pouvez utiliser un Hook à l'intérieur de votre fonction composant. Et c'est justement ce que nous allons faire !

Remarque

Des règles spécifiques existent pour savoir quand utiliser ou ne pas utiliser les Hooks dans un composant. Nous les découvrirons dans les [Règles des Hooks](#).

Déclarer une variable d'état {#declaring-a-state-variable}

Dans une classe, on initialise l'état local `count` à `0` en définissant `this.state` à `{ count: 0 }` dans le constructeur :

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
}
```

Dans une fonction composant, nous ne pouvons pas écrire ou lire `this.state` puisqu'il n'y a pas de `this`. Au lieu de ça, nous appelons directement le Hook `useState` dans notre composant :

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

function Example() {
  // Déclare une nouvelle variable d'état, que nous appellerons « count »
  const [count, setCount] = useState(0);
```

Appeler `useState`, qu'est-ce que ça fait ? Ça déclare une « variable d'état ». Notre variable est appelée `count` mais nous aurions pu l'appeler n'importe comment, par exemple `banane`. C'est un moyen de « préserver » des valeurs entre différents appels de fonctions. `useState` est une nouvelle façon d'utiliser exactement les mêmes possibilités qu'offre `this.state` dans une classe. Normalement, les variables « disparaissent » quand la fonction s'achève mais les variables d'état sont préservées par React.

Qu'est-ce qu'on passe à `useState` comme argument ? Le seul argument à passer au Hook `useState()` est l'état initial. Contrairement à ce qui se passe dans les classes, l'état local n'est pas obligatoirement un objet. Il peut s'agir d'un nombre ou d'une chaîne de caractères si ça nous suffit. Dans notre exemple, nous voulons simplement le nombre de fois qu'un utilisateur a cliqué sur le bouton, nous passerons donc `0` comme état initial pour notre variable. (Si nous voulions stocker deux valeurs différentes dans l'état, nous appellerions `useState()` deux fois.)

Que renvoie `useState` ? Elle renvoie une paire de valeurs : l'état actuel et une fonction pour le modifier. C'est pourquoi nous écrivons `const [count, setCount] = useState()`. C'est semblable à `this.state.count`

et `this.setState` dans une classe, mais ici nous les récupérons en même temps. Si vous n'êtes pas à l'aise avec la syntaxe que nous avons employée, nous y reviendrons [en bas de cette page](#).

Maintenant que nous savons ce que fait le Hook `useState`, notre exemple devrait commencer à être plus clair :

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

function Example() {
  // Déclare une nouvelle variable d'état, que nous appellerons « count »
  const [count, setCount] = useState(0);
```

Nous déclarons une variable d'état appelée `count`, et l'initialisons à `0`. React se rappellera sa valeur entre deux affichages et fournira la plus récente à notre fonction. Si nous voulons modifier la valeur de `count`, nous pouvons appeler `setCount`.

Remarque

Vous vous demandez peut-être pourquoi `useState` n'est pas plutôt appelée `createState` ?

En fait, "create" ne serait pas tout à fait correct puisque l'état n'est créé qu'au premier affichage de notre composant. Les fois suivantes, `useState` nous renvoie l'état actuel. Autrement, ce ne serait pas un état du tout ! Il y a aussi une raison pour laquelle les noms des Hooks commencent *toujours* par `use`. Nous découvrirons laquelle plus tard dans les [Règles des Hooks](#).

Lire l'état {#reading-state}

Quand nous voulons afficher la valeur actuelle de `count` dans une classe, nous récupérons la valeur de `this.state.count` :

```
<p>Vous avez cliqué {this.state.count} fois</p>
```

Dans une fonction, nous pouvons directement utiliser `count` :

```
<p>Vous avez cliqué {count} fois</p>
```

Mettre à jour l'état {#updating-state}

Dans une classe, nous devons appeler `this.setState()` pour mettre à jour l'état `count` :

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Cliquez ici
</button>
```

Dans une fonction, nous récupérons directement `setCount` et `count` comme variables, nous n'avons donc pas besoin de `this` :

```
<button onClick={() => setCount(count + 1)}>
  Cliquez ici
</button>
```

En résumé {#recap}

Il est maintenant temps de **récapituler ce que nous avons appris ligne par ligne** et vérifier que nous avons bien compris.

```
1: import React, { useState } from 'react';
2:
3: function Example() {
4:   const [count, setCount] = useState(0);
5:
6:   return (
7:     <div>
8:       <p>Vous avez cliqué {count} fois</p>
9:       <button onClick={() => setCount(count + 1)}>
10:         Cliquez ici
11:       </button>
12:     </div>
13:   );
14: }
```

- **Ligne 1** : nous importons le Hook `useState` depuis React. Il nous permet d'utiliser un état local dans une fonction composant.
- **Ligne 4** : dans le composant `Example`, nous déclarons une nouvelle variable d'état en appelant le Hook `useState`. Il renvoie une paire de valeurs que nous pouvons nommer à notre guise. Ici, nous appelons notre variable `count` puisqu'elle contient le nombre de clics sur le bouton. Nous l'initialisons à zéro en passant `0` comme seul argument à `useState`. Le second élément renvoyé est une fonction. Elle nous permet de modifier la variable `count`, nous l'appellerons donc `setCount`.
- **Ligne 9** : quand l'utilisateur clique, nous appelons `setCount` avec une nouvelle valeur. React rafraîchira le composant `Example` et lui passera la nouvelle valeur de `count`.

Ça fait peut-être beaucoup à digérer d'un coup. Ne vous pressez pas ! Si vous vous sentez un peu perdu·e, jetez un nouveau coup d'œil au code ci-dessus et essayez de le relire du début à la fin. Promis, une fois que vous essaierez « d'oublier » la manière dont fonctionne l'état local dans les classes, et que vous regarderez ce code avec un regard neuf, ça sera plus clair.

Astuce : que signifient les crochets ? {#tip-what-do-square-brackets-mean}

Vous avez peut-être remarqué les crochets que nous utilisons lorsque nous déclarons une variable d'état :

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

Les noms utilisés dans la partie gauche ne font pas partie de l'API React. Vous pouvez nommer vos variables d'état comme ça vous chante :

```
const [fruit, setFruit] = useState('banane');
```

Cette syntaxe Javascript est appelée « **déstructuration positionnelle** ». Ça signifie que nous créons deux nouvelles variables **fruit** et **setFruit**, avec **fruit** qui reçoit la première valeur renvoyée par **useState**, et **setFruit** qui reçoit la deuxième. C'est équivalent au code ci-dessous :

```
var fruitStateVariable = useState('banana'); // Renvoie une paire
var fruit = fruitStateVariable[0]; // Premier élément dans une paire
var setFruit = fruitStateVariable[1]; // Deuxième élément dans une paire
```

Quand nous déclarons une variable d'état avec **useState**, ça renvoie une paire (un tableau avec deux éléments). Le premier élément est la valeur actuelle, et le deuxième est une fonction qui permet de la modifier. Utiliser **[0]** et **[1]** pour y accéder est un peu déconcertant puisqu'ils ont un sens spécifique. C'est pourquoi nous préférons plutôt utiliser la déstructuration positionnelle.

Remarque

Vous vous demandez peut-être comment React sait à quel composant **useState** fait référence étant donné que nous ne lui passons plus rien de similaire à **this**. Nous répondrons à [cette question](#) ainsi qu'à plein d'autres dans la section FAQ.

Astuce : utiliser plusieurs variables d'état [\[#tip-using-multiple-state-variables\]](#)

Déclarer des variables d'état comme une paire de **[quelquechose, setQuelquechose]** est également pratique parce que ça nous permet de donner des noms *differents* à des variables d'état différentes si nous voulons en utiliser plus d'une :

```
function ExampleWithManyStates() {
  // Déclarer plusieurs variables d'état !
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banane');
  const [todos, setTodos] = useState([{ text: 'Apprendre les Hooks' }]);
}
```

Dans le composant ci-dessus, nous avons **age**, **fruit**, et **todos** comme variables locales, et nous pouvons les modifier indépendamment les unes des autres :

```
function handleOrangeClick() {
  // Similaire à this.setState({ fruit: 'orange' })
```

```
    setFruit('orange');  
}
```

Utiliser plusieurs variables d'état **n'est pas obligatoire**. Les variables d'état peuvent tout à fait contenir des objets et des tableaux, vous pouvez donc toujours regrouper des données ensemble. Cependant, lorsque l'on modifie une variable d'état sa valeur est *remplacée* et non fusionnée, contrairement à `this.setState` dans les classes.

Découvrez les raisons de préférer séparer vos variables d'état [dans la FAQ](#).

Les *Hooks* sont une nouveauté de React 16.8. Ils permettent de bénéficier d'un état local et d'autres fonctionnalités de React sans avoir à écrire de classes.

Le *Hook d'effet* permet l'exécution d'effets de bord dans les fonctions composants :

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

function Example() {
  const [count, setCount] = useState(0);

  // Similaire à componentDidMount et componentDidUpdate :
  useEffect(() => {
    // Met à jour le titre du document via l'API du navigateur
    document.title = `Vous avez cliqué ${count} fois`;
  });

  return (
    <div>
      <p>Vous avez cliqué {count} fois</p>
      <button onClick={() => setCount(count + 1)}>
        Cliquez ici
      </button>
    </div>
  );
}
```

Cet extrait se base sur [l'exemple de compteur présenté à la page précédente](#), avec toutefois une fonctionnalité supplémentaire : le titre du document est mis à jour avec un message personnalisé affichant le nombre de clics.

Charger des données depuis un serveur distant, s'abonner à quelque chose et modifier manuellement le DOM sont autant d'exemples d'effets de bord. Que vous ayez ou non l'habitude de les appeler « effets de bord » (ou juste « effets »), il est hautement probable que vous les ayez déjà utilisés dans vos composants par le passé.

Astuce

Si vous avez l'habitude des méthodes de cycle de vie des classes React, pensez au Hook `useEffect` comme à une combinaison de `componentDidMount`, `componentDidUpdate`, et `componentWillUnmount`.

Il existe deux grands types d'effets de bord dans les composants React : ceux qui ne nécessitent pas de nettoyage, et ceux qui en ont besoin. Examinons cette distinction en détail.

Effets sans nettoyage {#effects-without-cleanup}

Parfois, nous souhaitons **exécuter du code supplémentaire après que React a mis à jour le DOM**. Les requêtes réseau, les modifications manuelles du DOM, et la journalisation sont des exemples courants d'effets qui ne nécessitent aucun nettoyage. Cela s'explique par le fait qu'ils peuvent être oubliés immédiatement

après leur exécution. Comparons donc la manière dont les classes et les Hooks nous permettent d'exprimer ce genre d'effets de bord.

Exemple en utilisant les classes {#example-using-classes}

Dans les composants React à base de classe, la méthode `render` ne devrait causer aucun effet de bord par elle-même. Ce serait trop tôt : ces effets ne sont utiles qu'*après* que React a mis à jour le DOM.

C'est la raison pour laquelle, dans les classes React, nous plaçons les effets de bord dans les méthodes `componentDidMount` et `componentDidUpdate`. En reprenant notre exemple, voici un composant React à base de classe implémentant un compteur qui met à jour le titre du document juste après que React a modifié le DOM :

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `Vous avez cliqué ${this.state.count} fois`;
  }

  componentDidUpdate() {
    document.title = `Vous avez cliqué ${this.state.count} fois`;
  }

  render() {
    return (
      <div>
        <p>Vous avez cliqué {this.state.count} fois</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Cliquez ici
        </button>
      </div>
    );
  }
}
```

Notez la **duplication de code entre ces deux méthodes de cycle de vie du composant**.

En effet, la plupart du temps nous voulons que l'effet de bord soit exécuté tant au montage qu'à la mise à jour du composant. Conceptuellement, nous voulons que l'effet soit exécuté à chaque affichage, mais les composants React à base de classe ne disposent pas d'une telle méthode. Même en déplaçant l'effet de bord dans une fonction à part, on aurait tout de même besoin de l'appeler à deux endroits distincts.

Maintenant, voyons comment faire la même chose avec le Hook `useEffect`.

Exemple en utilisant les Hooks {#example-using-hooks}

Cet exemple figurait déjà en haut de page, mais examinons-le de plus près :

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

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Vous avez cliqué ${count} fois`;
  });

  return (
    <div>
      <p>Vous avez cliqué {count} fois</p>
      <button onClick={() => setCount(count + 1)}>
        Cliquez ici
      </button>
    </div>
  );
}
```

Que fait `useEffect` ? On utilise ce Hook pour indiquer à React que notre composant doit exécuter quelque chose après chaque affichage. React enregistre la fonction passée en argument (que nous appellerons « effet »), et l'appellera plus tard, après avoir mis à jour le DOM. L'effet ci-dessus met à jour le titre du document, mais il pourrait aussi bien charger des données distantes, ou appeler n'importe quelle autre API impérative.

Pourquoi `useEffect` est-elle invoquée à l'intérieur d'un composant ? Le fait d'appeler `useEffect` à l'intérieur de notre composant nous permet d'accéder à la variable d'état `count` (ou à n'importe quelle prop) directement depuis l'effet. Pas besoin d'une API dédiée pour les lire : elle est déjà dans la portée de la fonction. Les Hooks profitent pleinement des fermetures lexicales (*closures, NdT*) de JavaScript au lieu d'introduire de nouvelles API spécifiques à React, là où JavaScript propose déjà une solution.

Est-ce que `useEffect` est appelée après chaque affichage ? Oui ! Elle est exécutée par défaut après le premier affichage et après chaque mise à jour. (Nous verrons comment [personnaliser et optimiser ça](#) ultérieurement.) Au lieu de penser en termes de « montage » et de « démontage », pensez plutôt que les effets arrivent tout simplement « après l'affichage ». React garantit que le DOM a été mis à jour avant chaque exécution des effets.

Explication détaillée {#detailed-explanation}

À présent que nous en savons davantage sur les effets, ces quelques lignes devraient paraître plus claires :

```
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
```

```
    document.title = `Vous avez cliqué ${count} fois`;
  });
}
```

Nous déclarons la variable d'état `count`, puis indiquons à React que nous avons besoin d'utiliser un effet. Nous passons alors une fonction au Hook `useEffect`. Cette fonction est notre effet. À l'intérieur de celui-ci, nous mettons à jour le titre du document en utilisant l'API du navigateur `document.title`. Il est possible d'y lire la dernière valeur de `count`, puisqu'elle est accessible depuis la portée de notre fonction. Lorsque React affichera notre composant, il se souviendra de notre effet, et l'exécutera après avoir mis à jour le DOM. Ce procédé est le même à chaque affichage, dont le tout premier.

Les développeurs JavaScript expérimentés remarqueront sans doute que la fonction passée à `useEffect` sera différente à chaque affichage. C'est voulu, et c'est ce qui nous permet d'accéder à la valeur de `count` depuis l'intérieur de l'effet sans nous inquiéter de l'obsolescence de notre fonction. À chaque nouvel affichage, nous planifions un effet *different*, qui succède au précédent. Dans un sens, les effets font partie intégrante du résultat du rendu : chaque effet « appartient » à un rendu particulier. Nous reviendrons plus en détail sur l'utilité d'un tel comportement [plus bas](#).

Astuce

À l'inverse de `componentDidMount` ou de `componentDidUpdate`, les effets planifiés avec `useEffect` ne bloquent en rien la mise à jour de l'affichage par le navigateur, ce qui rend votre application plus réactive. La majorité des effets n'ont pas besoin d'être synchrones. Dans les cas plus rares où ils pourraient en avoir besoin (comme mesurer les dimensions d'un élément de l'interface), il existe un Hook particulier `useLayoutEffect` avec une API identique à celle de `useEffect`.

Effets avec nettoyage {#effects-with-cleanup}

Nous avons vu précédemment comment écrire des effets de bord ne nécessitant aucun nettoyage. Toutefois, quelques effets peuvent en avoir besoin. Par exemple, **nous pourrions souhaiter nous abonner** à une source de données externe. Dans ce cas-là, il est impératif de nettoyer par la suite pour éviter les fuites de mémoire ! Comparons les approches à base de classe et de Hooks pour y arriver.

Exemple en utilisant les classes {#example-using-classes-1}

Dans une classe React, on s'abonne généralement dans `componentDidMount`, et on se désabonne dans `componentWillUnmount`. Par exemple, imaginons que nous avons un module `ChatAPI` qui permet de nous abonner au statut de connexion d'un ami. Voici comment on pourrait s'abonner et l'afficher en utilisant une classe :

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
```

```

    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

handleStatusChange(status) {
  this.setState({
    isOnline: status.isOnline
  });
}

render() {
  if (this.state.isOnline === null) {
    return 'Chargement...';
  }
  return this.state.isOnline ? 'En ligne' : 'Hors-ligne';
}
}

```

Remarquez l'effet miroir de `componentDidMount` et `componentWillUnmount`. Les méthodes de cycle de vie nous forcent à séparer cette logique alors que conceptuellement le code des deux méthodes a trait au même effet.

Remarque

Les lecteurs les plus attentifs remarqueront sans doute que cet exemple nécessite aussi `componentDidUpdate` pour être tout à fait correct. Nous avons choisi d'ignorer ça pour l'instant mais nous y reviendrons dans [une section ultérieure](#) de cette page.

Exemple en utilisant les Hooks {#example-using-hooks-1}

Voyons comment réécrire notre exemple avec les Hooks.

Instinctivement, vous pourriez imaginer qu'un effet distinct est nécessaire pour le nettoyage. Mais les codes pour s'abonner et se désabonner sont si fortement liés que `useEffect` a été pensé pour les conserver ensemble. Si votre effet renvoie une fonction, React l'exécutera lors du nettoyage :

```

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {

```

```

function handleStatusChange(status) {
  setIsOnline(status.isOnline);
}

ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
// Indique comment nettoyer l'effet :
return function cleanup() {
  ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
};
});

if (isOnline === null) {
  return 'Changement...';
}
return isOnline ? 'En ligne' : 'Hors-ligne';
}

```

Pourquoi notre effet renvoie-t-il une fonction ? Il s'agit d'un mécanisme optionnel de nettoyage des effets. Tout effet peut renvoyer une fonction qui se chargera de son propre nettoyage. Cela permet de conserver les logiques d'abonnement et de désabonnement proches l'une de l'autre. Elles font partie du même effet !

À quel moment précis React nettoie-t-il un effet ? React effectue le nettoyage lorsqu'il démonte le composant. Cependant, comme nous l'avons appris précédemment, les effets sont exécutés à chaque affichage, donc potentiellement plus d'une fois. C'est la raison pour laquelle React nettoie *aussi* les effets du rendu précédent avant de les exécuter une nouvelle fois. Nous verrons [pourquoi ça permet d'éviter des bugs](#) et [comment éviter ce comportement s'il nuit aux performances](#) dans un instant.

Remarque

La fonction renvoyée par l'effet peut parfaitement être anonyme. Dans notre exemple, nous l'avons nommée `cleanup` par souci de clarté, mais vous pouvez renvoyer une fonction fléchée ou lui donner n'importe quel nom.

En résumé {#recap}

Nous avons appris que `useEffect` nous permet d'exprimer différentes sortes d'effets de bord après l'affichage d'un composant. Certains effets ont besoin de nettoyer derrière eux, et peuvent renvoyer une fonction pour ça :

```

useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});

```

D'autres effets n'ont pas besoin de nettoyage, et ne renvoient rien.

```
useEffect(() => {
  document.title = `Vous avez cliqué ${count} fois`;
});
```

Le Hook d'effet traite ces deux cas en une seule API.

Si vous pensez avoir déjà bien saisi le fonctionnement du Hook d'effet, ou si c'en est déjà trop pour vous, n'hésitez pas à passer dès maintenant à la [prochaine page sur les règles des Hooks](#).

Astuces pour l'utilisation des effets {#tips-for-using-effects}

Nous allons maintenant nous pencher sur certaines caractéristiques de `useEffect` qui ne manqueront pas de susciter la curiosité des utilisateurs les plus expérimentés de React. Ne vous sentez pas tenu·e d'y plonger dès à présent. Vous pourrez toujours revenir plus tard sur cette page afin d'y parfaire votre connaissance du Hook d'effet.

Astuce : Utiliser plusieurs effets pour séparer les sujets {#tip-use-multiple-effects-to-separate-concerns}

Un des problèmes soulignés dans les [raisons](#) pour les Hooks, c'est que les méthodes de cycle de vie d'une classe de composant deviennent souvent des ramassis de logiques différentes, alors que celles qui sont liées entre elles sont éparpillées dans plusieurs méthodes. Voici un composant qui implémente à la fois notre exemple de compteur et celui du statut de connexion d'un ami :

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `Vous avez cliqué ${this.state.count} fois`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate() {
    document.title = `Vous avez cliqué ${this.state.count} fois`;
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus()
  }
}
```

```

        this.props.friend.id,
        this.handleStatusChange
    );
}

handleStatusChange(status) {
    this.setState({
        isOnline: status.isOnline
    });
}
// ...

```

Remarquez comme le code qui modifie `document.title` est découpé entre `componentDidMount` et `componentDidUpdate`. Quant à la gestion de l'abonnement, elle est éparpillée entre `componentDidMount` et `componentWillUnmount`. De plus, `componentDidMount` contient du code relatif aux deux tâches.

Comment les Hooks résolvent-ils ce problème ? À l'instar du [Hook useState qui peut être utilisé plusieurs fois](#), il est possible d'utiliser plusieurs effets. Cela nous permet de séparer correctement les sujets sans rapport au sein d'effets distincts :

```

function FriendStatusWithCounter(props) {
    const [count, setCount] = useState(0);
    useEffect(() => {
        document.title = `Vous avez cliqué ${count} fois`;
    });

    const [isOnline, setIsOnline] = useState(null);
    useEffect(() => {
        function handleStatusChange(status) {
            setIsOnline(status.isOnline);
        }
        ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
        return () => {
            ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
        };
    });
    // ...
}

```

Les Hooks permettent de découper le code selon ce qu'il fait plutôt qu'en fonction des noms de méthodes de cycle de vie. React appliquera *tous* les effets utilisés par le composant, dans l'ordre de leur déclaration.

Explication : raisons pour lesquelles les effets sont exécutés à chaque mise à jour
[{#explanation-why-effects-run-on-each-update}](#)

Si vous avez l'habitude des classes, vous pourriez vous demander pourquoi le nettoyage des effets s'effectue après chaque rendu, au lieu d'une seule fois au démontage. Voyons un exemple pratique pour comprendre en

quoi ce choix de conception nous aide à réduire les bugs dans nos composants.

Plus haut dans cette page, nous avons présenté le composant d'exemple `FriendStatus` qui affiche le statut de connexion d'un ami. Notre classe récupère `friend.id` depuis `this.props`, s'abonne au statut de connexion une fois le composant monté, et se désabonne au démontage :

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
```

Mais que se passe-t-il si la propriété `friend` change alors que le composant est affiché ? Notre composant continuerait d'afficher le statut de connexion de notre ami initial. C'est un bug. Nous causerions également une fuite de mémoire ou un plantage au démontage, la fonction de désabonnement utilisant l'ID du nouvel ami.

Dans un composant à base de classe, il faudrait ajouter `componentDidUpdate` pour gérer ce cas :

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentDidUpdate(prevProps) {
  // Se désabonne du statut de l'ami précédent
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // S'abonne au statut du prochain ami
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
```

```
        this.handleStatusChange  
    );  
}
```

Les applications React souffrent fréquemment de bugs dus à l'oubli d'une gestion correcte de `componentDidUpdate`.

Maintenant, examinez ce même composant qui utiliserait des Hooks :

```
function FriendStatus(props) {  
    // ...  
    useEffect(() => {  
        // ...  
        ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);  
        return () =>  
            ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);  
    });  
}
```

Il ne rencontre pas ce bug. (Mais nous n'y avons apporté aucun changement.)

Pas besoin de code spécifique pour gérer les mises à jour puisque `useEffect` les traite *par défaut*. Le hook nettoie les effets précédents avant d'appliquer les suivants. Pour illustrer ça, voici la séquence des abonnements et des désabonnements que ce composant pourrait produire au fil du temps :

```
// Montage avec les propriétés { friend: { id: 100 } }  
ChatAPI.subscribeToFriendStatus(100, handleStatusChange);      // Exécute l'effet 1  
  
// Mise à jour avec les propriétés { friend: { id: 200 } }  
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Nettoie l'effet 1  
ChatAPI.subscribeToFriendStatus(200, handleStatusChange);      // Exécute l'effet 2  
  
// Mise à jour avec les propriétés { friend: { id: 300 } }  
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // Nettoie l'effet 2  
ChatAPI.subscribeToFriendStatus(300, handleStatusChange);      // Exécute l'effet 3  
  
// Démontage  
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // Nettoie l'effet 3
```

Ce comportement par défaut garantit la cohérence et résout les bugs les plus courants des composants à base de classe qui oublient de gérer les mises à jour.

Astuce : optimiser les performances en sautant des effets {#tip-optimizing-performance-by-skipping-effects}

Dans certains cas, nettoyer ou exécuter l'effet après chaque affichage risque de nuire aux performances. Dans les composants à base de classe, une solution consiste à comparer `prevProps` ou `prevState` dans

`componentDidUpdate` :

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `Vous avez cliqué ${this.state.count} fois`;
  }
}
```

Ce genre de comportement est tellement courant qu'il est intégré dans l'API du Hook `useEffect`. Il est possible d'indiquer à React de *sauter* l'exécution d'un effet si certaines valeurs n'ont pas été modifiées entre deux affichages. Pour cela, il suffit de passer une liste comme second argument optionnel à `useEffect` :

```
useEffect(() => {
  document.title = `Vous avez cliqué ${count} fois`;
}, [count]); // N'exécute l'effet que si count a changé
```

Dans l'exemple ci-dessus, nous passons `[count]` comme second argument. Qu'est-ce que ça signifie ? Si `count` vaut `5`, et que notre composant est ré-affiché avec `count` toujours égal à `5`, React comparera le `[5]` de l'affichage précédent au `[5]` du suivant. Comme tous les éléments de la liste sont identiques (`5 === 5`), React n'exécutera pas l'effet. Et voilà notre optimisation.

Quand le composant est ré-affiché avec `count` égal à `6`, React comparera la liste d'éléments `[5]` de l'affichage précédent avec la liste `[6]` du suivant. Cette fois, React ré-exécutera l'effet car `5 !== 6`. Dans le cas où la liste contiendrait plusieurs éléments, React ré-appliquera l'effet si au moins l'un d'entre eux est différent de sa version précédente.

Le fonctionnement est le même pour la phase de nettoyage :

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // Ne se ré-abonne que si props.friend.id change
```

À l'avenir, ce second argument pourrait être injecté automatiquement au moment de la compilation.

Remarque

Si vous utilisez cette optimisation, assurez-vous que votre tableau inclut bien **toutes les valeurs dans la portée du composant (telles que les props et l'état local) qui peuvent changer avec le temps et sont utilisées par l'effet**. Sinon, votre code va référencer des valeurs obsolètes issues des rendus

précédents. Vous pouvez en apprendre davantage sur [la façon de gérer les dépendances à des fonctions](#) et comment faire quand [les dépendances listées changent trop souvent](#).

Si vous voulez exécuter un effet et le nettoyer une seule fois (au montage puis au démontage), vous pouvez passer un tableau vide (`[]`) comme second argument. Ça indique à React que votre effet ne dépend *d'aucune* valeur issue des props ou de l'état local, donc il n'a jamais besoin d'être ré-exécuté. Il ne s'agit pas d'un cas particulier : ça découle directement de la façon dont le tableau des dépendances fonctionne à la base.

Si vous passez un tableau vide (`[]`), les props et l'état local vus depuis l'intérieur de l'effet feront toujours référence à leurs valeurs initiales. Même si passer `[]` comme second argument vous rapproche du modèle mental habituel de `componentDidMount` et `componentWillUnmount`, il y a en général de [meilleures solutions](#) pour éviter de ré-exécuter les effets trop souvent. Par ailleurs, ne perdez pas de vue que React défère l'exécution de `useEffect` jusqu'à ce que le navigateur ait fini de rafraîchir l'affichage, du coup y faire plus de travail est moins un problème.

Nous vous conseillons d'utiliser la règle `exhaustive-deps` fournie par le module [`eslint-plugin-react-hooks`](#). Elle vous avertira si des dépendances sont mal spécifiées et vous suggèrera un correctif.

Les *Hooks* sont une nouveauté de React 16.8. Ils permettent de bénéficier d'un état local et d'autres fonctionnalités de React sans avoir à écrire de classes.

Construire vos propres Hooks vous permet d'extraire la logique d'un composant sous forme de fonctions réutilisables.

Lorsque nous apprenions à utiliser [le Hook d'effet](#), nous avons vu ce composant d'une application de chat qui affiche un message selon qu'un ami est en ligne ou hors-ligne.

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () =>
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  });
}

if (isOnline === null) {
  return 'Chargement...';
}
return isOnline ? 'En ligne' : 'Hors-ligne';
}
```

Disons maintenant que notre application de chat possède aussi une liste de contacts et que nous souhaitons afficher en vert les noms des utilisateurs qui sont en ligne. Nous pourrions copier et coller une logique similaire à celle ci-dessus dans notre composant **FriendListItem** mais ça ne serait pas idéal :

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

function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () =>
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  });
}
```

```

    });

    return (
      <li style={{ color: isOnline ? 'green' : 'black' }}>
        {props.friend.name}
      </li>
    );
}

```

Nous aimerais plutôt partager cette logique entre `FriendStatus` et `FriendListItem`.

Traditionnellement en React, nous avions deux manières répandues de partager une logique d'état entre des composants : les [props de rendu](#) et les [composants d'ordre supérieur](#). Nous allons voir comment les Hooks règlent la majeure partie de ces problèmes sans vous obliger à ajouter des composants dans l'arbre.

Extraire un Hook personnalisé {#extracting-a-custom-hook}

Lorsque nous souhaitons partager de la logique entre deux fonctions JavaScript, nous l'extrayons dans une troisième fonction. Les composants et les Hooks sont des fonctions, ça fonctionne donc aussi pour eux !

Un Hook personnalisé est une fonction JavaScript dont le nom commence par "use" et qui peut appeler d'autres Hooks. Par exemple, `useFriendStatus` ci-dessous est notre premier Hook personnalisé :

```

import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}

```

Il n'y a rien de nouveau à l'intérieur ; la logique provient des composants vus plus haut. Comme lorsque vous êtes dans un composant, assurez-vous d'appeler les autres Hooks de façon inconditionnelle et au niveau racine de votre Hook personnalisé.

Contrairement à un composant React, un Hook personnalisé n'a pas besoin d'avoir une signature particulière. Nous pouvons décider s'il a besoin d'accepter des arguments et ce qu'il doit éventuellement renvoyer. En

d'autres termes, c'est une simple fonction. Son nom doit toujours commencer par `use` pour qu'au premier coup d'œil vous sachiez que les [règles des Hooks](#) lui sont applicables.

L'objectif de notre Hook `useFriendStatus` est de nous abonner au statut d'un ami. C'est pourquoi il prend `friendID` comme argument et nous renvoie si notre ami est en ligne :

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  return isOnline;
}
```

Voyons maintenant comment nous pouvons utiliser notre Hook personnalisé.

Utiliser un Hook personnalisé {#using-a-custom-hook}

À la base, notre but était de supprimer la logique dupliquée entre les composants `FriendStatus` et `FriendListItem`. Les deux veulent savoir si un ami est en ligne.

Maintenant que nous avons extrait cette logique dans un hook `useFriendStatus`, nous pouvons *simplement l'utiliser* :

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Chargement...';
  }
  return isOnline ? 'En ligne' : 'Hors-ligne';
}
```

```
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

Ce code est-il équivalent aux exemples de départ ? Oui, il fonctionne exactement de la même manière. Si vous regardez de plus près, vous remarquerez que nous n'avons en rien changé le comportement. Tout ce que nous avons fait, c'est déplacer du code commun à deux fonctions dans une fonction séparée. [Les Hooks](#)

personnalisés sont une convention qui découle naturellement du principe des Hooks, plutôt qu'une véritable fonctionnalité de React.

Dois-je nommer mes Hooks personnalisés en commençant par "use" ? Oui, s'il vous plaît. Cette convention est très importante. Sans elle, nous ne pourrions pas vérifier automatiquement les violations des règles des Hooks car nous ne pourrions être sûrs qu'une fonction contient des appels à des Hooks.

Est-ce que deux composants utilisant le même Hook partagent le même état ? Non. Les Hooks personnalisés sont un mécanisme de réutilisation de *logique à état* (comme la mise en place d'un abonnement et la mémorisation de sa valeur courante), mais chaque fois qu'on utilise un Hook personnalisé, tous les états et effets qu'il utilise sont totalement isolés.

Comment l'état d'un Hook personnalisé est-il isolé ? Chaque *appel* à un Hook se voit attribuer un état isolé. Comme nous appelons `useFriendStatus` directement, du point de vue de React notre composant appelle simplement `useState` et `useEffect`. Et comme nous l'avons [appris précédemment](#), nous pouvons appeler `useState` et `useEffect` plusieurs fois dans un composant et ils seront complètement indépendants.

Astuce: passer de l'information entre les Hooks {#tip-pass-information-between-hooks}

Comme les Hooks sont des fonctions, nous pouvons passer de l'information entre eux.

Pour illustrer ça, nous allons utiliser un autre composant de notre hypothétique exemple de chat. Voici un sélecteur de destinataire de message qui affiche si l'ami sélectionné est en ligne :

```
const friendList = [
  { id: 1, name: 'Phoebe' },
  { id: 2, name: 'Rachel' },
  { id: 3, name: 'Ross' },
];

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID);

  return (
    <>
      <Circle color={isRecipientOnline ? 'green' : 'red'} />
      <select
        value={recipientID}
        onChange={e => setRecipientID(Number(e.target.value))}>
        </select>
      {friendList.map(friend => (
        <option key={friend.id} value={friend.id}>
          {friend.name}
        </option>
      ))}
    </>
  );
}
```

Nous gardons l'ID de l'ami sélectionné dans la variable d'état `recipientID`, et nous la mettons à jour si l'utilisateur sélectionne un ami différent dans le `<select>` de la liste.

Puisque l'appel au Hook `useState` nous renvoie la dernière valeur de la variable d'état `recipientID`, nous pouvons la passer en argument à notre Hook personnalisé `useFriendStatus` :

```
const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);
```

Nous pouvons ainsi savoir si l'ami *actuellement sélectionné* est en ligne. Si nous sélectionnons un autre ami et mettons à jour la variable d'état `recipientID`, notre Hook `useFriendStatus` va se désabonner de l'ami précédemment sélectionné et s'abonner au statut de son remplaçant.

useYourImagination() {#useyourimagination}

Les Hooks personnalisés offrent une souplesse de partage de logique qui n'était pas possible avec les composants React auparavant. Vous pouvez écrire des Hooks personnalisés qui couvrent un large éventail de cas d'usage tels que la gestion de formulaires, les animations, les abonnements déclaratifs, les horloges et probablement de nombreux autres auxquels nous n'avons pas pensé. Qui plus est, vous pouvez construire des Hooks qui sont aussi simples à utiliser que les fonctionnalités fournies par React.

Essayez de résister à la tentation de faire des extractions prématurées de Hooks. À présent que les fonctions composants peuvent en faire plus, il est probable que les fonctions composants de votre base de code grossissent, en moyenne. C'est normal : ne vous sentez pas *obligé-e* d'en extraire des Hooks. Ceci dit, nous vous encourageons tout de même à commencer à repérer des cas où un Hook personnalisé pourrait masquer une logique complexe derrière une interface simple, ou aider à démêler un composant dont le code est incompréhensible.

Par exemple, peut-être avez-vous un composant complexe qui contient beaucoup d'états locaux gérés de manière *ad hoc*. `useState` ne facilite pas la centralisation de la logique de mise à jour, du coup vous préféreriez peut-être la réécrire sous forme de réducteur [Redux](#) :

```
function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, {
        text: action.text,
        completed: false
      }];
    // ... autres actions ...
    default:
      return state;
  }
}
```

Les réducteurs sont très pratiques à tester en isolation, et permettent d'exprimer lisiblement des logiques de mise à jour complexes. Vous pouvez toujours les découper en réducteurs plus petits si besoin. Cependant,

vous pourriez aussi apprécier la gestion d'état local de React, ou ne pas vouloir installer une autre bibliothèque.

Et si nous pouvions écrire un Hook `useReducer` qui nous permettrait de gérer l'état *local* de notre composant à l'aide d'un réducteur ? Une version simplifiée pourrait ressembler à ceci :

```
function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}
```

Nous pourrions maintenant l'utiliser dans notre composant, et laisser le réducteur piloter sa gestion d'état :

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer, []);

  function handleAddClick(text) {
    dispatch({ type: 'add', text });
  }

  // ...
}
```

Le besoin de gérer un état local avec un réducteur dans un composant complexe est si fréquent que nous avons intégré le Hook `useReducer` directement dans React. Vous le trouverez avec d'autres Hooks prédéfinis dans la [référence de l'API des Hooks](#).

Les *Hooks* sont une nouveauté de React 16.8. Ils permettent de bénéficier d'un état local et d'autres fonctionnalités de React sans avoir à écrire de classes.

Cette page décrit l'API des Hooks prédéfinis de React.

Si les Hooks sont nouveaux pour vous, vous voudrez peut-être consulter [l'aperçu](#) en premier. Vous trouverez peut-être aussi des informations utiles dans [la foire aux questions](#).

- [Les Hooks de base](#)
 - `useState`
 - `useEffect`
 - `useContext`
- [Hooks supplémentaires](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`

Les Hooks de base {#basic-hooks}

`useState` {#usestate}

```
const [state, setState] = useState(initialState);
```

Renvoie une valeur d'état local et une fonction pour la mettre à jour.

Pendant le rendu initial, l'état local (`state`) a la même valeur que celle passée en premier argument (`initialState`).

La fonction `setState` permet de mettre à jour l'état local. Elle accepte une nouvelle valeur d'état local et planifie un nouveau rendu du composant.

```
setState(newState);
```

Au cours des rendus suivants, la première valeur renvoyée par `useState` sera toujours celle de l'état local le plus récent, une fois les mises à jour effectuées.

Remarque

React garantit que l'identité de la fonction `setState` est stable et ne changera pas d'un rendu à l'autre. C'est pourquoi on peut l'omettre de la liste des dépendances de `useEffect` et `useCallback` en tout sécurité.

Mises à jour fonctionnelles {#functional-updates}

Si le nouvel état local est déduit de l'état local précédent, vous pouvez passer une fonction à `useState`. Cette fonction recevra la valeur précédente de l'état local et renverra une nouvelle valeur de l'état local. Voici un exemple d'un composant compteur qui utilise les deux formes de `useState` :

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Total : {count}
      <button onClick={() => setCount(initialCount)}>Réinitialiser</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

Les boutons « + » et « - » utilisent la forme fonctionnelle, puisque la nouvelle valeur est calculée à partir de la valeur précédente. Le bouton « Réinitialiser » utilise quant à lui la forme normale puisqu'il remet toujours le total à sa valeur initiale.

Si votre fonction de mise à jour renvoie exactement la même valeur que l'état courant, le rendu ultérieur sera carrément sauté.

Remarque

À l'inverse de la méthode `useState` que l'on trouve dans les composants définis à l'aide d'une classe, `useState` ne fusionne pas automatiquement les objets de mise à jour. Vous pouvez imiter ce comportement en combinant la forme fonctionnelle de mise à jour avec la syntaxe de *spread* des objets :

```
setState(prevState => {
  // Object.assign marcherait aussi
  return {...prevState, ...updatedValues};
});
```

Il est aussi possible d'utiliser `useReducer`, qui est plus adapté pour gérer les objets d'état local qui contiennent plusieurs sous-valeurs.

État local initial paresseux {#lazy-initial-state}

Le rendu initial utilise l'argument `initialState` comme état local. Au cours des rendus suivants, il est ignoré. Si l'état local initial est le résultat d'un calcul coûteux, vous pouvez plutôt fournir une fonction qui sera exécutée seulement au cours du rendu initial :

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

Abandon de la mise à jour de l'état local {#bailing-out-of-a-state-update}

Si vous mettez à jour un Hook d'état avec la même valeur que son état actuel, React abandonnera cette mise à jour, ce qui signifie qu'aucun nouveau rendu des enfants ne sera effectué et qu'aucun effet ne sera déclenché. (React utilise [l'algorithme de comparaison `Object.is`](#).)

Remarquez que React peut quand même avoir besoin d'afficher ce composant à nouveau avant d'abandonner. Ça ne devrait pas poser problème car React n'ira pas « plus profondément » dans l'arbre. Si vous effectuez des calculs coûteux lors du rendu, vous pouvez les optimiser avec [useMemo](#).

useEffect {#useeffect}

```
useEffect(didUpdate);
```

Accepte une fonction qui contient du code impératif, pouvant éventuellement produire des effets.

L'utilisation de mutations, abonnements, horloges, messages de journalisation, et autres effets de bord n'est pas autorisée au sein du corps principal d'une fonction composant (qu'on appelle la *phase de rendu* de React). Autrement ça pourrait entraîner des bugs déconcertants et des incohérences dans l'interface utilisateur (UI).

Pour ce faire, utilisez plutôt [useEffect](#). La fonction fournie à [useEffect](#) sera exécutée après que le rendu est apparu sur l'écran. Vous pouvez considérer les effets comme des échappatoires pour passer du monde purement fonctionnel de React au monde impératif.

Par défaut, les effets de bord s'exécutent après chaque rendu, mais vous pouvez choisir d'en exécuter certains [uniquement quand certaines valeurs ont changé](#).

Nettoyage d'un effet de bord {#cleaning-up-an-effect}

Souvent, les effets de bord créent des ressources qui nécessitent d'être nettoyées avant que le composant ne quitte l'écran, telles qu'un abonnement ou l'ID d'une horloge. Pour ce faire, la fonction fournie à [useEffect](#) peut renvoyer une fonction de nettoyage. Par exemple, pour créer un abonnement :

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Nettoyage de l'abonnement
    subscription.unsubscribe();
  };
});
```

La fonction de nettoyage est exécutée avant que le composant ne soit retiré de l'UI pour éviter les fuites de mémoire. Par ailleurs, si un composant s'affiche plusieurs fois (comme c'est typiquement le cas), **l'effet de bord précédent est nettoyé avant l'exécution du prochain effet de bord**. Dans notre exemple, ça veut dire qu'un nouvel abonnement est créé à chaque mise à jour. Pour éviter d'exécuter un effet de bord à chaque mise à jour, voyez la section sur l'exécution conditionnelle un peu plus loin.

Moment d'exécution des effets de bord {#timing-of-effects}

Contrairement à `componentDidMount` et `componentDidUpdate`, la fonction fournie à `useEffect` est exécutée de façon différée, **après** la mise en page et l'affichage. `useEffect` est donc bien adapté pour une grande partie des effets de bord, comme la mise en place d'abonnements et de gestionnaires d'événements, puisque la plupart des types de tâche ne devraient pas gêner la mise à jour de l'affichage par le navigateur.

Cependant, tous les effets de bord ne peuvent pas être différés. Par exemple, une mutation du DOM qui est visible pour l'utilisateur doit s'exécuter de manière synchrone avant l'affichage suivant, afin que l'utilisateur ne puisse pas percevoir une incohérence visuelle. (La distinction est conceptuellement similaire à celle entre écouteur d'événement passif et actif.) Pour ces types d'effets de bord, React fournit un Hook supplémentaire appelé `useLayoutEffect`. Il a la même signature que `useEffect`, et s'en distingue seulement par le moment où il s'exécute.

Bien que `useEffect` soit différé jusqu'à ce que le navigateur ait terminé l'affichage, son exécution est garantie avant les rendus ultérieurs. React traitera toujours les effets de bord des rendus précédents avant de commencer une nouvelle mise à jour.

Exécution conditionnelle d'un effet de bord {#conditionally-firing-an-effect}

Le comportement par défaut des effets de bord consiste à exécuter l'effet après chaque affichage. Ainsi, un effet est toujours recréé si une de ses entrées (les données dont il dépend) change.

Cependant, ça pourrait être exagéré dans certains cas, comme dans l'exemple avec l'abonnement dans la section précédente. On n'a pas besoin d'un nouvel abonnement à chaque mise à jour, mais seulement si la prop `source` a changé.

Pour mettre ça en œuvre, fournissez un deuxième argument à `useEffect` qui consiste en un tableau de valeurs dont l'effet dépend. Notre exemple mis à jour ressemble maintenant à ça :

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

L'abonnement sera maintenant recréé uniquement quand `props.source` change.

Remarque

Si vous utilisez cette optimisation, assurez-vous que votre tableau inclut bien **toutes les valeurs dans la portée du composant (telles que les props et l'état local) qui peuvent changer avec le temps et sont utilisées par l'effet**. Sinon, votre code va référencer des valeurs obsolètes issues des rendus précédents. Vous pouvez en apprendre davantage sur [la façon de gérer les dépendances à des fonctions](#) et comment faire quand [les dépendances listées changent trop souvent](#).

Si vous voulez exécuter un effet et le nettoyer une seule fois (au montage puis au démontage), vous pouvez passer un tableau vide ([]) comme deuxième argument. Ça indique à React que votre effet ne dépend *d'aucune* valeur issue des props ou de l'état local, donc il n'a jamais besoin d'être ré-exécuté. Il ne s'agit pas d'un cas particulier : ça découle directement de la façon dont le tableau des dépendances fonctionne.

Si vous passez un tableau vide ([]), les props et l'état local vus depuis l'intérieur de l'effet feront toujours référence à leurs valeurs initiales. Même si passer [] comme deuxième argument vous rapproche du modèle mental habituel de `componentDidMount` et `componentWillUnmount`, il y a en général de [meilleures solutions](#) pour éviter de ré-exécuter les effets trop souvent. Par ailleurs, ne perdez pas de vue que React défère l'exécution de `useEffect` jusqu'à ce que la navigateur ait fini de rafraîchir l'affichage, du coup y faire plus de travail est moins un problème.

Nous vous conseillons d'utiliser la règle `exhaustive-deps` fournie par le module `eslint-plugin-react-hooks`. Elle vous avertira si des dépendances sont mal spécifiées et vous suggérera un correctif.

Le tableau d'entrées n'est pas fourni comme argument à la fonction d'effet. Conceptuellement cependant, c'est en quelque sorte ce qui se passe : chaque valeur référencée dans la fonction d'effet devrait aussi apparaître dans le tableau d'entrées. À l'avenir, un compilateur suffisamment avancé pourrait créer ce tableau automatiquement.

`useContext` {#usecontext}

```
const value = useContext(MyContext);
```

Accepte un objet contexte (la valeur renvoyée par `React.createContext`), et renvoie la valeur actuelle du contexte. Celle-ci est déterminée par la prop `value` du plus proche `<MyContext.Provider>` au-dessus du composant dans l'arbre.

Quand le plus proche `<MyContext.Provider>` au-dessus du composant est mis à jour, ce Hook va déclencher un rafraîchissement avec la `value` la plus récente passée au fournisseur `MyContext`. Même si un ancêtre utilise `React.memo` ou `shouldComponentUpdate`, le rendu aura quand même à nouveau lieu à partir du composant qui recourt à `useContext`.

N'oubliez pas que l'argument de `useContext` doit être *l'objet contexte lui-même* :

- **Correct** : `useContext(MyContext)`
- **Erroné** : `useContext(MyContext.Consumer)`
- **Erroné** : `useContext(MyContext.Provider)`

Un composant qui appelle `useContext` se rafraîchira toujours quand la valeur du contexte change. Si ce rafraîchissement est coûteux, vous pouvez [l'optimiser grâce à la mémoïsation](#).

Astuce

Si vous aviez l'habitude de l'API de Contexte avant les Hooks, `useContext(MyContext)` est équivalent à `static contextType = MyContext` dans une classe, ou à `<MyContext.Consumer>`.

`useContext(MyContext)` vous permet seulement de *lire* le contexte et de vous abonner à ses modifications. Vous aurez toujours besoin d'un `<MyContext.Provider>` plus haut dans l'arbre pour *fournir* une valeur de contexte.

Un exemple consolidé avec `Context.Provider`

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);

  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      Je suis stylé par le contexte de thème !
    </button>
  );
}
```

Cet exemple est une version modifiée pour utiliser les Hooks de l'exemple dans le [guide avancé des Contextes](#), au sein duquel vous pourrez trouver davantage d'informations sur l'utilisation appropriée de Context.

Hooks supplémentaires {#additional-hooks}

Les Hooks qui suivent sont soit des variantes des Hooks basiques des sections précédentes, soit seulement nécessaires pour des cas à la marge spécifiques. Ne vous sentez pas obligé·e de les apprendre dès le départ.

useReducer {#usereducer}

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

Alternative à [useState](#). Accepte un réducteur de type `(state, action) => newState`, et renvoie l'état local actuel accompagné d'une méthode `dispatch`. (Si vous avez l'habitude de Redux, vous savez déjà comment ça fonctionne.)

`useReducer` est souvent préférable à [useState](#) quand vous avez une logique d'état local complexe qui comprend plusieurs sous-valeurs, ou quand l'état suivant dépend de l'état précédent. `useReducer` vous permet aussi d'optimiser les performances pour des composants qui déclenchent des mises à jours profondes puisque [vous pouvez fournir dispatch à la place de fonctions de rappel](#).

Voici l'exemple du composant compteur du paragraphe [useState](#) ré-écrit avec un réducteur :

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Total : {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

Remarque

React garantit que l'identité de la fonction `dispatch` est stable et ne changera pas d'un rendu à l'autre. C'est pourquoi on peut l'omettre de la liste des dépendances de `useEffect` et `useCallback` en tout sécurité.

Préciser l'état local initial {#specifying-the-initial-state}

Il existe deux manières différentes d'initialiser l'état de `useReducer`. Vous pouvez choisir l'une ou l'autre suivant le cas. La manière la plus simple consiste à fournir l'état initial comme deuxième argument :

```
const [state, dispatch] = useReducer(  
  reducer,  
  {count: initialCount}  
)
```

Remarque

React n'utilise pas la convention d'argument `state = initialState` popularisée par Redux. La valeur initiale doit parfois dépendre de props et c'est donc plutôt l'appel du Hook qui la précise. Si vous avez déjà une préférence bien arrêtée là-dessus, vous pouvez utiliser `useReducer(reducer, undefined, reducer)` pour simuler le comportement de Redux, mais nous ne vous le conseillons pas.

Initialisation paresseuse {#lazy-initialization}

Vous pouvez aussi créer l'état local initial paresseusement. Pour ce faire, vous pouvez fournir une fonction `init` comme troisième argument. L'état initial sera alors égal à `init(initialArg)`.

Ça vous permet d'extraire la logique pour calculer l'état local initial hors du réducteur. C'est aussi pratique pour réinitialiser l'état local en réponse à une action ultérieure :

```
function init(initialCount) {  
  return {count: initialCount};  
}  
  
function reducer(state, action) {  
  switch (action.type) {  
    case 'increment':  
      return {count: state.count + 1};  
    case 'decrement':  
      return {count: state.count - 1};  
    case 'reset':  
      return init(action.payload);  
    default:  
      throw new Error();  
  }  
}  
  
function Counter({initialCount}) {
```

```

const [state, dispatch] = useReducer(reducer, initialCount, init);
return (
  <>
    Total : {state.count}
    <button
      onClick={() => dispatch({type: 'reset', payload: initialCount})}>
      Réinitialiser
    </button>
    <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    <button onClick={() => dispatch({type: 'increment'})}>+</button>
  </>
);
}

```

Abandon d'un dispatch [\[#bailing-out-of-a-dispatch\]](#)

Si vous renvoyez la même valeur que l'état actuel dans un Hook de réduction, React abandonnera la mise à jour, ce qui signifie qu'aucun nouveau rendu des enfants ne sera effectué et qu'aucun effet ne sera déclenché. (React utilise [l'algorithme de comparaison Object.is](#).)

Remarquez que React pourrait encore avoir besoin de mettre à jour ce composant spécifique avant de lâcher l'affaire. Ça ne devrait pas vous soucier car React n'ira pas inutilement « plus profond » dans l'arbre. Si vous effectuez des calculs coûteux lors du rendu, vous pouvez les optimiser avec [useMemo](#).

useCallback [\[#usecallback\]](#)

```

const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);

```

Renvoie une fonction de rappel [mémoisée](#)

Fournissez une fonction de rappel et un tableau d'entrées. [useCallback](#) renverra une version mémoisée de la fonction de rappel qui changera uniquement si une des entrées a changé. C'est utile pour passer des fonctions de rappel à des composants enfants optimisés qui se basent sur une égalité référentielle pour éviter des rendus superflus (par exemple avec [shouldComponentUpdate](#)).

[useCallback\(fn, inputs\)](#) est équivalent à [useMemo\(\(\) => fn, inputs\)](#).

Remarque

Le tableau d'entrées n'est pas fourni comme argument à la fonction de rappel. Conceptuellement cependant, c'est en quelque sorte ce qui se passe : chaque valeur référencée dans la fonction de rappel devrait aussi apparaître dans le tableau d'entrées. À l'avenir, un compilateur suffisamment avancé pourrait créer ce tableau automatiquement.

Nous vous conseillons d'utiliser la règle `exhaustive-deps` fournie par le module `eslint-plugin-react-hooks`. Elle vous avertira si des dépendances sont mal spécifiées et vous suggérera un correctif.

useMemo {#usememo}

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Renvoie une valeur **mémoisée**

Fournissez une fonction de « création » et un tableau d'entrées. `useMemo` recalculera la valeur mémoisée seulement si une des entrées a changé. Cette optimisation permet d'éviter des calculs coûteux à chaque rendu.

Rappelez-vous que la fonction fournie à `useMemo` s'exécute pendant le rendu. N'y faites rien que vous ne ferez pas normalement pendant un rendu. Par exemple, les effets de bord doivent passer par `useEffect`, et non `useMemo`.

Si vous ne fournissez aucun tableau, une nouvelle valeur sera calculée à chaque appel.

Vous pouvez vous appuyer sur `useMemo` comme un moyen d'optimiser les performances, mais pas comme une garantie sémantique. À l'avenir, React pourrait choisir « d'oublier » certaines valeurs précédemment mémoisées et de les recalculer au rendu suivant, par exemple pour libérer la mémoire exploitée par des composants présents hors de l'écran. Écrivez votre code de façon à ce qu'il fonctionne sans `useMemo` et ajoutez-le ensuite pour optimiser les performances.

Remarque

Le tableau d'entrées n'est pas fourni comme argument à la fonction. Conceptuellement cependant, c'est en quelque sorte ce qui se passe : chaque valeur référencée dans la fonction devrait aussi apparaître dans le tableau d'entrées. À l'avenir, un compilateur suffisamment avancé pourrait créer ce tableau automatiquement.

Nous vous conseillons d'utiliser la règle `exhaustive-deps` fournie par le module `eslint-plugin-react-hooks`. Elle vous avertira si des dépendances sont mal spécifiées et vous suggérera un correctif.

useRef {#useref}

```
const refContainer = useRef(initialValue);
```

`useRef` renvoie un objet `ref` modifiable dont la propriété `current` est initialisée avec l'argument fourni (`initialValue`). L'objet renvoyé persistera pendant toute la durée de vie composant.

Un cas d'usage courant consiste à accéder à un enfant de manière impérative :

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
```

```
// `current` fait référence au champ textuel monté dans le DOM
inputEl.current.focus();
};

return (
  <>
  <input ref={inputEl} type="text" />
  <button onClick={onButtonClick}>Donner le focus au champ</button>
</>
);
}
```

En gros, `useRef` est comme une « boîte » qui pourrait contenir une valeur modifiable dans sa propriété `.current`.

Vous avez peut-être l'habitude d'utiliser des refs principalement pour [accéder au DOM](#). Si vous passez un objet ref à React avec `<div ref={myRef} />`, React calera sa propriété `.current` sur le nœud DOM correspondant chaque fois que ce dernier change.

Ceci dit, `useRef()` est utile au-delà du seul attribut `ref`. C'est [pratique pour garder des valeurs modifiables sous la main](#), comme lorsque vous utilisez des champs d'instance dans les classes.

Ça fonctionne parce que `useRef()` crée un objet JavaScript brut. La seule différence entre `useRef()` et la création manuelle d'un objet `{current: ...}`, c'est que `useRef` vous donnera le même objet à chaque rendu.

Gardez à l'esprit que `useRef` ne vous *notify pas* quand le contenu change. Modifier la propriété `.current` n'entraîne pas un rafraîchissement. Si vous voulez exécuter du code quand React attache ou détache une ref sur un nœud DOM, vous voudrez sans doute utiliser plutôt une [ref à base de fonction de rappel](#).

`useImperativeHandle` #useimperativehandle

```
useImperativeHandle(ref, createHandle, [deps])
```

`useImperativeHandle` personnalise l'instance qui est exposée au composant parent lors de l'utilisation de `ref`. Comme toujours, il vaut mieux s'abstenir d'utiliser du code impératif manipulant des refs dans la plupart des cas. `useImperativeHandle` est conçu pour être utilisé en conjonction avec `forwardRef` :

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

Dans cet exemple, un composant parent qui utiliserait `<FancyInput ref={inputRef} />` pourrait appeler `inputRef.current.focus()`.

`useLayoutEffect` {#uselayouтеffect}

La signature est identique à celle de `useEffect`, mais `useLayoutEffect` s'exécute de manière synchrone après que toutes les mutations du DOM ont eu lieu. Utilisez-le pour inspecter la mise en page du DOM et effectuer un nouveau rendu de manière synchrone. Les mises à jour planifiées dans `useLayoutEffect` seront traitées de manière synchrone avant que le navigateur ait pu procéder à l'affichage.

Préférez l'utilisation du `useEffect` standard chaque fois que possible, pour éviter de bloquer les mises à jour visuelles.

Astuce

Si vous migrez du code depuis un composant écrit à l'aide d'une classe, sachez que `useLayoutEffect` s'exécute dans la même phase que `componentDidMount` et `componentDidUpdate`. **Nous vous conseillons de commencer avec `useEffect`**, et de ne tenter `useLayoutEffect` que si vous rencontrez des problèmes.

Si vous faites du rendu côté serveur, n'oubliez pas que *ni `useLayoutEffect` ni `useEffect`* ne seront exécutés jusqu'à ce que votre code JS soit téléchargé et exécuté côté client. C'est pourquoi React vous averti quand un composant utilise `useLayoutEffect` dans le cadre d'un rendu côté serveur. Pour corriger ça, vous pouvez soit déplacer la logique dans `useEffect` (si elle n'est pas nécessaire pour le premier affichage), soit reporter l'affichage du composant jusqu'à ce que l'affichage côté client soit effectué (si le HTML aurait eu l'air cassé avant exécution du `useLayoutEffect`).

Pour exclure un composant nécessitant des effets de mise en page (*layout effects, NdT*) du HTML généré côté serveur, vous pouvez l'afficher conditionnellement avec un `showChild && <Child />`, et différer son affichage grâce à un `useEffect(() => { setShowChild(true); }, [])`. Ainsi, l'UI ne semblera pas cassé avec son hydratation.

`useDebugValue` {#usedebugvalue}

```
useDebugValue(value)
```

Vous pouvez utiliser `useDebugValue` pour afficher une étiquette pour les Hooks personnalisés dans les outils de développement React (*React DevTools, NdT*).

Par exemple, prenez le hook personnalisé `useFriendStatus` décrit dans « [Construire vos propres Hooks](#) » :

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  // Affiche une étiquette dans les DevTools à côté de ce Hook
  // par exemple, "FriendStatus: En ligne"
```

```
useDebugValue(isOnline ? 'En ligne' : 'Hors-ligne');

return isOnline;
}
```

Astuce

Nous déconseillons d'ajouter ces étiquettes à chaque Hook personnalisé. C'est surtout utile pour les Hooks personnalisés provenant de bibliothèques partagées.

Différer le formatage des valeurs de débogage {#defer-formatting-debug-values}

Formater une valeur à afficher peut parfois s'avérer coûteux. C'est par ailleurs inutile tant que le Hook n'est pas effectivement inspecté.

C'est pourquoi `useDebugValue` accepte une fonction de formatage comme deuxième argument optionnel. Cette fonction est appelée uniquement si les Hooks sont inspectés. Elle reçoit la valeur de débogage comme argument et devrait renvoyer la valeur formatée.

Par exemple, un Hook personnalisé qui renvoie une valeur `Date` pourrait éviter d'appeler inutilement la fonction `toString` en fournissant le formateur suivant :

```
useDebugValue(date, date => date.toString());
```



HOC

Un composant d'ordre supérieur (*Higher-Order Component* ou *HOC*, NdT) est une technique avancée de React qui permet de réutiliser la logique de composants. Les HOC ne font pas partie de l'API de React à proprement parler, mais découlent de sa nature compositionnelle.

Concrètement, **un composant d'ordre supérieur est une fonction qui accepte un composant et renvoie un nouveau composant.**

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Là où un composant transforme des props en interface utilisateur (UI), un composant d'ordre supérieur transforme un composant en un autre composant.

Les HOC sont courants dans des bibliothèques tierces de React, comme `connect` dans Redux et `createFragmentContainer` dans Relay.

Dans ce guide, nous verrons pourquoi les composants d'ordre supérieurs sont utiles, et comment créer le vôtre.

Utiliser les HOC pour les questions transversales {#use-hocs-for-cross-cutting-concerns}

Remarque

Auparavant, nous recommandions d'employer des *mixins* pour gérer les questions transversales.

Depuis, nous nous sommes rendus compte que les *mixins* créent plus de problèmes qu'ils n'en résolvent. Vous pouvez [lire le détail](#) des raisons qui nous ont fait renoncer aux *mixins*, et de la façon dont vous pouvez faire de même pour vos composants existants.

Les composants sont le principal moyen de réutiliser du code en React. Cependant, vous remarquerez que les composants classiques ne conviennent pas à tous les modèles.

Imaginez que vous ayez créé un composant `CommentList` qui s'abonne à une source externe de données pour afficher une liste de commentaires :

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // `DataSource` est une source de données quelconque
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // On s'abonne aux modifications
    DataSource.addChangeListener(this.handleChange);
  }
}
```

```

componentWillUnmount() {
  // On se désabonne
  DataSource.removeChangeListener(this.handleChange);
}

handleChange() {
  // Met à jour l'état local quand la source de données est modifiée
  this.setState({
    comments: DataSource.getComments()
  });
}

render() {
  return (
    <div>
      {this.state.comments.map((comment) => (
        <Comment comment={comment} key={comment.id} />
      ))}
    </div>
  );
}
}

```

Plus tard, vous créez un composant `BlogPost` qui s'abonne à un unique article, et dont la structure est similaire :

```

class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {

```

```

        return <TextBlock text={this.state.blogPost} />;
    }
}

```

`CommentList` et `BlogPost` ne sont pas identiques : ils appellent des méthodes différentes sur `DataSource`, et ont des affichages distincts. Pourtant une grande partie de leur implémentation est la même :

- Au montage (*quand le composant entre dans la couche d'affichage, NdT*), ils ajoutent un écouteur d'événements à `DataSource`.
- Dans l'écouteur, ils appellent `setState` quand la source de données est modifiée.
- Au démontage (*quand le composant sort de la couche d'affichage, NdT*), ils enlèvent l'écouteur d'événements.

Vous imaginez bien que dans une appli importante, ce motif d'abonnement à une `DataSource` et d'appel à `setState` sera récurrent. Il nous faut une abstraction qui nous permette de définir cette logique en un seul endroit et de la réutiliser pour de nombreux composants. C'est là que les composants d'ordre supérieur sont particulièrement utiles.

Nous pouvons écrire une fonction qui crée des composants qui s'abonnent à une `DataSource`, comme `CommentList` et `BlogPost`. La fonction acceptera parmi ses arguments un composant initial, qui recevra les données suivies en props. Appelons cette fonction `withSubscription` :

```

const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);

```

Le premier paramètre est le composant initial. Le second charge les données qui nous intéressent, en fonction de la `DataSource` et des props existantes.

Lorsque `CommentListWithSubscription` et `BlogPostWithSubscription` s'affichent, `CommentList` et `BlogPost` reçoivent une prop `data` qui contient les données les plus récentes issues de la `DataSource` :

```

// Cette fonction accepte un composant...
function withSubscription(WrappedComponent, selectData) {
  // ... et renvoie un autre composant...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }
  };
}

```

```

}

componentDidMount() {
  // ... qui s'occupe de l'abonnement...
  DataSource.addChangeListener(this.handleChange);
}

componentWillUnmount() {
  DataSource.removeChangeListener(this.handleChange);
}

handleChange() {
  this.setState({
    data: selectData(DataSource, this.props)
  });
}

render() {
  // ... et affiche le composant enrobé avec les données à jour !
  // Remarquez qu'on passe aussi toute autre prop reçue.
  return <WrappedComponent data={this.state.data} {...this.props} />;
}
};

}

```

Remarquez qu'un HOC ne modifie pas le composant qu'on lui passe, et ne recourt pas non plus à l'héritage pour copier son comportement. Un HOC *compose* le composant initial en l'*enrobant* dans un composant conteneur. Il s'agit d'une fonction pure, sans effets de bord.

Et voilà ! Le composant enrobé reçoit toutes les props du conteneur ainsi qu'une nouvelle prop, `data`, qu'il emploie pour produire son résultat. Le HOC ne se préoccupe pas de savoir comment ou pourquoi les données sont utilisées, et le composant enrobé ne se préoccupe pas de savoir d'où viennent les données.

Puisque `withSubscription` est juste une fonction, vous pouvez lui définir autant ou aussi peu de paramètres que vous le souhaitez. Par exemple, vous pourriez rendre configurable le nom de la prop `data`, afin d'isoler encore davantage le HOC et le composant enrobé. Ou alors, vous pourriez accepter un argument qui configure `shouldComponentUpdate`, ou un autre qui configure la source de données. Tout ça est possible parce que le HOC a un contrôle total sur la façon dont le composant est défini.

Comme pour les composants, le rapport entre `withSubscription` et le composant enrobé se base entièrement sur les props. Ça facilite l'échange d'un HOC pour un autre, du moment qu'ils fournissent les mêmes props au composant enrobé. Ça peut s'avérer utile si vous changez de bibliothèque pour charger vos données, par exemple.

Ne modifiez pas le composant initial : composez-le. {#dont-mutate-the-original-component-use-composition}

Résistez à la tentation de modifier le prototype d'un composant (ou de le modifier de quelque façon que ce soit) dans un HOC.

```

function logProps(InputComponent) {
  InputComponent.prototype.componentDidUpdate = function(prevProps) {
    console.log('Props actuelles : ', this.props);
    console.log('Props précédentes : ', prevProps);
  };
  // Le fait que le composant initial soit renvoyé est un signe qu'il a été
  modifié.
  return InputComponent;
}

// EnhancedComponent fera un log à chaque fois qu'il reçoit des props.
const EnhancedComponent = logProps(InputComponent);

```

Ce genre d'approche pose quelques problèmes. Pour commencer, le composant initial ne peut pas être réutilisé indépendamment du composant amélioré. Plus important encore, si vous appliquez un autre HOC sur `EnhancedComponent` qui modifie *aussi* `componentDidUpdate`, les fonctionnalités du premier HOC seront perdues ! Enfin, ce HOC ne fonctionnera pas avec des fonctions composants, qui n'ont pas de méthodes de cycle de vie.

Les HOC qui modifient le composant enrobé sont une abstraction foireuse : leurs utilisateurs doivent savoir comment ils sont implémentés afin d'éviter des conflits avec d'autres HOC.

Plutôt que la mutation, les HOC devraient utiliser la composition, en enrobant le composant initial dans un composant conteneur.

```

function logProps(WrappedComponent) {
  return class extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('Props actuelles : ', this.props);
      console.log('Props précédentes : ', prevProps);
    }
    render() {
      // Enrobe le composant initial dans un conteneur, sans le modifier. Mieux !
      return <WrappedComponent {...this.props} />;
    }
  }
}

```

Ce HOC a la même fonctionnalité que la version modifiante, tout en évitant le risque de conflits. Il fonctionne tout aussi bien avec les composants à base de classe et les fonctions composants. Et puisqu'il s'agit d'une fonction pure, il est composable avec d'autres HOC voire même avec lui-même.

Vous avez peut-être remarqué des ressemblances entre les HOC et le motif des **composants conteneurs**. Les composants conteneurs participent à des stratégies de séparation de responsabilités entre les préoccupations de haut et de bas niveau. Les conteneurs se préoccupent par exemple des abonnements et de l'état, et passent des props à d'autres composants qui se préoccupent par exemple d'afficher l'UI. Les HOC utilisent des conteneurs dans leur implémentation. Vous pouvez voir les HOC comme des définitions paramétrables de composants conteneurs.

Convention : transmettez les props annexes au composant enrobé {#convention-pass-unrelated-props-through-to-the-wrapped-component}

Les HOC ajoutent des fonctionnalités à un composant. Ils ne devraient pas drastiquement modifier son contrat. On s'attend à ce que le composant renvoyé par un HOC ait une interface semblable au composant initial.

Les HOC devraient transmettre les props sans rapport avec leur propre fonctionnement. La plupart des HOC ont une méthode de rendu qui ressemble à ça :

```
render() {
  // Filtre les props supplémentaires propres à ce HOC
  // qui ne devraient pas être transmises
  const { extraProp, ...passThroughProps } = this.props;

  // Injecte les props dans le composant enrobé. Il s'agit en général
  // de valeurs de l'état local ou de méthodes d'instance.
  const injectedProp = someStateOrInstanceMethod;

  // Transmet les props au composant enrobé
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}
```

Cette convention améliore la flexibilité et la réutilisabilité de nos HOC.

Convention : maximisez la composabilité {#convention-maximizing-composability}

Tous les HOC n'ont pas la même interface. Ils n'acceptent parfois qu'un seul argument, le composant enrobé :

```
const NavbarWithRouter = withRouter(Navbar);
```

Mais en général, les HOC acceptent des arguments supplémentaires. Dans cet exemple tiré de Relay, un objet de configuration `config` est transmis pour spécifier les dépendances d'un composant aux données :

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

La signature la plus courante pour un HOC ressemble à ceci :

```
// `connect` de React Redux
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

Pardon?! Il est plus facile de voir ce qui se passe si on décortique l'appel.

```
// connect est une fonction qui renvoie une autre fonction
const enhance = connect(commentListSelector, commentListActions);
// La fonction renvoyée est un HOC, qui renvoie un composant connecté au store
// Redux
const ConnectedComment = enhance(CommentList);
```

Autrement dit, `connect` est une fonction d'ordre supérieur... qui renvoie un composant d'ordre supérieur !

Cette forme peut sembler déroutante ou superflue, pourtant elle a une propriété utile. Les HOC n'acceptant qu'un argument comme celui que renvoie la fonction `connect` ont une signature `Composant => Composant`. Les fonctions dont le type de données est le même en sortie qu'en entrée sont beaucoup plus faciles à composer.

```
// Plutôt que de faire ceci...
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent))

// ... vous pouvez utiliser un utilitaire de composition de fonction.
// compose(f, g, h) est l'équivalent de (...args) => f(g(h(...args)))
const enhance = compose(
  // Ces deux-là sont des HOC n'acceptant qu'un argument.
  withRouter,
  connect(commentSelector)
)
const EnhancedComponent = enhance(WrappedComponent)
```

(C'est aussi cette propriété qui permet à `connect` et à d'autres HOC du même type d'être utilisés comme décorateurs, une proposition expérimentale JavaScript.)

La fonction utilitaire `compose` est fournie par de nombreuses bibliothèques tierces, dont `lodash` (sous le nom `lodash.flowRight`), `Redux`, et `Ramda`.

Convention : enrobez le `displayName` pour faciliter le débogage {#convention-wrap-the-display-name-for-easy-debugging}

Tout comme n'importe quel autre composant, les composants conteneurs créés par des HOC apparaissent dans les [Outils de développement React](#). Pour faciliter votre débogage, donnez-leur un nom affichable qui indique qu'ils sont le résultat d'un HOC.

La technique la plus répandue consiste à enrober le nom d'affichage du composant enrobé. Par exemple, si votre composant d'ordre supérieur s'appelle `withSubscription`, et que le nom d'affichage du composant enrobé est `CommentList`, utilisez le nom d'affichage `WithSubscription(CommentList)` :

```

function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName =
`WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}

```

Limitations {#caveats}

L'utilisation de composants d'ordre supérieur est sujette à quelques limitations qui ne sont pas tout de suite évidentes si vous débutez avec React.

Pas de HOC à l'intérieur de la méthode de rendu {#dont-use-hocs-inside-the-render-method}

L'algorithme de comparaison de React (qu'on appelle la réconciliation) utilise l'identité des composants pour déterminer s'il faut mettre à jour l'arborescence existante ou la jeter et en monter une nouvelle. Si le composant renvoyé par `render` est identique (`==`) au composant du rendu précédent, React met récursivement à jour l'arborescence en la comparant avec la nouvelle. S'ils ne sont pas identiques, l'ancienne arborescence est intégralement démontée.

En général, vous ne devriez pas avoir à y penser. Mais dans le cadre des HOC c'est important, puisque ça signifie que vous ne pouvez pas appliquer un HOC au sein de la méthode de rendu d'un composant :

```

render() {
  // Une nouvelle version de EnhancedComponent est créée à chaque rendu
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // Ça entraîne un démontage/remontage complet à chaque fois !
  return <EnhancedComponent />;
}

```

Il ne s'agit pas uniquement d'un problème de performances : remonter un composant signifie que l'état local de ce composant ainsi que ceux de tous ses enfants seront perdus.

Appliquez plutôt les HOC à l'extérieur de la définition d'un composant, afin de créer le composant enveloppé une seule fois. Son identité sera alors constante d'un rendu à l'autre. C'est généralement ce que vous voulez, de toutes façons.

Dans les rares cas où vous avez besoin d'appliquer un HOC de façon dynamique, vous pouvez le faire au sein des méthodes de cycle de vie d'un composant ou dans son constructeur.

Les méthodes statiques doivent être copiées {#static-methods-must-be-copied-over}

Il est parfois utile de définir une méthode statique dans un composant React. Par exemple, les conteneurs Relay exposent une méthode statique `getFragment` pour simplifier la composition de fragments GraphQL.

Cependant, quand vous appliquez un HOC à un composant, le composant initial est enrobé par un composant conteneur. Ça signifie que le nouveau composant ne comporte aucune des méthodes statiques du composant initial.

```
// Définit une méthode statique
WrappedComponent.staticMethod = function() {/*...*/}
// Applique un HOC
const EnhancedComponent = enhance(WrappedComponent);

// Le composant amélioré n'a pas de méthode statique
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

Pour résoudre ça, vous pouvez copier les méthodes dans le conteneur avant de le renvoyer :

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Doit savoir exactement quelles méthodes recopier :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

Le problème, c'est que ça exige que vous sachiez exactement quelles méthodes doivent être recopiées. Vous devriez plutôt utiliser `hoist-non-react-statics` pour copier automatiquement toutes les méthodes statiques qui ne viennent pas de React :

```
import hoistNonReactStatics from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatics(Enhance, WrappedComponent);
  return Enhance;
}
```

Une autre solution consiste à exporter les méthodes statiques de façon séparée du composant lui-même.

```
// Plutôt que...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ... exportez les méthodes séparément...
export { someFunction };
```

```
// ... et dans le module qui les utilise, importez les deux.  
import MyComponent, { someFunction } from './MyComponent.js';
```

Les refs ne sont pas transmises {#refs-arent-passed-through}

Bien que la convention pour les composants d'ordre supérieur soit de transmettre toutes les props au composant enrobé, ça ne marche pas pour les refs. C'est parce que `ref` n'est pas vraiment une prop : comme `key`, React la traite de façon particulière. Si vous ajoutez une ref à un élément dont le composant résulte d'un HOC, la ref fait référence à une instance du composant conteneur extérieur, et non au composant enrobé.

La solution à ce problème réside dans l'utilisation de l'API `React.forwardRef` (introduite dans React 16.3). [Vous pouvez en apprendre davantage dans la section sur la transmission des refs.](#)



Context

Le Contexte offre un moyen de faire passer des données à travers l'arborescence du composant sans avoir à passer manuellement les props à chaque niveau.

Dans une application React typique, les données sont passées de haut en bas (du parent à l'enfant) via les props, mais cela peut devenir lourd pour certains types de props (ex. les préférences régionales, le thème de l'interface utilisateur) qui s'avèrent nécessaires pour de nombreux composants au sein d'une application. Le Contexte offre un moyen de partager des valeurs comme celles-ci entre des composants sans avoir à explicitement passer une prop à chaque niveau de l'arborescence.

- [Quand utiliser le Contexte](#)
- [Avant d'utiliser le Contexte](#)
- [API](#)
 - [React.createContext](#)
 - [Context.Provider](#)
 - [Class.contextType](#)
 - [Context.Consumer](#)
 - [Context.displayName](#)
- [Exemples](#)
 - [Contexte dynamique](#)
 - [Mettre à jour le Contexte à partir d'un composant imbriqué](#)
 - [Consommer plusieurs Contextes](#)
- [Limitations](#)
- [API historique](#)

Quand utiliser le Contexte {#when-to-use-context}

Le Contexte est conçu pour partager des données qui peuvent être considérées comme « globales » pour une arborescence de composants React, comme l'utilisateur actuellement authentifié, le thème, ou la préférence de langue. Par exemple, dans le code ci-dessous nous faisons passer manuellement la prop `theme` afin de styler le composant `Button` :

`embed:context/motivation-problem.js`

En utilisant le Contexte, nous pouvons éviter de passer les props à travers des éléments intermédiaires :

`embed:context/motivation-solution.js`

Avant d'utiliser le Contexte {#before-you-use-context}

Le Contexte est principalement utilisé quand certaines données doivent être accessibles par de *nombreux* composants à différents niveaux d'imbrication. Utilisez-le avec parcimonie car il rend la réutilisation des composants plus difficile.

Si vous voulez seulement éviter de passer certaines props à travers de nombreux niveaux, la composition des composants est souvent plus simple que le contexte.

Par exemple, prenez un composant `Page` qui passe des props `user` et `avatarSize` plusieurs niveaux plus bas pour que les composants profondément imbriqués `Link` et `Avatar` puissent les lire :

```

<Page user={user} avatarSize={avatarSize} />
// ... qui affiche ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... qui affiche ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... qui affiche ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>

```

Ça peut paraître redondant de passer les props `user` et `avatarSize` à travers plusieurs niveaux, si au final seul le composant `Avatar` en a réellement besoin. Il est également pénible qu'à chaque fois que le composant `Avatar` a besoin de davantage de props d'en haut, vous ayez à les ajouter à tous les niveaux.

Un des moyens de résoudre ce problème **sans le contexte** consisterait à [transmettre le composant `Avatar` lui-même](#) de façon à ce que les composants intermédiaires n'aient pas besoin de connaître les props `user` ou `avatarSize`:

```

function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// À présent nous avons :
<Page user={user} avatarSize={avatarSize} />
// ... qui affiche ...
<PageLayout userLink={...} />
// ... qui affiche ...
<NavigationBar userLink={...} />
// ... qui affiche ...
{props.userLink}

```

Avec cette modification, seulement le composant le plus haut placé, `Page`, a besoin de connaître l'utilisation de `user` et `avatarSize` par les composants `Link` et `Avatar`.

Cette *inversion de contrôle* peut rendre votre code plus propre dans de nombreux cas en réduisant le nombre de props que vous avez besoin de passer à travers votre application et vous donne plus de contrôle sur les composants racines. Cependant, ce n'est pas toujours la bonne approche : déplacer la complexité vers le haut de l'arborescence rend les composants des niveaux supérieurs plus compliqués et force les composants de plus bas niveau à être plus flexibles que vous pourriez le souhaiter.

Vous n'êtes pas limité·e à un unique enfant pour un composant. Vous pouvez passer plusieurs enfants, ou même prévoir dans votre JSX plusieurs emplacements séparés pour les enfants [comme documenté ici](#) :

```

function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
      </Link>
    </NavigationBar>
  );
  return (
    <PageLayout
      topBar={topBar}
      content={content}
    />
  );
}

```

Ce motif est suffisant pour les nombreux cas où vous avez besoin de découpler un enfant de ses parents directs. Vous pouvez aller encore plus loin avec les [props de rendu](#) si l'enfant a besoin de communiquer avec le parent avant de s'afficher.

Cependant, parfois les mêmes données ont besoin d'être accessibles par de nombreux composants dans l'arborescence, et à différents niveaux d'imbrication. Le Contexte vous permet de « diffuser » ces données, et leurs mises à jour, à tous les composants plus bas dans l'arbre. Les exemples courants où l'utilisation du Contexte apporte une simplification incluent la gestion des préférences régionales, du thème ou d'un cache de données.

API {#api}

React.createContext {#reactcreatecontext}

```
const MyContext = React.createContext(defaultValue);
```

Crée un objet Context. Lorsque React affiche un composant qui s'abonne à cet objet [Context](#), il lira la valeur actuelle du contexte depuis le [Provider](#) le plus proche situé plus haut dans l'arborescence.

L'argument [defaultValue](#) est **uniquement** utilisé lorsqu'un composant n'a pas de [Provider](#) correspondant au-dessus de lui dans l'arborescence. Ça peut être utile pour tester des composants de manière isolée sans les enrober. Remarquez que passer [undefined](#) comme valeur au [Provider](#) n'aboutit pas à ce que les composants consommateurs utilisent [defaultValue](#).

Context.Provider {#contextprovider}

```
<MyContext.Provider value{/* une valeur */}>
```

Chaque objet Contexte est livré avec un composant React `Provider` qui permet aux composants consommateurs de s'abonner aux mises à jour du contexte.

Il accepte une prop `value` à transmettre aux composants consommateurs descendants de ce `Provider`(plus bas dans l'arbre, donc). Un `Provider` peut être connecté à plusieurs consommateurs. Les `Provider` peuvent être imbriqués pour remplacer leur valeur plus profondément dans l'arbre.

Tous les consommateurs qui sont descendants d'un `Provider` se rafraîchiront lorsque la prop `value` du `Provider` change. La propagation du `Provider` vers ses consommateurs descendants (y compris `.contextType` et `useContext`) n'est pas assujettie à la méthode `shouldComponentUpdate`, de sorte que le consommateur est mis à jour même lorsqu'un composant ancêtre saute sa mise à jour.

On détermine si modification il y a en comparant les nouvelles et les anciennes valeurs avec le même algorithme que `Object.is`.

Remarque

La manière dont les modifications sont déterminées peut provoquer des problèmes lorsqu'on passe des objets dans `value` : voir les [limitations](#).

`Class.contextType` {#classcontexttype}

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* produit un effet de bord au montage sur la valeur de MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* affiche quelque chose basé sur la valeur de MyContext */
  }
}
MyClass.contextType = MyContext;
```

La propriété `contextType` d'une classe peut recevoir un objet Contexte créé par `React.createContext()`. Ça vous permet de consommer la valeur la plus proche de ce Contexte en utilisant `this.context`. Vous pouvez la référencer dans toutes les méthodes de cycle de vie, y compris la fonction de rendu.

Remarque

Vous pouvez vous abonner à un unique contexte en utilisant cette API. Si vous voulez lire plus d'un contexte, voyez [Consommer plusieurs contextes](#).

Si vous utilisez la [syntaxe expérimentale des champs publics de classe](#), vous pouvez utiliser un champ **statique** de classe pour initialiser votre `contextType`.

```
class MyClass extends React.Component {  
  static contextType = MyContext;  
  render() {  
    let value = this.context;  
    /* Affiche quelque chose basé sur la valeur */  
  }  
}
```

Context.Consumer {#contextconsumer}

```
<MyContext.Consumer>  
  {value => /* affiche quelque chose basé sur la valeur du contexte */}  
</MyContext.Consumer>
```

Un composant React qui s'abonne aux modifications de contexte. Ça permet de s'abonner à un contexte au sein d'une [fonction composant](#).

Nécessite une [fonction enfant](#). La fonction reçoit le contexte actuel et renvoie un nœud React. L'argument `value` envoyé à la fonction sera égal à la prop `value` du [Provider](#) le plus proche (plus haut dans l'arbre) pour le contexte en question. Si il n'y pas de [Provider](#) pour le contexte voulu, l'argument `value` sera égal à la `defaultValue` passée lors de son `createContext()`.

Remarque

Pour en apprendre davantage sur l'approche « fonction enfant », voyez les [props de rendu](#).

Context.displayName {#contextdisplayname}

Les objets Contexte permettent une propriété textuelle `displayName`. Les Outils de développement React l'utilisent pour déterminer comment afficher le contexte.

Par exemple, le composant ci-après apparaîtra dans les Outils de développement en tant que `MyDisplayName` :

```
const MyContext = React.createContext(/* une valeur */);  
MyContext.displayName = 'MyDisplayName';  
  
<MyContext.Provider> // "MyDisplayName.Provider" dans les DevTools  
<MyContext.Consumer> // "MyDisplayName.Consumer" dans les DevTools
```

Exemples {#examples}

Contexte dynamique {#dynamic-context}

Un exemple plus complexe avec des valeurs dynamiques pour le thème :

theme-context.js [embed:context/theme-detailed-theme-context.js](#)

themed-button.js [embed:context/theme-detailed-themed-button.js](#)

app.js [embed:context/theme-detailed-app.js](#)

Mettre à jour le Contexte à partir d'un composant imbriqué {#updating-context-from-a-nested-component}

Il est souvent nécessaire de mettre à jour le contexte à partir d'un composant imbriqué profondément dans l'arbre des composants. Dans un tel cas, vous pouvez passer une fonction à travers le contexte qui permet aux consommateurs de le mettre à jour :

theme-context.js [embed:context/updating-nested-context-context.js](#)

theme-toggler-button.js [embed:context/updating-nested-context-theme-toggler-button.js](#)

app.js [embed:context/updating-nested-context-app.js](#)

Consommer plusieurs Contextes {#consuming-multiple-contexts}

Pour conserver un rafraîchissement rapide du contexte, React a besoin que chaque consommateur de contexte soit un nœud à part dans l'arborescence.

[embed:context/multiple-contexts.js](#)

Si plusieurs valeurs de contexte sont souvent utilisées ensemble, vous voudrez peut-être créer votre propre composant avec prop de rendu qui fournira les deux.

Limitations {#caveats}

Dans la mesure où le contexte utilise une identité référentielle pour déterminer quand se rafraîchir, il y a des cas piégeux qui peuvent déclencher des rafraîchissements involontaires pour les consommateurs lorsque le parent d'un fournisseur se rafraîchit. Par exemple, le code ci-dessous va rafraîchir chaque consommateur, le **Provider** se rafraîchissant lui-même parce qu'un nouvel objet est créé à chaque fois pour **value** :

[embed:context/reference-caveats-problem.js](#)

Pour contourner ce problème, placez la valeur dans l'état du parent :

[embed:context/reference-caveats-solution.js](#)

API historique {#legacy-api}

Remarque

React fournissait auparavant une API de contextes expérimentale. L'ancienne API restera prise en charge par toutes les versions 16.x, mais les applications qui l'utilisent devraient migrer vers la nouvelle version. L'API historique sera supprimée dans une future version majeure de React. Lisez la [documentation sur l'API historique de contexte ici](#).



Appliation State

Flux/Redux : présentation. Propagation de données.



Flux & Redux

Face au besoin continu d'ajout de nouvelles features, l'équipe frontend de Facebook s'est vite retrouvée bloquée à cause d'architectures Modèle-Vue-Contrôleur. Pour eux, le MVC n'est pas un pattern prévu pour scaler : la maintenabilité d'une application basée sur le design pattern MVC est laborieuse. Plus on ajoute de modèles, de contrôleurs et de vues, plus la complexité augmente.

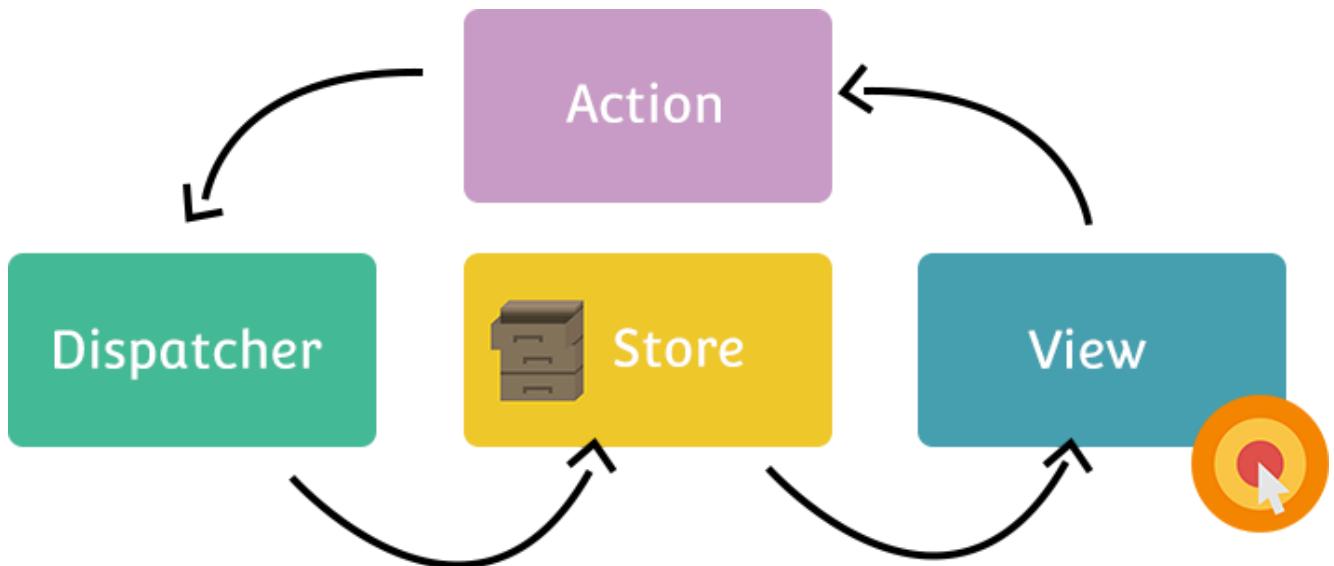
Pour surmonter cette difficulté, Facebook a créé une « nouvelle » architecture appelée **Flux**. Contrairement au MVC, celle-ci est une architecture garantissant (en théorie) un **flux unidirectionnel**, permettant à un développeur d'identifier rapidement le chemin critique d'un événement et ses conséquences.

Redux est une implémentation dérivée de **Flux**. Ça permet de créer un Store qui contient un état, réagit à des actions dispatchées, et auquel on peut souscrire pour être notifié des changements. Il permet également l'ajout de middlewares, qui peuvent en quelque sorte pré-process les actions.

Structurer les composants avec Redux

Flux s'appuie sur plusieurs concepts :

- action ;
- dispatcher ;
- store ;
- view (composants React).



Chaque interaction de l'utilisateur sur la `view` déclenche une `action`, celle-ci passe ensuite dans l'unique `dispatcher` qui notifie les `store`. **Le store prévient le composant qu'il a été modifié et celui-ci se met à jour.**

Un store gère l'ensemble de données et la logique métier d'un domaine de l'application. Le `dispatcher` est quant à lui le **seul point d'entrée des actions**, ce qui permet de garder la main sur le code flow et de prévenir d'éventuels effets de bord.

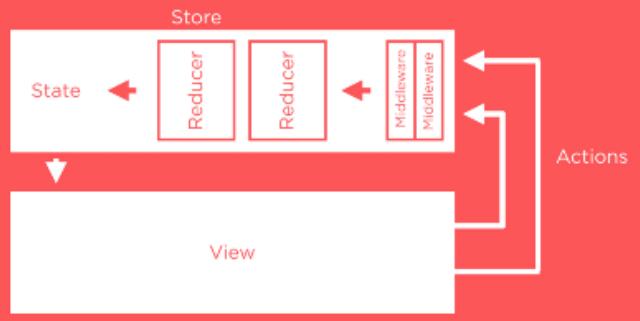
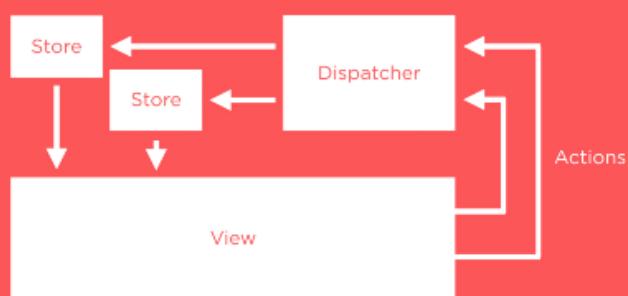
Comparaison des architectures.

Dans Redux, il n'y a qu'un seul `store` pour l'état de l'application. Ce `store` est mis à jour par des `action` utilisant des `reducer`.

Dans Flux il ya plusieurs `store` qui peuvent avoir des dépendances internes. Cette petite différence rend plus facile de raisonner sur l'état en utilisant Redux.

Flux

Redux



La principale idée de flux est de faire passer le moindre évènement de votre application au travers d'une boucle qui va parcourir tous vos stores (les éléments qui contiennent les données de votre application).

Le code de Flux vous ne se compose en fait que d'un dispatcher (et de quelques helpers). Flux n'est pas du code mais plutôt une nouvelle façon de penser son code.

Avec Redux votre codebase se décompose de la façon suivante :

- store : l'endroit où votre modèle va être contenu.
- action : représentant toutes les actions possibles.
- dispatcher : unique, qui notifie les stores des actions effectuées.
- view-controller : qui transforme et affiche les données transmises.

Les vues

L'idée principale du view-controller est de regrouper ensemble le contrôleur et sa vue associée.

Facebook distingue deux types de vues différentes dans Flux : **les views simples et les containers.**

Les containers

Les containers sont des view-controller un spéciaux : **ils écoutent les changements d'un store.**

L'idée du container est de centraliser les données relatives à une partie de l'application à un seul endroit.

Le container passera ensuite ces données à ses enfants pour affichage.

Pour reprendre le cas de notre application de TODOS on va avoir envie d'avoir un container qui a accès à l'utilisateur loggé et un autre qui s'occupe des TODOS. Ils écouteront respectivement les userStore et todosStore.

```

React.createClass({
  //...
  componentWillMount: function () {
    // à la création du composant, on enregistre le listener
    userStore.addListener(this.onChange);
  },
  componentWillUnmount: function () {
    // à la suppression du composant, on retire le listener
    userStore.removeListener(this.onChange);
  },
  // ce callback est appelé à chaque fois que le userStore change
  onChange: function () {
    this.setState({
      user: userStore.get()
    });
  },
  render: function () {
    // on a ici la dernière valeur du user
    return (<App user={this.state.user}>);
  }
});

```

Les view-controller

Les view-controller ne sont là que pour transformer les données brutes et les afficher.

Il reçoivent donc toutes leurs données via des propriétés et sont censés être complètement stateless.

On retrouve, avec les vues, un concept propre à la programmation fonctionnelle : une fonction retournera toujours le même résultat si on lui donne les mêmes paramètres. On appelle ça les fonctions pures.

D'ailleurs avec la nouvelle syntaxe de composant que React v0.14 a introduit récemment, vous pouvez déclarer un composant comme étant une simple fonction qui prend en paramètre des props et retourne du JSX. Avec cette syntaxe, pas de state et on est alors obligé de faire un composant stateless.

```
var App = function (props) {
  var user = props.user;
  return (
    <div className="header">
      <span>{user.firstname}</span>
      <span>{user.lastname}</span>
      <LogoutButton user={user} />
    </div>
    <TodoContainer />
  );
}
```

Rôle du “Dispatcher” dans Flux pour les actions.

Le `dispatcher` est là pour faire transiter absolument tout ce qu'il se passe sur l'application (événements) par les `store`.

Lors du bootstrap de l'application tous les stores devront donc être enregistrés auprès de ce dispatcher unique.

Les “Stores”, gestionnaire d’états logique dans Flux.

Les stores sont en fait une représentation complète de l'état, à un instant donné, de l'application. Il ne doit y avoir aucun de vos modèles qui vit en dehors d'un store.

Si vous vous pliez à cette règle, il vous suffira de faire une sauvegarde des données de vos stores et la recharger plus tard pour pouvoir retrouver l'application dans l'état exact dans laquelle vous l'aviez laissée.

Un `store` se comportent comme un modèle observable (il dispose de `getters` et d'une méthode `addListener`) à la différence près qu'il n'a pas de `setter`.

Seul le `store` peut mettre à jour ses propres données.

```

var events = {
    //quand L'application reçoit une liste de todos du serveur (ou d'ailleurs)
    RECEIVED_TODOS: "RECEIVED_TODOS",

    //quand L'utilisateur ajoute un nouveau todo
    TODO_ADDED: "TODO_ADDED",

    //quand Les todos ont été sauvegardés sur le serveur
    TODOS_SAVED: "TODOS_SAVED"
    // [...] plein d'autres évènements que nous ne traiterons pas ici
};

//L'instance unique de notre dispatcher de l'application
var appDispatcher = require("../dispatcher/appDispatcher"),
    //les évènements définis précédemment
    events = require("./events");

//nos todos. inaccessible depuis l'extérieur et vide pour le moment
//c'est une action qui viendra remplir tout ça
var todos = [];

var TodosStore = {
    //l'unique méthode accessible depuis l'extérieur
    //qui nous retourne simplement les todos
    get: function () {
        return todos;
    }

    //ps il faut rajouter ici les méthodes d'ajout/suppression
    //d'event listeners (addListener/removeListener)
    //qui serviront pour prévenir les vues que quelque chose
    //a changé
};

//c'est ici que la magie s'opère. On enregistre un callback qui sera appelé dès que qu
appDispatcher.register(function (payload) {
    //dans le payload on a tous les détails de l'évènement (type et données qui va ave
    var action = payload.actionType,
        data = payload.data;

    //on répond uniquement aux évènements qui nous intéressent
    switch (action) {
        //quand on reçoit les todos, on remplace simplement nos todos
        //par les todos reçus
        case events.RECEIVED_TODOS:
            todos = data.todos;
            this.emitChange(); //stay tuned, on parle de ça bientôt
            break;

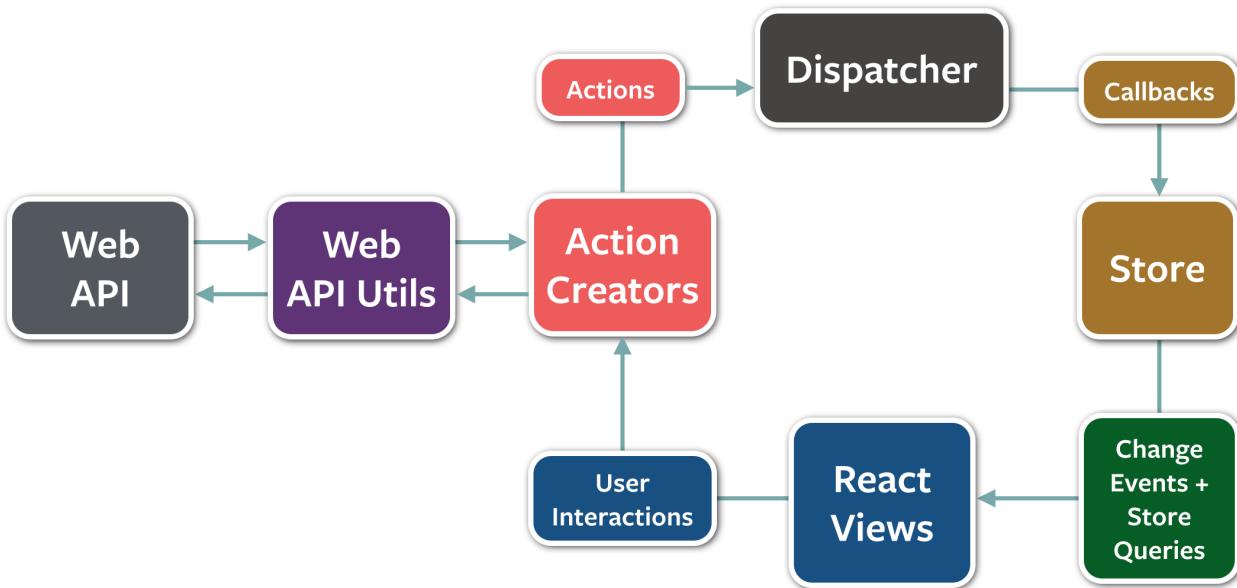
        //quand un todo est créé sur le serveur on l'ajoute dans notre
        //liste de todos
        case events.TODO_ADDED:
            todos.push(data.todo); //on ajoute la nouvelle todo dans la liste
    }
});

```

```
this.emitChange(); //stay tuned, on parle de ça bientôt
break;

//par défaut, on ne fait rien
//vous noterez par exemple qu'on ne répond pas ici
//à l'évènement TODOS_SAVED car il n'aurait aucune influence
//sur les données brute de ce store
default:
    break;
}
}.bind(todosStore));

module.exports = todosStore;
```



- **Les stores contiennent toutes les données brutes de l'application ;**
- Les stores représentent l'état de l'application à l'instant donné et sont donc complètement synchrone ;
- Seuls les stores peuvent modifier leurs données ;
- **Les stores réagissent aux évènements du dispatcher ;**
- Les stores sont comme un modèle observable mais sans setter ;
- Les vues écoutent les changements d'un (ou plusieurs) store(s).
- **Une vue qui est reliée à un store est appelée “container”.**
- Le dispatcher informe les stores.
- Les actions sont là pour appeler le dispatcher et éventuellement des services externes (API, localStorage ...) ;

Définition du Functional Programming.

La programmation fonctionnelle est un paradigme de programmation de type déclaratif qui considère le calcul en tant qu'évaluation de fonctions mathématiques.

Comme le changement d'état et la mutation des données ne peuvent pas être représentés par des évaluations de fonctions la programmation fonctionnelle ne les admet pas, au contraire elle met en avant l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état.

Approche avec Redux. Le “Reducer”.

Redux est un “state container” créé pour être totalement prévisible, et donc de pouvoir simplifier des tests et le debug. Pour cela, il se base sur une philosophie très stricte et rigide basé sur 3 principes :

- Les **données** dynamiques de l'application **existent en un seul endroit**.
- *Modifier les données de l'application ne peut se faire qu'à travers une action, et les actions ne peuvent faire autre chose que modifier les données de l'application.*
- Modifier les données crée une nouvelle “version” des données.

Ça permet de créer un Store qui contient un état, réagit à des actions dispatchées, et auquel on peut souscrire pour être notifié des changements.

Il permet également l'ajout de middlewares, qui peuvent en quelque sorte pre-process les actions.

Utilitaire Redux

Redux reprend les concepts de Flux mais en simplifiant beaucoup le processus de développement. Cette simplification est en partie due au fait que Redux utilise des concepts liés à la programmation fonctionnelle et s'inspire d'Elm pour changer l'état de l'application.

Redux en exemples

```

import { createStore } from 'redux'

/**
 * Le reducer est une fonction dite "pure" ayant (state, action) => state comme signa
 * Il va décrire comment une action transforme le state (l'état) de l'application
 * en un nouvel état.
 *
 * L'implémentation de l'état de l'application dépend totalement de votre y * use case
 * une structure de données
 * immutable (basé sur Immutable.js par exemple).
 * La seule chose à retenir est que cette partie ne DOIT PAS modifier
 * l'objet correspondant à l'état de l'application lorsque l'état change.
 * Dans cet exemple, on utilise un switch et des strings, mais on pourra * très bien
 */
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

// On crée un Redux store, qui va garder l'état de notre app.
// L'api correspond à trois fonctions { subscribe, dispatch, getState }.
let store = createStore(counter)

// On peut s'abonner manuellement ou bien lier l'état à une vue automatique// ment à
store.subscribe(() =>
  console.log(store.getState())
)

// Le seul moyen de modifier l'état de l'application est de dispatcher des actions.
// Les actions peuvent être serialisées, loggées ou sauvegardées pour plus tard.
store.dispatch({ type: 'INCREMENT' })
// 1
store.dispatch({ type: 'INCREMENT' })
// 2
store.dispatch({ type: 'DECREMENT' })
// 1

```

Plutôt que de modifier l'état de l'application directement, on spécifie les modifications qui peuvent arriver avec de simples objets appelés actions.

Puis on écrit une fonction appelée reducer, qui se chargera de décider comment chaque action transforme l'état de l'application.

DIFFÉRENCES AVEC FLUX ?

- Redux n'a pas de dispatcher
- Redux n'a pas la possibilité de définir plusieurs Stores.

A la place, nous avons un store unique avec un seul reducer . Au fur et à mesure que l'application se complexifie, l'unique reducer va être découpé en plusieurs petits reducers indépendants.

Extension pour ReactJS : “hot-loader”.

[React Hot Loader](#) permet la modification à chaud d'un composant React.

Le **hot-reload** est rendu possible par les fonctionnalités conjuguées de 3 composantes:

- **Webpack**, qui permet le remplacement à chaud de n'importe quel module CommonJS (HMR).
- **react-hot-loader**, qui intervient en cas de modification de composant React, il agit comme un proxy sur les méthodes du composant.
- **React** lui-même, qui propose une fonction de rendu pure : le render ne dépend que des propriétés en entrée (props / state). De fait, le remplacement de cette fonction par react-hot-loader permet à lui seul d'obtenir le nouveau rendu.

[React Hot Loader](#)



Mode Concurrent

Attention

Cette page décrit **des fonctionnalités expérimentales qui ne sont pas encore disponibles dans une version stable**. Ne vous basez pas sur les builds expérimentaux de React pour vos applis en production. Ces fonctionnalités sont susceptibles d'évoluer de façon significative et sans avertissement avant d'intégrer officiellement React.

Cette documentation est destinée aux personnes curieuses ou habituées à adopter les nouvelles technologies très tôt. **Si vous débutez en React, ne vous préoccupez pas de ces fonctionnalités :** vous n'avez pas besoin de les apprendre pour le moment.

Cette page fournit un aperçu théorique du mode concurrent. **Pour une introduction plus orientée vers la pratique, vous voudrez sans doute consulter les prochaines sections :**

- [Suspense pour le chargement de données](#) décrit un nouveau mécanisme de chargement de données distantes au sein de composants React.
- [Approches pour une UI concurrente](#) illustre quelques approches de conception d'UI rendues possibles par le mode concurrent et Suspense.
- [Adopter le mode concurrent](#) explique comment vous pouvez essayer le mode concurrent dans votre projet.
- [Référence de l'API du mode concurrent](#) documente les nouvelles API disponibles dans les builds expérimentaux de React.

Qu'est-ce que le mode concurrent ? {#what-is-concurrent-mode}

Le mode concurrent est un ensemble de nouvelles fonctionnalités qui aident les applis React à rester réactives et à s'adapter de façon fluide aux capacités et au débit réseau de l'appareil de l'utilisateur.

Ces fonctionnalités sont encore expérimentales et peuvent changer. Elles ne font pas encore partie d'une version stable de React, mais vous pouvez les essayer dès maintenant au moyen d'un build expérimental.

Rendu bloquant vs. interruptible {#blocking-vs-interruptible-rendering}

Pour expliquer le mode concurrent, nous allons utiliser la gestion de versions comme métaphore. Si vous travaillez en équipe, vous utilisez probablement un système de gestion de versions tel que Git, et travaillez sur des branches. Quand une branche est finalisée, vous pouvez en fusionner le travail dans `master` afin que d'autres puissent le récupérer.

Avant que la gestion de versions n'apparaisse, les flux de développement étaient très différents. On n'y trouvait aucun concept de branche. Si vous aviez besoin de modifier certains fichiers, il fallait prévenir tout le monde de ne pas y toucher pendant ce temps-là. Vous ne pouviez même pas commencer à travailler dessus en parallèle les uns des autres : vous étiez littéralement **bloqué·e** par l'autre personne.

Ce scénario illustre bien la façon dont les bibliothèques d'interface utilisateur (UI) fonctionnent généralement aujourd'hui. Une fois qu'elles démarrent le rendu d'une mise à jour, y compris la création de nouveaux nœuds DOM et l'exécution du code au sein des composants, elles ne peuvent pas être interrompues. Nous appelons cette approche le « rendu bloquant ».

Avec le mode concurrent, le rendu n'est pas bloquant : il est interruptible. Ça améliore l'expérience utilisateur, et en prime, ça ouvre la porte à de nouvelles fonctionnalités qui étaient impossibles jusque-là. Avant de nous pencher sur des exemples concrets dans les [prochaines sections](#), survolons rapidement ces nouvelles fonctionnalités.

Rendu interruptible {#interruptible-rendering}

Prenez une liste de produits filtrable. Vous est-il déjà arrivé de taper dans le champ de filtrage pour ressentir un affichage saccadé à chaque touche pressée ? Une partie du travail de mise à jour de la liste de produits est peut-être incontournable, telle que la création des nouveaux nœuds DOM ou la mise en page effectuée par le navigateur. En revanche, le *moment* d'exécution de ce travail et la *manière* dont il est exécuté jouent un rôle crucial.

Une manière courante de contourner ces saccades consiste à "*debounce*" la saisie dans le champ. En lissant ainsi le traitement de la saisie, nous ne mettons à jour la liste *qu'après* que l'utilisateur·rice a cessé de saisir sa valeur. Cependant, il peut être frustrant de ne constater aucune mise à jour de l'UI lors de la frappe. On pourrait plutôt « ralentir » (*throttle*, *NdT*) la gestion de la saisie, et ne mettre à jour la liste *qu'à hauteur* d'une fréquence maximale définie. Mais sur des appareils de faible puissance nous constaterions toujours une saccade. Tant le *debouncing* que le *throttling* aboutissent à une expérience utilisateur sous-optimale.

La raison de cette saccade est simple : une fois que le rendu commence, il ne peut être interrompu. Ainsi le navigateur ne peut plus mettre à jour le texte dans le champ de saisie juste après que vous avez pressé une touche. Peu importent les scores mirifiques que votre bibliothèque UI (telle que React) obtient dans tel ou tel comparatif, si elle recourt à un rendu bloquant, à partir d'une certaine charge de travail sur vos composants vous obtiendrez toujours un affichage saccadé. Et la plupart du temps, il n'existe pas de correctif simple.

Le mode concurrent corrige cette limitation fondamentale en utilisant un rendu interruptible. Ça signifie que lorsque l'utilisateur·rice presse une touche, React n'a pas besoin d'empêcher le navigateur de mettre à jour le champ de saisie. Il va plutôt laisser le navigateur afficher cette mise à jour, et continuer le rendu de la liste à jour *en mémoire*. Quand le rendu sera fini, React mettra à jour le DOM, et les modifications seront ainsi reflétées à l'écran.

Conceptuellement, vous pouvez imaginer que React prépare chaque mise à jour « sur une branche ». Tout comme vous êtes libre d'abandonner le travail d'une branche ou de passer d'une branche à l'autre, React en mode concurrent peut interrompre une mise à jour en cours afin de prioriser une tâche plus critique, puis revenir à ce qu'il était en train de faire. Cette technique n'est pas sans rappeler le [double buffering](#) des jeux vidéos.

Les techniques du mode concurrent réduisent le besoin de *debouncing* et de *throttling* dans l'UI. Le rendu étant interruptible, React n'a plus besoin de *différer* artificiellement du travail afin d'éviter les saccades. Il peut commencer le rendu immédiatement, et interrompre ce travail si nécessaire afin de préserver la réactivité de l'appli.

Séquences de chargement intentionnelles {#intentional-loading-sequences}

Nous disions tout à l'heure que pour comprendre le mode concurrent, on peut imaginer que React travaille « sur une branche ». Les branches ne sont pas seulement utiles pour des correctifs à court terme, mais aussi pour des fonctionnalités plus longues à écrire. Parfois vous pouvez travailler sur une fonctionnalité qui va

mettre des semaines avant d'être « assez finie » pour être fusionnée dans `master`. Ici aussi, la métaphore de la gestion de versions s'applique bien au rendu.

Imaginez que vous naviguez entre deux écrans d'une appli. Parfois, nous n'aurons peut-être pas assez de code et de données pour afficher un état de chargement « assez fini » à l'utilisateur au sein du nouvel écran. Transiter vers un écran vide (ou doté d'un gros *spinner*) n'est pas une expérience agréable. Et pourtant, il arrive fréquemment que les chargements du code et des données nécessaires ne prennent en fait que peu de temps. **Ne serait-il pas plus agréable que React puisse rester sur l'ancien écran un tout petit peu plus longtemps, pour ensuite « sauter » l'état de « chargement désagréable » lors de la bascule vers le nouvel écran ?**

C'est certes possible aujourd'hui, mais au prix d'une orchestration délicate. Avec le mode concurrent, cette fonctionnalité est directement disponible. React commence à préparer le nouvel écran en mémoire d'abord—ou, pour revenir à notre métaphore, « sur une autre branche ». Ainsi, React peut attendre que davantage de contenu ait été chargé avant de mettre à jour le DOM. Avec le mode concurrent, nous pouvons dire à React de continuer à afficher l'ancien écran, pleinement interactif, avec peut-être un indicateur de chargement dans un coin. Et lorsque le nouvel écran est prêt, React peut nous y amener.

Concurrence {#concurrency}

Résumons les deux exemples ci-avant pour voir comment le mode concurrent en unifie le traitement. **Avec le mode concurrent, React peut travailler à plusieurs mises à jour de l'état en exécution concurrente**, tout comme les branches permettent à divers membres d'une équipe de travailler indépendamment les uns des autres :

- Pour les mises à jour dépendantes du processeur (CPU, telles que la création des noeuds DOM et l'exécution du code des composants), la concurrence permet à une mise à jour plus urgente « d'interrompre » le rendu qui a déjà démarré.
- Pour les mises à jour dépendantes des entrées/sorties (I/O, telles que le chargement de code ou de données à partir du réseau), la concurrence permet à React de commencer le rendu en mémoire avant même que les données n'arrivent, et de sauter des états de chargement désagréables.

Ce qui est critique, c'est que la façon dont vous *utilisez* React reste inchangée. Les concepts tels que les composants, les props et l'état local continuent fondamentalement à marcher de la même façon. Quand vous voulez mettre à jour l'écran, vous ajustez l'état.

React utilise des heuristiques pour déterminer le degré « d'urgence » d'une mise à jour, et vous permet d'ajuster ces choix au moyen de quelques lignes de code pour aboutir à l'expérience utilisateur que vous souhaitez suite à chaque interaction.

Bénéficier de la recherche dans la production {#putting-research-into-production}

Les fonctionnalités du mode concurrent ont toutes le même objectif. **Leur mission consiste à faire bénéficier de véritables UI des dernières trouvailles de la recherche en Interactions Humain-Machine.**

Par exemple, la recherche montre qu'afficher trop d'états de chargement intermédiaires lors d'une transition entre écrans entraîne un sentiment accru de *lenteur*. C'est pourquoi le mode concurrent n'affiche de nouveaux

états de chargement que selon un « planning » fixe afin d'éviter des mises à jour trop fréquentes ou désagréables.

Dans le même esprit, la recherche nous dit que des interactions telles que le survol du pointeur ou la saisie de texte doivent être traitées dans un très court laps de temps, alors que les clics et les transitions de pages peuvent durer un peu plus longtemps sans pour autant sembler lentes. Les différentes « priorités » que le mode concurrent utilise en interne correspondent à peu près aux catégories d'interactions qu'on peut trouver dans la recherche en perception humaine.

Les équipes accordant une importance primordiale à l'expérience utilisateur résolvent parfois ce type de problème avec une solution *ad hoc*. Néanmoins, ces solutions survivent rarement à l'épreuve du temps, et sont difficiles à maintenir. Avec le mode concurrent, nous tentons de condenser les résultats de la recherche UI directement dans l'abstraction proposée par React, et d'en rendre l'utilisation idiomatique. React, en tant que bibliothèque UI, est bien placée pour ça.

Prochaines étapes {#next-steps}

Vous savez désormais à quoi sert le mode concurrent !

Dans les prochaines pages, vous en apprendrez davantage sur des sujets plus spécifiques :

- [Suspense pour le chargement de données](#) décrit un nouveau mécanisme de chargement de données distantes au sein de composants React.
- [Approches pour une UI concurrente](#) illustre quelques approches de conception d'UI rendues possibles par le mode concurrent et Suspense.
- [Adopter le mode concurrent](#) explique comment vous pouvez essayer le mode concurrent dans votre projet.
- [Référence de l'API du mode concurrent](#) documente les nouvelles API disponibles dans les builds expérimentaux de React.

Attention

Cette page décrit **des fonctionnalités expérimentales qui ne sont pas encore disponibles dans une version stable**. Ne vous basez pas sur les builds expérimentaux de React pour vos applis en production. Ces fonctionnalités sont susceptibles d'évoluer de façon significative et sans avertissement avant d'intégrer officiellement React.

Cette documentation est destinée aux personnes curieuses ou habituées à adopter les nouvelles technologies très tôt. **Si vous débutez en React, ne vous préoccupez pas de ces fonctionnalités :** vous n'avez pas besoin de les apprendre pour le moment. Par exemple, si vous cherchez un tutoriel sur le chargement de données qui fonctionne dès maintenant, lisez plutôt [cet article](#).

D'habitude, quand nous mettons à jour l'état, nous nous attendons à ce que nos changements se manifestent immédiatement à l'écran. C'est logique, dans la mesure où nous voulons que notre appli réagisse rapidement aux saisies utilisateurs. Néanmoins, il existe des cas dans lesquels nous pourrions préférer **différer l'apparition d'une mise à jour à l'écran**.

Par exemple, si nous passons d'une page à une autre et que ni le code ni les données pour ce prochain écran ne sont encore chargés, on pourrait trouver frustrant de voir immédiatement s'afficher une page vierge avec un indicateur de chargement. Nous préférerions peut-être rester un peu plus longtemps sur l'écran précédent. Historiquement, implémenter cette approche en React n'était pas chose aisée. Le mode concurrent offre un nouveau jeu d'outils pour y arriver.

- [Transitions](#)
 - [Enrober `setState` dans une transition](#)
 - [Ajouter un indicateur d'attente](#)
 - [Le point sur les changements](#)
 - [Où survient la mise à jour ?](#)
 - [Les transitions sont partout](#)
 - [Intégrer les transitions au système de conception](#)
- [Les trois étapes](#)
 - [Par défaut : En retrait → Squelette → Terminé](#)
 - [Préférable : En attente → Squelette → Terminé](#)
 - [Enrobez les fonctionnalités paresseuses avec `<Suspense>`](#)
 - [Le « train » de révélations de `Suspense`](#)
 - [Différer un indicateur d'attente](#)
 - [En résumé](#)
- [Autres approches](#)
 - [Dissocier les états à forte et faible priorité](#)
 - [Différer une valeur](#)
 - [`SuspenseList`](#)
- [Prochaines étapes](#)

Transitions {#transitions}

Reprenez [cette démo](#) de la page précédente sur `Suspense` pour le chargement de données.

Lorsqu'on clique sur le bouton « Suivant » pour basculer le profil actif, les données de la page existante disparaissent immédiatement, et nous avons à nouveau un indicateur de chargement pour la page entière. On pourrait qualifier cet état de chargement « d'indésirable ». **Ce serait sympa si nous pouvions « sauter » cet état et attendre qu'un peu de contenu arrive avant de transiter vers le nouvel écran.**

React offre un nouveau Hook intégré `useTransition()` pour nous y aider.

On peut l'utiliser en trois temps.

Tout d'abord, nous devons nous assurer d'utiliser effectivement le mode concurrent. Nous en reparlerons plus tard dans [Adopter le mode concurrent](#), mais pour l'instant il suffit de vérifier qu'on utilise bien `ReactDOM.createRoot()` au lieu de `ReactDOM.render()` afin que ce mode fonctionne :

```
const rootElement = document.getElementById("root");
// Activation explicite du mode concurrent
ReactDOM.createRoot(rootElement).render(<App />);
```

Ensuite, nous ajouterons un import du Hook `useTransition` de React :

```
import React, { useState, useTransition, Suspense } from "react";
```

Enfin, nous l'utiliserons au sein de notre composant `App` :

```
function App() {
  const [resource, setResource] = useState(initialResource);
  const [startTransition, isPending] = useTransition({
    timeoutMs: 3000
  });
  // ...
```

Pour le moment, par lui-même, ce code ne fait rien. Nous allons devoir utiliser les valeurs renvoyées par ce Hook pour mettre en place notre transition d'état. Voici les deux valeurs que renvoie `useTransition` :

- `startTransition` est une fonction. Nous l'utiliserons pour indiquer à React *quelle* mise à jour d'état nous souhaitons différer.
- `isPending` est un booléen, grâce auquel React nous indique si nous sommes actuellement en train d'attendre la fin de la transition.

Nous allons les utiliser dans un instant.

Remarquez que nous passons un objet de configuration à `useTransition`. Sa propriété `timeoutMs` indique **combien de temps nous acceptons d'attendre que la transition se termine**. En passant `{ timeoutMs: 3000 }`, nous disons « si le prochain profil prend plus de 3 secondes à charger, affiche le gros spinner—mais d'ici là, tu peux rester sur l'écran précédent ».

Enrober `setState` dans une transition {#wrapping-setstate-in-a-transition}

Notre gestionnaire de clic pour le bouton « Suivant » déclenche la bascule du profil courant dans notre état local :

```
<button
  onClick={() => {
    const nextUserId = getNextId(resource.userId);
    setResource(fetchProfileData(nextUserId));
  }}
>
```

Nous allons enrober cette mise à jour de l'état dans un appel à `startTransition`. C'est ainsi que nous indiquons à React que **ça ne nous dérange pas que React diffère cette mise à jour de l'état** si elle entraînait un état de chargement indésirable :

```
<button
  onClick={() => {
    startTransition(() => {
      const nextUserId = getNextId(resource.userId);
      setResource(fetchProfileData(nextUserId));
    });
  }}
>
```

Essayez sur [CodeSandbox](#)

Cliquez sur « Suivant » plusieurs fois. Remarquez comme une différence se fait déjà bien sentir. **Au lieu de voir immédiatement un écran vide suite au clic, nous continuons à voir l'écran précédent pendant un instant.** Une fois les données chargées, React transite sur le nouvel écran.

Si nous ajustons nos API pour mettre 5 secondes à répondre, [nous pouvons confirmer](#) que React décide alors « d'abandonner » en transitant vers le prochain écran au bout de 3 secondes. C'est dû à notre argument `{ timeoutMs: 3000 }` dans `useTransition()`. À titre d'exemple, si nous avions plutôt passé `{ timeoutMs: 60000 }`, il aurait attendu une minute entière.

Ajouter un indicateur d'attente {#adding-a-pending-indicator}

Il reste quelque chose qui semble cassé dans [notre dernier exemple](#). Bien sûr, c'est sympa de ne pas voir un « mauvais » état de chargement. **Mais n'avoir aucun indicateur de progression est quelque part encore pire !** Quand on clique sur « Suivant », rien ne se passe et on dirait que l'appli est cassée.

Notre appel à `useTransition()` renvoie deux valeurs : `startTransition` et `isPending`.

```
const [startTransition, isPending] = useTransition({ timeoutMs: 3000 });
```

Nous avons déjà utilisé `startTransition` pour enrober la mise à jour de l'état. Nous allons maintenant utiliser `isPending` en prime. React nous fournit ce booléen pour nous indiquer que **nous sommes en train**

d'attendre la fin d'une transition. Nous l'utiliserons pour indiquer que quelque chose se passe :

```
return (
  <>
  <button
    disabled={isPending}
    onClick={() => {
      startTransition(() => {
        const nextUserId = getNextId(resource.userId);
        setResource(fetchProfileData(nextUserId));
      });
    }}
  >
    Suivant
  </button>
  {isPending ? "Chargement..." : null}
  <ProfilePage resource={resource} />
</>
);
```

Essayez sur [CodeSandbox](#)

Voilà qui rend beaucoup mieux ! Quand nous cliquons sur « Suivant », le bouton est désactivé puisque cliquer dessus plusieurs fois n'aurait pas de sens. Et le nouveau texte « Chargement... » indique à l'utilisateur que l'appli n'a pas gelé.

Le point sur les changements {#reviewing-the-changes}

Revoyons l'ensemble des modifications apportées à notre [exemple d'origine](#) :

```
function App() {
  const [resource, setResource] = useState(initialResource);
  const [startTransition, isPending] = useTransition({
    timeoutMs: 3000
  });
  return (
    <>
    <button
      disabled={isPending}
      onClick={() => {
        startTransition(() => {
          const nextUserId = getNextId(resource.userId);
          setResource(fetchProfileData(nextUserId));
        });
      }}
    >
      Suivant
    </button>
    {isPending ? "Chargement..." : null}
    <ProfilePage resource={resource} />
  
```

```
</>
);
}
```

Essayez sur CodeSandbox

Il ne nous aura fallu que sept lignes de code pour ajouter cette transition :

- Nous avons importé le Hook `useTransition` et l'avons utilisé dans le composant pour mettre à jour l'état.
- Nous avons passé `{ timeoutMs: 3000 }` pour rester sur l'écran précédent à raison de 3 secondes maximum.
- Nous avons enrobé la mise à jour de l'état par un `startTransition` pour indiquer à React qu'il pouvait choisir de la différer.
- Nous utilisons `isPending` pour communiquer la notion d'une transition d'état en cours à l'utilisateur et désactiver le bouton.

Résultat : cliquer sur « Suivant » n'entraîne pas une transition d'état immédiate vers un état de chargement « indésirable » mais reste plutôt sur l'écran précédent pour y communiquer une progression.

Où survient la mise à jour ? {#where-does-the-update-happen}

Voilà qui n'était pas très difficile à implémenter. Cependant, si vous commencez à réfléchir sur les mécanismes qui rendent ce résultat possible, ça risque de vous faire quelques noeuds au cerveau. Si nous définissons l'état, comment se fait-il que nous n'en constatons pas le résultat immédiatement ? Où a lieu le prochain rendu de `<ProfilePage>` ?

Clairement, les deux « versions » de `<ProfilePage>` existent en même temps. On sait que l'ancienne existe parce qu'on la voit à l'écran et qu'on y affiche même un indicateur de progression. Et on sait que la nouvelle version existe aussi *quelque part*, parce que c'est celle qu'on attend !

Mais comment peut-on avoir en même temps deux versions du même composant ?

On touche là à l'essence-même du mode concurrent. Nous avons dit précédemment que c'était un peu comme si React travaillait sur la mise à jour de l'état sur une « branche ». Une autre façon de conceptualiser ça consiste à se dire qu'enrober une mise à jour de l'état avec `startTransition` déclenche son rendu « *dans un univers parallèle* », comme dans les films de science-fiction. Nous ne « voyons » pas cet univers directement—mais nous pouvons en détecter des signaux qui nous informent que quelque chose s'y passe (`isPending`). Quand la mise à jour est enfin prête, nos « univers » fusionnent, et nous voyons le résultat à l'écran !

Jouez un peu plus avec la [démo](#), et tentez d'imaginez ce comportement derrière elle.

Bien entendu, ces deux versions de l'arbre effectuant leur rendu *en même temps* ne sont qu'une illusion, tout comme l'idée que tous les programmes tournent sur votre ordinateur en même temps est une illusion. Un système d'exploitation bascule entre les différentes applications très rapidement. De façon similaire, React peut basculer entre la version de votre arbre affichée à l'écran et celle « en préparation » pour l'affichage suivant.

Une API comme `useTransition` vous permet de vous concentrer sur l'expérience utilisateur souhaitée, sans avoir à vous encombrer l'esprit avec les détails techniques de son implémentation. Néanmoins, imaginer que

les mises à jour enrobées par `useTransition` surviennent « sur une branche » ou dans un « monde parallèle » reste une métaphore utile.

Les transitions sont partout {#transitions-are-everywhere}

Comme nous l'avons appris dans [Suspense pour le chargement de données](#), tout composant peut « se suspendre » à tout moment s'il a besoin de données qui ne sont pas encore disponibles. Nous pouvons positionner stratégiquement des périmètres `<Suspense>` dans différentes parties de l'arbre pour gérer ça, mais ça ne sera pas toujours suffisant.

Reprendons notre [première démo de Suspense](#) qui ne se préoccupait que d'un profil. Pour le moment, elle ne récupère les données qu'une seule fois. Ajoutons un bouton « Rafraîchir » qui vérifiera si le serveur a des mises à jour à proposer.

Notre premier essai pourrait ressembler à ceci :

```
const initialResource = fetchUserAndPosts();

function ProfilePage() {
  const [resource, setResource] = useState(initialResource);

  function handleRefreshClick() {
    setResource(fetchUserAndPosts());
  }

  return (
    <Suspense fallback={<h1>Chargement du profil...</h1>}>
      <ProfileDetails resource={resource} />
      <button onClick={handleRefreshClick}>
        Rafraîchir
      </button>
      <Suspense fallback={<h1>Chargement des publications...</h1>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
    </Suspense>
  );
}
```

Essayez sur [CodeSandbox](#)

Dans cet exemple, nous commençons à charger les données lorsque le composant se charge et à chaque fois que nous activons « Rafraîchir ». Nous plaçons le résultat de l'appel à `fetchUserAndPosts()` dans l'état pour que les composants plus bas dans l'arbre puissent commencer à lire les nouvelles données de la requête que nous venons de déclencher.

On peut voir dans [cet exemple](#) qu'activer « Rafraîchir » fonctionne bien. Les composants `<ProfileDetails>` et `<ProfileTimeline>` reçoivent une nouvelle prop `resource` qui représente les données à jour, ils « se suspendent » parce que la réponse n'est pas encore là, et nous en voyons les UI de repli. Une fois la réponse chargée, nous voyons les publications mises à jour (notre API factice en ajoute toutes les 3 secondes).

Cependant, l'expérience obtenue est très saccadée. Nous étions en train de consulter une page, mais celle-ci a été remplacée par un état de chargement alors même que nous étions en train d'interagir avec. C'est déroutant. **Tout comme précédemment, pour éviter d'afficher un état de chargement indésirable, nous pouvons enrober la mise à jour de l'état par une transition :**

```
function ProfilePage() {
  const [startTransition, isPending] = useTransition({
    // Attendre 10 secondes avant d'afficher l'UI de repli
    timeoutMs: 10000
  });
  const [resource, setResource] = useState(initialResource);

  function handleRefreshClick() {
    startTransition(() => {
      setResource(fetchProfileData());
    });
  }

  return (
    <Suspense fallback={<h1>Chargement du profil...</h1>}>
      <ProfileDetails resource={resource} />
      <button
        onClick={handleRefreshClick}
        disabled={isPending}
      >
        {isPending ? "Rafraîchissement..." : "Rafraîchir"}
      </button>
      <Suspense fallback={<h1>Chargement des publications...</h1>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
    </Suspense>
  );
}
```

Essayez sur CodeSandbox

Voilà qui est beaucoup plus agréable ! Cliquer sur « Rafraîchir » ne nous vire plus de la page que nous étions en train de consulter. Nous voyons que quelque chose est en train de charger « en place » dans le contenu, et lorsque les données sont enfin prêtes, elles sont automatiquement affichées.

Intégrer les transitions au système de conception {#baking-transitions-into-the-design-system}

Nous voyons désormais qu'il est très courant d'avoir besoin de `useTransition`. Presque chaque clic sur un bouton ou autre interaction qui pourrait entraîner la suspension d'un composant bénéficierait d'un enrobage dans `useTransition` pour éviter de masquer accidentellement du contenu avec lequel l'utilisateur interagit.

Ça peut vite entraîner beaucoup de code répétitif d'un composant à l'autre. C'est pourquoi **nous conseillons généralement d'intégrer `useTransition` dans le système de conception des composants de votre appli**. On pourrait par exemple extraire la logique de transition dans notre propre composant `<Button>` :

```

function Button({ children, onClick }) {
  const [startTransition, isPending] = useTransition({
    timeoutMs: 10000
  });

  function handleClick() {
    startTransition(() => {
      onClick();
    });
  }

  const spinner = (
    // ...
  );

  return (
    <>
      <button
        onClick={handleClick}
        disabled={isPending}
      >
        {children}
      </button>
      {isPending ? spinner : null}
    </>
  );
}

```

Essayez sur [CodeSandbox](#)

Remarquez que le bouton ne se soucie pas de savoir *quel* état vous mettez à jour. Il enrobe dans une transition *n'importe quelle* mise à jour d'état qui survient au sein du gestionnaire `onClick`. À présent que notre `<Button>` s'occupe tout seul de mettre la transition en place, le composant `<ProfilePage>` n'a plus besoin de s'en occuper lui-même :

```

function ProfilePage() {
  const [resource, setResource] = useState(initialResource);

  function handleRefreshClick() {
    setResource(fetchProfileData());
  }

  return (
    <Suspense fallback={<h1>Chargement du profil...</h1>}>
      <ProfileDetails resource={resource} />
      <Button onClick={handleRefreshClick}>
        Rafraîchir
      </Button>
      <Suspense fallback={<h1>Chargement des publications...</h1>}>
        <ProfileTimeline resource={resource} />
    </Suspense>
  );
}

```

```

        </Suspense>
    </Suspense>
);
}

```

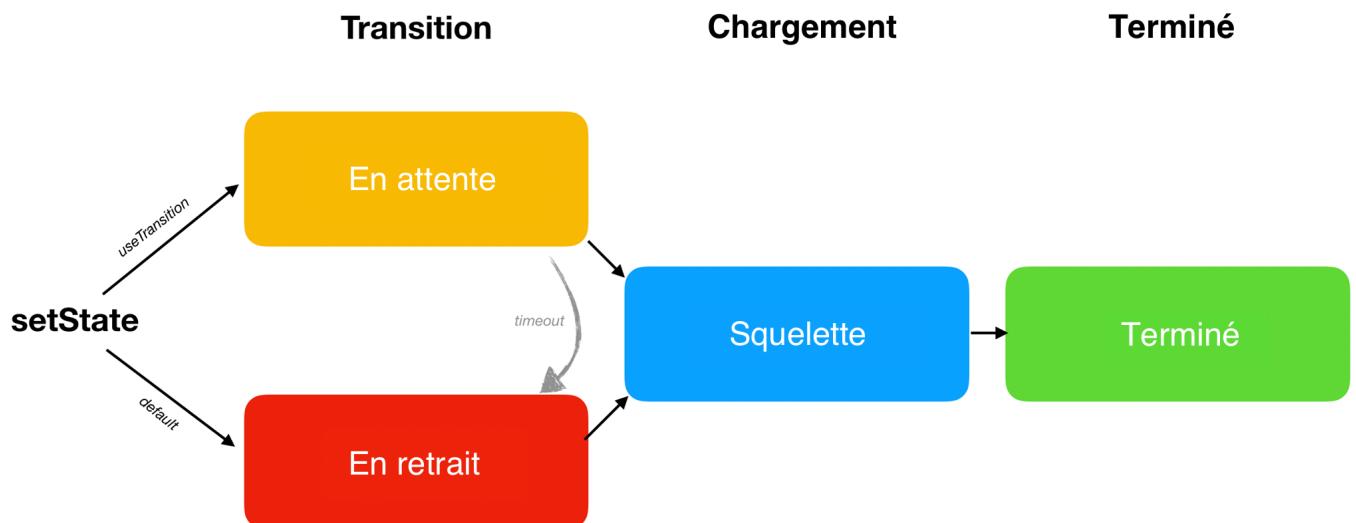
Essayez sur CodeSandbox

Quand on clique sur un bouton, celui-ci démarre une transition et appelle `props.onClick()` à l'intérieur—ce qui déclenche `handleRefreshClick` dans le composant `<ProfilePage>`. On commence à charger les données à jour, mais ça n'active pas l'UI de repli car nous sommes au sein d'une transition, et que l'expiration de 10 secondes spécifiée dans l'appel à `useTransition` n'est pas encore atteinte. Pendant que la transition est active, le bouton affiche un indicateur de chargement intégré.

On voit maintenant comment le mode concurrent nous aide à obtenir une bonne expérience utilisateur sans pour autant sacrifier l'isolation et la modularité des composants. React coordonne la transition.

Les trois étapes {#the-three-steps}

À ce stade, nous avons exploré tous les états visuels distincts à travers lesquels passe une mise à jour. Dans cette section, nous allons leur donner des noms et discuter de la progression de l'un à l'autre.



Tout au bout, nous avons l'état **Terminé**. C'est là que nous voulons arriver au final. Il représente le moment où le prochain écran est pleinement affiché et ne charge plus de données supplémentaires.

Mais avant que notre écran soit Terminé, nous aurons peut-être besoin de charger des données ou du code. Lorsque nous serons sur le prochain écran, mais que certaines parties seront encore en train de charger, nous appellerons ça l'état **Squelette**.

Pour finir, il y a deux principales façons d'arriver à l'état Squelette. Nous illustrerons la différence entre les deux à l'aide d'un exemple concret.

Par défaut : En retrait → Squelette → Terminé {#default-receded-skeleton-complete}

Ouvrez [cet exemple](#) et cliquez sur « Ouvrir le profil ». Vous verrez plusieurs états visuels l'un après l'autre :

- **En retrait** : pendant une seconde, vous verrez l'UI de repli `<h1>Chargement de l'appli...</h1>`.
- **Squelette** : vous verrez le composant `<ProfilePage>` avec à l'intérieur `<h2>Chargement des publications...</h2>`.
- **Terminé** : vous verrez le composant `<ProfilePage>` sans UI de repli à l'intérieur. Tout aura été chargé.

Comment sépare-t-on les états En retrait et Squelette ? La différence tient au fait que l'état **En retrait** donne plus l'impression de « faire un pas en arrière » pour l'utilisateur, alors que l'état **Squelette** donne le sentiment de « faire un pas en avant » dans notre progression vers davantage de contenu.

Dans cet exemple, nous avons commencé notre parcours sur la `<HomePage>` :

```
<Suspense fallback={...}>
  {/* écran précédent */}
  <HomePage />
</Suspense>
```

Après le clic, React a commencé à afficher l'écran suivant :

```
<Suspense fallback={...}>
  {/* prochain écran */}
  <ProfilePage>
    <ProfileDetails />
    <Suspense fallback={...}>
      <ProfileTimeline />
    </Suspense>
  </ProfilePage>
</Suspense>
```

Tant `<ProfileDetails>` que `<ProfileTimeline>` ont besoin de données pour s'afficher, alors ils se suspendent :

```
<Suspense fallback={...}>
  {/* prochain écran */}
  <ProfilePage>
    <ProfileDetails /> {/* se suspend ! */}
    <Suspense fallback={<h2>Chargement des publications...</h2>}>
      <ProfileTimeline /> {/* se suspend ! */}
    </Suspense>
  </ProfilePage>
</Suspense>
```

Quand un composant se suspend, React a besoin d'afficher l'UI de repli la plus proche. Mais dans le cas de `<ProfileDetails>` celle-ci est au niveau racine :

```

<Suspense fallback={

    // On voit cette UI de repli à cause de de <ProfileDetails>
    <h1>Chargement de l'appli...</h1>
}>

    /* prochain écran */

    <ProfilePage>
        <ProfileDetails /> {/* se suspend ! */}

        <Suspense fallback={...}>
            <ProfileTimeline />
        </Suspense>
    </ProfilePage>
</Suspense>

```

C'est pourquoi lorsque nous cliquons sur un bouton, on a l'impression de « faire un pas en arrière ». Le périmètre `<Suspense>` qui affichait jusque-là du contenu utile (`<HomePage />`) a dû « se mettre en retrait » pour afficher l'UI de repli (`<h1>Chargement de l'appli...</h1>`). On appelle ça l'état **En retrait**.

Au fil du chargement des données, React retentera l'affichage, et `<ProfileDetails>` pourra s'afficher correctement. Nous aboutirons alors à l'état **Squelette**. On voit la nouvelle page avec des parties manquantes :

```

<Suspense fallback={...}>

    /* prochain écran */

    <ProfilePage>
        <ProfileDetails />
        <Suspense fallback={

            // On voit cette UI de repli à cause de <ProfileTimeline>
            <h2>Chargement des publications...</h2>
}>

        <ProfileTimeline /> {/* se suspend ! */}

    </Suspense>
</ProfilePage>
</Suspense>

```

Là aussi, au final, nous atteindrons l'état **Terminé**.

Ce scénario (En retrait → Squelette → Terminé) est celui par défaut. Cependant, l'état En retrait est désagréable parce qu'il « masque » des informations existantes. C'est pourquoi React nous permet de choisir une séquence différente (**En attente** → Squelette → Terminé) avec `useTransition`.

Préférable : En attente → Squelette → Terminé {#preferred-pending-skeleton-complete}

Quand nous utilisons `useTransition`, React nous permet de « rester » sur l'écran précédent—and d'y placer un indicateur d'attente. Nous appelons ça l'état **En attente**. Le ressenti est nettement meilleur que pour l'état En retrait, car aucun contenu existant ne disparaît, et la page reste interactive.

Vous pouvez comparer ces deux exemples pour ressentir la différence :

- Défaut : **En retrait → Squelette → Terminé**

- **Préférable : En attente → Squelette → Terminé**

La seule différence entre ces deux exemples tient à ce que le premier utilise des `<button>` classiques, alors que le second utilise notre composant personnalisé `<Button>`, qui intègre un `useTransition`.

Enrobez les fonctionnalités paresseuses avec `<Suspense> {#wrap-lazy-features-in-suspense}`

Ouvrez [cet exemple](#). Quand vous activez un bouton, vous voyez l'état En attente pendant une seconde avant de passer à la suite. Cette transition est agréable et fluide.

Nous allons maintenant ajouter une fonctionnalité toute neuve à la page de profil : une liste de faits amusants relatifs à la personne :

```
function ProfilePage({ resource }) {
  return (
    <>
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h2>Chargement des publications...</h2>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
      <ProfileTrivia resource={resource} />
    </>
  );
}

function ProfileTrivia({ resource }) {
  const trivia = resource.trivia.read();
  return (
    <>
      <h2>Faits amusants</h2>
      <ul>
        {trivia.map(fact => (
          <li key={fact.id}>{fact.text}</li>
        )));
      </ul>
    </>
  );
}
```

Essayez sur [CodeSandbox](#)

Si à présent vous activez « Ouvrir le profil », vous verrez que quelque chose cloche. Ça prend désormais sept bonnes secondes pour effectuer la transition ! C'est parce que notre API de faits amusants est trop lente.

Imaginons que nous ne puissions pas l'accélérer : comment alors améliorer l'expérience utilisateur malgré cette contrainte ?

Si nous ne voulons pas rester à l'état En attente trop longtemps, notre premier instinct pourrait être d'ajuster le `timeoutMs` dans `useTransition` pour le réduire, par exemple à `3000`. Vous pouvez essayer ça [ici](#). Ça nous permet d'échapper à un état En attente prolongé, mais ça ne nous donne pas pour autant des contenus utiles à afficher !

Il y a un moyen plus simple de résoudre ça. **Plutôt que d'abréger la transition, nous pouvons « déconnecter » le composant lent de celle-ci** en l'enrobant dans son propre périmètre `<Suspense>` :

```
function ProfilePage({ resource }) {
  return (
    <>
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h2>Chargement des publications...</h2>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
      <Suspense fallback={<h2>Chargement des faits amusants...</h2>}>
        <ProfileTrivia resource={resource} />
      </Suspense>
    </>
  );
}
```

Essayez sur [CodeSandbox](#)

Voilà qui met en lumière un point important. React préfèrera toujours arriver à l'état Squelette le plus tôt possible. Même si nous utilisons partout des transitions à expiration longue, React ne restera pas dans l'état En attente plus longtemps que le strict nécessaire pour éviter l'état En retrait.

Si une fonctionnalité ne constitue pas une partie vitale du prochain écran, enrobez-la dans un `<Suspense>` pour la laisser se charger paresseusement. Vous gardez ainsi que le reste du contenu sera visible le plus tôt possible. Réciproquement, si un écran n'a aucun intérêt à être affiché sans un composant donné, tel que `<ProfileDetails>` dans notre exemple, ne l'enrobez pas dans un `<Suspense>`. Ainsi les transitions « attendront » que ce composant soit disponible.

Le « train » de révélations de Suspense (#suspense-reveal-train)

Lorsque nous sommes déjà sur le prochain écran, il peut arriver que les données nécessaires pour « déverrouiller » des périmètres `<Suspense>` distincts arrivent en succession rapide. Par exemple, deux réponses distinctes pourraient arriver respectivement après 1 000 et 1 050 ms. Si vous avez déjà attendu une seconde, vous ne percevez pas l'attente supplémentaire de 50 ms. C'est pourquoi React ne révèle les périmètres `<Suspense>` que selon un horaire déterminé, comme un « train » qui arriverait périodiquement. Il échange un court délai contre une réduction des modifications majeures à la mise en page et du nombre de changements visuels présentés à l'utilisateur.

Vous pouvez voir une démo de ça [ici](#). Les réponses « publications » et « faits amusants » arrivent à 100 ms l'une de l'autre. Mais React les regroupe pour « révéler » leurs périmètres en une seule passe.

Différer un indicateur d'attente (#delaying-a-pending-indicator)

Notre composant `Button` affichera immédiatement l'indicateur d'attente une fois cliqué :

```
function Button({ children, onClick }) {
  const [startTransition, isPending] = useTransition({
```

```

    timeoutMs: 10000
});

// ...

return (
<>
  <button onClick={handleClick} disabled={isPending}>
    {children}
  </button>
  {isPending ? spinner : null}
</>
);
}

```

[Essayez sur CodeSandbox](#)

Ça signale à l'utilisateur qu'un travail a lieu. Toutefois, si cette transition est relativement courte (disons, moins de 500 ms), cet affichage est plus une distraction qu'autre chose et peut faire que la transition elle-même semble *plus lente*.

Une solution possible à ça consiste à *différer l'affichage du spinner lui-même* :

```

.DelayedSpinner {
  animation: 0s linear 0.5s forwards makeVisible;
  visibility: hidden;
}

@keyframes makeVisible {
  to {
    visibility: visible;
  }
}

```

```

const spinner = (
  <span className="DelayedSpinner">
    {/* ... */}
  </span>
);

return (
<>
  <button onClick={handleClick}>{children}</button>
  {isPending ? spinner : null}
</>
);

```

[Essayez sur CodeSandbox](#)

Avec cette modification, même si nous sommes dans l'état En attente, nous n'affichons aucune indication à l'utilisateur jusqu'à ce que 500 ms se soient écoulés. Ça peut sembler une amélioration superflue quand les réponses API sont lentes. Mais comparez le ressenti [avant](#) et [après](#) lorsque l'appel API est rapide. Même si le reste du code n'a pas changé, supprimer un état de chargement « trop rapide » améliore les performances perçues en évitant d'attirer l'attention de l'utilisateur.

En résumé {#recap}

Les points les plus importants que nous avons appris jusqu'à présent sont :

- Par défaut, notre séquence de chargement est En retrait → Squelette → Terminé.
- L'état En retrait n'est pas très agréable parce qu'il masque du contenu existant.
- Avec `useTransition`, nous pouvons choisir de commencer plutôt par un état En attente. Ça nous gardera sur l'écran précédent pendant que le prochain se prépare.
- Si nous ne voulons pas qu'un composant ralentisse une transition, nous pouvons l'enrober dans son propre périmètre `<Suspense>`.
- Plutôt que de faire un `useTransition` au sein de tous nos composants, nous pouvons l'intégrer à notre système de conception.

Autres approches {#other-patterns}

Les transitions sont probablement l'approche pour une UI concurrente que vous rencontrerez le plus fréquemment, mais il existe d'autres approches que vous pourriez trouver utiles.

Dissocier les états à forte et faible priorité {#splitting-high-and-low-priority-state}

Quand vous concevez des composants React, vous cherchez dans l'idéal à déterminer la « représentation minimale » de l'état. Par exemple, au lieu de conserver dans l'état `firstName`, `lastName` et `fullName`, il est généralement préférable de n'y stocker que `firstName` et `lastName`, et de calculer `fullName` lors du rendu. Ça nous permet d'éviter les erreurs dues à une mise à jour partielle de l'état.

En revanche, le mode concurrent recèle des cas où vous pourriez *vouloir* « dupliquer » des données dans des variables d'état distinctes. Prenez cette minuscule appli de traduction :

```
const initialQuery = "Bonjour, monde";
const initialResource = fetchTranslation(initialQuery);

function App() {
  const [query, setQuery] = useState(initialQuery);
  const [resource, setResource] = useState(initialResource);

  function handleChange(e) {
    const value = e.target.value;
    setQuery(value);
    setResource(fetchTranslation(value));
  }

  return (
    <>
    <input
```

```

        value={query}
        onChange={handleChange}
      />
      <Suspense fallback={<p>Changement...</p>}>
        <Translation resource={resource} />
      </Suspense>
    </>
  );
}

function Translation({ resource }) {
  return (
    <p>
      <b>{resource.read()}</b>
    </p>
  );
}

```

Essayez sur [CodeSandbox](#)

Remarquez comme, lorsque vous tapez dans le champ de saisie, le composant `<Translation>` se suspend, vous affichant l'UI de repli `<p>Changement...</p>` jusqu'à obtenir des résultats à jour. Ce n'est pas idéal. Il serait préférable que vous puissiez brièvement voir la *précédente* traduction, tandis que nous chargeons la prochaine.

D'ailleurs, si vous ouvrez la console, vous y verrez cet avertissement :

Warning: App triggered a user-blocking update that suspended.

The fix is to split the update into multiple parts: a user-blocking update to provide immediate feedback, and another update that triggers the bulk of the changes.

Refer to the documentation for `useTransition` to learn how to implement this pattern.

Avertissement : l'appli a déclenché une mise à jour suspensive, bloquante pour l'utilisateur.

Pour corriger ça, découpez la mise à jour en plusieurs parties : une bloquante qui fournit un retour visuel immédiat, et une qui déclenche l'essentiel des modifications.

Consultez la documentation de `useTransition` pour en apprendre davantage sur la façon d'implémenter cette approche.

Comme nous l'avons vu auparavant, si une mise à jour d'état entraîne la suspension d'un composant, cette mise à jour devrait être enrobée dans une transition. Essayons d'ajouter `useTransition` à notre composant :

```

function App() {
  const [query, setQuery] = useState(initialQuery);

```

```

const [resource, setResource] = useState(initialResource);
const [startTransition, isPending] = useTransition({
  timeoutMs: 5000
});

function handleChange(e) {
  const value = e.target.value;
  startTransition(() => {
    setQuery(value);
    setResource(fetchTranslation(value));
  });
}

// ...
}

```

Essayez sur CodeSandbox

Essayez de saisir une valeur à présent. Quelque chose cloche ! Le champ n'est mis à jour que très lentement.

Nous avons corrigé le premier problème (la suspension hors d'une transition). Mais maintenant, à cause de la transition, notre état n'est pas mis à jour immédiatement : il ne peut donc pas « piloter » le champ contrôlé !

La solution **consiste à découper l'état en deux parties** : une partie à « forte priorité » qui est mise à jour tout de suite, et une à « faible priorité » qui peut se permettre d'attendre la transition.

Dans notre exemple, on a déjà deux variables d'état. Le texte saisi est dans `query` et la traduction est lue depuis `resource`. Nous voulons que les modifications apportées à `query` soient traitées immédiatement, mais que celles de `resource` (c'est-à-dire le chargement d'une nouvelle traduction) déclenchent une transition.

Du coup le bon correctif consiste à mettre `setQuery` (qui ne suspend rien) *hors* de la transition, mais de placer `setResource` (qui suspendra) à *l'intérieur* de celle-ci.

```

function handleChange(e) {
  const value = e.target.value;

  // Hors de la transition (urgent)
  setQuery(value);

  startTransition(() => {
    // Dans la transition (peut être différé)
    setResource(fetchTranslation(value));
  });
}

```

Essayez sur CodeSandbox

Avec cet ajustement, tout fonctionne comme on le souhaite. On peut taper une valeur et la voir immédiatement, quant à la traduction, elle « rattrape » ce qu'on a saisi un peu plus tard.

Différer une valeur {#deferring-a-value}

Par défaut, React assurera toujours un rendu cohérent de l'UI. Prenez le code suivant :

```
<>
  <ProfileDetails user={user} />
  <ProfileTimeline user={user} />
</>
```

React garantit qu'à tout moment, quand nous regardons ces composants à l'écran, ils reflèteront les données issues du même `user`. Si un `user` différent nous est passé suite à une mise à jour d'état, vous verrez les deux composants se mettre à jour d'un bloc. Il serait impossible d'enregistrer une vidéo de l'écran et de trouver ensuite un seul *frame* où ces composants afficheraient des données issues d'objets `user` différents. (Et si vous y arrivez un jour, ouvrez un ticket, c'est un bug !)

Cette approche a du sens dans la vaste majorité des cas. Une UI incohérente est déroutante voire dangereuse pour les utilisateurs. (Par exemple, vous imaginez bien que ce serait l'enfer si le bouton Envoyer de Messenger et le panneau de conversation n'étaient « pas d'accord » sur la conversation en cours.)

Ceci dit, il peut parfois être utile d'introduire volontairement un décalage. On pourrait le faire manuellement en « découpant » l'état comme on l'a fait ci-dessus, mais React nous offre un Hook prédéfini pour ça :

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

const deferredValue = useDeferredValue(value, {
  timeoutMs: 5000
});
```

Pour illustrer cette fonctionnalité, nous allons utiliser [l'exemple de la bascule de profil](#). Cliquez sur le bouton « Suivant » et remarquez que ça prend une seconde pour achever la transition.

Disons que la récupération des détails utilisateurs est très rapide et ne prend que 300 millisecondes. Pour le moment, nous attendons une seconde entière parce que nous avons besoin tant des détails de l'utilisateur que de ses publications pour afficher une page de profil cohérente. Mais qu'en serait-il si nous voulions afficher les détails plus tôt ?

Si nous acceptons de sacrifier la cohérence, nous pouvons **passer des données potentiellement obsolètes aux composants qui retardent notre transition**. C'est précisément ce que `useDeferredValue()` nous permet de faire :

```
function ProfilePage({ resource }) {
  const deferredResource = useDeferredValue(resource, {
    timeoutMs: 1000
});
  return (
    <Suspense fallback={<h1>Chargement du profil...</h1>}>
      <ProfileDetails resource={resource} />
    </Suspense>
  );
}
```

```

        <Suspense fallback={<h1>Chargement des publications...</h1>}>
          <ProfileTimeline
            resource={deferredResource}
            isStale={deferredResource !== resource}
          />
        </Suspense>
      </Suspense>
    );
}

function ProfileTimeline({ isStale, resource }) {
  const posts = resource.posts.read();
  return (
    <ul style={{ opacity: isStale ? 0.7 : 1 }}>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}

```

Essayez sur [CodeSandbox](#)

Le compromis que nous faisons ici tient à ce que `<ProfileTimeline>` sera incohérente vis-à-vis des autres composants et affichera potentiellement un élément plus ancien. En cliquant sur « Suivant » plusieurs fois, vous allez le remarquer. Mais grâce à ça, nous pouvons raccourcir le temps de la transition de 1 000 ms à 300 ms.

La pertinence d'un tel compromis dépend de votre situation. Mais ça reste un outil bien pratique, surtout quand le contenu ne change pas de façon très prononcée d'un élément à l'autre, et que l'utilisateur est susceptible de ne même pas remarquer qu'ils ont des données obsolètes pendant une seconde.

Notez bien que `useDeferredValue` n'est pas *seulement* utile pour le chargement de données. Elle nous aide aussi lorsqu'une arborescence de composants lourde ralentit une interaction (par ex. la saisie dans un champ). Tout comme nous pouvons « différer » une valeur qui prend trop longtemps à se charger (et afficher l'ancienne valeur en dépit des mises à jour d'autres composants), nous pouvons faire la même chose à des arbres qui prennent trop de temps pour leur rendu.

Par exemple, prenez une liste filtrable comme celle-ci :

```

function App() {
  const [text, setText] = useState("bonjour");

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <div className="App">
      <label>
        Tapez une valeur dans le champ :{" "}
      </label>
    
```

```

        <input value={text} onChange={handleChange} />
      </label>
    ...
    <MySlowList text={text} />
  </div>
);
}

```

Essayez sur CodeSandbox

Dans cet exemple, **chaque élément dans <MySlowList> est artificiellement ralenti : chacun d'eux bloque le thread pour quelques millisecondes**. On ne ferait jamais ça dans une véritable appli, mais ça nous aide à simuler ce qui pourrait se passer dans une arborescence de composants profonde qui ne contiendrait pas pour autant d'endroits évidents à optimiser.

On peut voir comme la saisie dans le champ cause une expérience saccadée. Ajoutons maintenant `useDeferredValue` :

```

function App() {
  const [text, setText] = useState("bonjour");
  const deferredText = useDeferredValue(text, {
    timeoutMs: 5000
  });

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <div className="App">
      <label>
        Tapez une valeur dans le champ :{" "}
        <input value={text} onChange={handleChange} />
      </label>
    ...
    <MySlowList text={deferredText} />
  </div>
);
}

```

Essayez sur CodeSandbox

À présent la frappe cause beaucoup moins de saccade, mais au prix d'un affichage différé des résultats.

En quoi est-ce différent du *debouncing* ? Notre exemple avait un délai artificiel fixe (3 ms pour chacun des 80 éléments), donc il y aura toujours un délai, peu importe la vitesse de notre ordinateur. En revanche, la valeur de `useDeferredValue` n'est « à la traîne » que si le rendu prend du temps. React n'impose aucun retard minimum. Avec une charge de travail plus réaliste, vous pouvez vous attendre à ce que le retard s'adapte à l'appareil de l'utilisateur. Sur des machines rapides, le retard sera plus court voire inexistant, et sur des

machines lentes, il se fera davantage sentir. Dans les deux cas, l'appli restera réactive. C'est l'avantage de ce mécanisme par rapport au *debouncing* ou au *throttling*, qui imposent toujours un délai minimum et par ailleurs ne permettent pas d'éviter de bloquer le thread pendant le rendu.

Même si on améliore bien ici la réactivité, cet exemple n'est pas encore engageant parce que le mode concurrent manque de certaines optimisations cruciales pour ce cas d'usage. Quoi qu'il en soit, il reste intéressant de voir que des fonctionnalités comme `useDeferredValue` (ou `useTransition`) sont utiles lorsqu'on attend après aussi bien une réponse réseau qu'un travail de calcul pur.

SuspenseList {#suspenselist}

`<SuspenseList>` est la dernière approche liée à l'orchestration des états de chargement.

Prenez cet exemple :

```
function ProfilePage({ resource }) {
  return (
    <>
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h2>Chargement des publications...</h2>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
      <Suspense fallback={<h2>Chargement des faits amusants...</h2>}>
        <ProfileTrivia resource={resource} />
      </Suspense>
    </>
  );
}
```

Essayez sur CodeSandbox

La durée de l'appel API dans cet exemple est aléatoire. Si vous rafraîchissez encore et encore, vous remarquerez que parfois les publications arrivent en premier, alors que d'autres fois ce sont les « faits amusants ».

C'est un problème. Si la réponse des faits amusants arrive en premier, on verra les faits amusants sous l'UI de repli `<h2>Chargement des publications...</h2>` des publications. On pourrait alors commencer à les lire, sauf que soudainement la réponse des *publications* arrive, et décale nos faits vers le bas. C'est très désagréable.

Une manière de corriger ça consiste à les placer tous deux dans le même périmètre :

```
<Suspense fallback={<h2>Chargement des publications et des faits amusants...
</h2>}>
  <ProfileTimeline resource={resource} />
  <ProfileTrivia resource={resource} />
</Suspense>
```

Essayez sur CodeSandbox

Le souci avec ce correctif est qu'à présent nous devons *toujours* attendre que les deux contenus soient chargés. Et pourtant, si ce sont les *publications* qui répondent en premier, il n'y a pas de raison d'attendre pour les afficher. Quand les faits amusants arriveront plus tard, ils ne décaleront pas la mise en page parce qu'ils seront situés sous les publications.

D'autres voies de correction, telles que la composition sur-mesure de promesses, deviennent vite délicates à mettre en œuvre lorsque les états de chargement sont situés dans des composants distincts plus bas dans l'arbre.

Pour résoudre ça, nous allons importer `SuspenseList` :

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

`<SuspenseList>` coordonne « l'ordre de révélation » des noeuds descendants `<Suspense>` les plus proches à l'intérieur de lui :

```
function ProfilePage({ resource }) {
  return (
    <SuspenseList revealOrder="forwards">
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h2>Chargement des publications...</h2>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
      <Suspense fallback={<h2>Chargement des faits amusants...</h2>}>
        <ProfileTrivia resource={resource} />
      </Suspense>
    </SuspenseList>
  );
}
```

Essayez sur CodeSandbox

L'option `revealOrder="forwards"` signifie que les noeuds `<Suspense>` les plus proches dans la liste **ne « révèleront » leur contenu que dans l'ordre de leur apparition dans l'arbre, même si leurs données arrivent dans un ordre différent.** `<SuspenseList>` a d'autres modes intéressants : essayez de remplacer `"forwards"` par `"backwards"` ou `"together"` et regardez ce que ça donne.

Vous pouvez contrôler combien d'états de chargement sont visibles à un instant donné grâce à la prop `tail`. Si nous précisons `tail="collapsed"`, nous verrons *au maximum une* UI de repli à la fois. Vous pouvez jouer avec [ici](#).

Gardez à l'esprit que `<SuspenseList>` est composable, comme tout dans React. Par exemple, vous pouvez créer une grille en plaçant plusieurs lignes `<SuspenseList>` au sein d'un tableau `<SuspenseList>`.

Prochaines étapes {#next-steps}

Le mode concurrent offre un puissant modèle de programmation d'UI et un jeu de nouvelles primitives composable pour vous aider à orchestrer de délicieuses expériences utilisateurs.

C'est le résultat de plusieurs années de recherche et développement, et il n'est pas terminé. Dans la section sur [l'adoption du mode concurrent](#), nous vous expliquerons comment vous pouvez l'essayer et ce que vous pouvez en attendre.

Attention

Cette page décrit **des fonctionnalités expérimentales qui ne sont pas encore disponibles dans une version stable**. Ne vous basez pas sur les builds expérimentaux de React pour vos applis en production. Ces fonctionnalités sont susceptibles d'évoluer de façon significative et sans avertissement avant d'intégrer officiellement React.

Cette documentation est destinée aux personnes curieuses ou habituées à adopter les nouvelles technologies très tôt. **Si vous débutez en React, ne vous préoccupez pas de ces fonctionnalités :** vous n'avez pas besoin de les apprendre pour le moment. Par exemple, si vous cherchez un tutoriel sur le chargement de données qui fonctionne dès maintenant, lisez plutôt [cet article](#).

React 16.6 ajoutait un composant `<Suspense>` qui vous permettait « d'attendre » que du code soit chargé en spécifiant déclarativement un état de chargement (tel qu'un *spinner*) pendant l'attente :

```
const ProfilePage = React.lazy(() => import('./ProfilePage')); // Chargé à la demande

// Affiche un spinner pendant que le profil se charge
<Suspense fallback={<Spinner />}>
  <ProfilePage />
</Suspense>
```

Suspense pour le chargement de données est une nouvelle fonctionnalité qui vous permet d'utiliser également `<Suspense>` pour « attendre » déclarativement n'importe quoi d'autre, y compris le chargement de données distantes. Cette page se concentre sur ce cas d'utilisation, mais vous pouvez utiliser cette technique pour attendre des images, des scripts, ou d'autres traitements asynchrones.

- Qu'est-ce que Suspense, exactement ?
 - Ce que Suspense n'est pas
 - Ce que Suspense vous permet de faire
- Utiliser Suspense en pratique
 - Et si je n'utilise pas Relay ?
 - À l'attention des auteurs de bibliothèques
- Les approches traditionnelles vs. Suspense
 - Approche 1 : *fetch-on-render* (sans utiliser Suspense)
 - Approche 2 : *fetch-then-render* (sans utiliser Suspense)
 - Approche 3 : *render-as-you-fetch* (en utilisant Suspense)
- Démarrer le chargement tôt
 - On expérimente encore
- Suspense et les situations de compétition (*race conditions, NdT*)
 - Compétitions avec `useEffect`
 - Compétitions avec `componentDidUpdate`
 - Le problème
 - Résoudre les situations de compétition avec Suspense
- Gérer les erreurs
- Prochaines étapes

Qu'est-ce que Suspense, exactement ? {#what-is-suspense-exactly}

Suspense permet à vos composants « d'attendre » quelque chose avant qu'ils s'affichent. Dans [cet exemple](#), deux composants attendent le résultat d'un appel API asynchrone destiné à charger des données :

```
const resource = fetchProfileData();

function ProfilePage() {
  return (
    <Suspense fallback={<h1>Chargement du profil...</h1>}>
      <ProfileDetails />
      <Suspense fallback={<h1>Chargement des publications...</h1>}>
        <ProfileTimeline />
      </Suspense>
    </Suspense>
  );
}

function ProfileDetails() {
  // Essaie de lire les infos utilisateur, bien qu'elles puissent ne pas être
  encore chargées
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}

function ProfileTimeline() {
  // Essaie de lire les publications, bien qu'elles puissent ne pas être encore
  chargées
  const posts = resource.posts.read();
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}
```

Essayez sur [CodeSandbox](#)

Cette démo est là pour vous ouvrir l'appétit. Ne vous inquiétez pas si elle est déroutante à ce stade, nous la décrirons plus en détail dans un instant. Gardez à l'esprit que Suspense est davantage un *mécanisme*, et que les API spécifiques dans l'exemple ci-avant, telles que `fetchProfileData()` ou `resource.posts.read()`, n'ont que peu d'importance. Si vous êtes curieux·se, vous pouvez toujours en consulter l'implémentation directement dans la [sandbox de démonstration](#).

Suspense n'est pas une bibliothèque de chargement de données. C'est un **mécanisme à destination des bibliothèques de chargement de données** pour qu'elles puissent indiquer à React que *les données que lit un composant ne sont pas encore disponibles*. React peut alors attendre qu'elles le deviennent et mettre à jour

l'interface utilisateur (UI). Chez Facebook, nous utilisons Relay et sa [nouvelle intégration avec Suspense](#). Nous pensons que d'autres bibliothèques, telles qu'Apollo, fourniront des intégrations similaires.

Sur le long terme, nous prévoyons que Suspense deviendra le moyen principal de lire des données asynchrone depuis des composants, et ce quelle que soit la provenance des données.

Ce que Suspense n'est pas {#what-suspense-is-not}

Suspense est significativement différent des approches existantes pour ce type de problème, de sorte qu'au premier abord il est facile de l'interpréter de travers. Permettez-nous de clarifier les principales erreurs de perception :

- **Ce n'est pas une implémentation de chargement de données.** Ça ne suppose pas que vous utilisiez GraphQL, REST ou tout autre format, bibliothèque, transport ou protocole spécifiques.
- **Ce n'est pas un client prêt à l'emploi.** Vous ne pouvez pas « remplacer » `fetch` ou Relay par Suspense. Mais vous pouvez utiliser une bibliothèque qui s'intègre avec Suspense (par exemple, les [nouvelles API de Relay](#)).
- **Ça ne lie pas le chargement des données à la couche vue.** Ça aide à orchestrer l'affichage des états de chargement dans votre UI, mais ça ne lie pas votre logique réseau à vos composants React.

Ce que Suspense vous permet de faire {#what-suspense-lets-you-do}

Alors quel est le but de Suspense ? Il y a plusieurs manières de répondre à cette question :

- **Ça permet aux bibliothèques de chargement de données de s'intégrer finement avec React.** Si une bibliothèque de chargement de données implémente la prise en charge de Suspense, son utilisation au sein des composants React devient très naturelle.
- **Ça vous permet d'orchestrer vos états de chargement de façon choisie.** Ça ne dit pas *comment* les données sont chargées, mais ça vous permet de contrôler finement la séquence visuelle de chargement de votre appli.
- **Ça vous aide à éviter les situations de compétition** (*race conditions*, *NdT*). Même avec `await`, le code asynchrone est souvent sujet aux erreurs. Suspense donne davantage l'impression de lire les données *de façon synchrone*, comme si elles étaient en fait déjà chargées.

Utiliser Suspense en pratique {#using-suspense-in-practice}

Chez Facebook, nous n'avons pour le moment utilisé en production que l'intégration de Relay avec Suspense.

Si vous cherchez un guide pratique pour démarrer maintenant, [jetez un coup d'œil au guide de Relay !](#)

Il illustre des approches qui ont déjà fait leurs preuves chez nous en production.

Les démos de code sur cette page utilisent une implémentation d'une API « factice » plutôt que Relay. Ça simplifie leur compréhension si vous n'avez pas déjà l'habitude de GraphQL, mais ça ne veut pas dire qu'il s'agisse là de la « bonne manière » de construire une appli avec Suspense. Cette page est davantage conceptuelle, et cherche à vous aider à comprendre *pourquoi* Suspense fonctionne comme il le fait, et quels problèmes il résout.

Et si je n'utilise pas Relay ? {#what-if-i-dont-use-relay}

Si vous n'utilisez pas Relay aujourd'hui, vous aurez peut-être besoin d'attendre avant de pouvoir véritablement essayer Suspense dans votre appli. Pour le moment, c'est la seule implémentation que nous

ayons testée en production et qui nous a satisfaits.

Pendant les prochains mois, plusieurs bibliothèques vont apparaître qui exploiteront de diverses façons les API Suspense. **Si vous préférez apprendre une fois que les choses sont raisonnablement stables, vous voudrez peut-être ignorer tout ça pour le moment, et revenir lorsque l'écosystème Suspense sera plus mûr.**

Vous pouvez aussi écrire votre propre intégration pour une bibliothèque de chargement de données, si vous le souhaitez.

À l'attention des auteur·e·s de bibliothèques {#for-library-authors}

Nous nous attendons à voir la communauté expérimenter largement avec d'autres bibliothèques. Il y a un point important à noter pour les personnes qui maintiennent des bibliothèques de chargement de données.

Bien que ce soit techniquement faisable, Suspense n'est **pas** pour le moment conçu comme une façon de charger les données lorsqu'un composant s'affiche. Il sert plutôt à permettre aux composants d'exprimer qu'ils « attendent » des données qui sont *déjà en cours de chargement*. L'article [Construire des super expériences utilisateurs avec le mode concurrent et Suspense](#) décrit en quoi cette distinction est importante, et comment implémenter cette approche en pratique.

À moins que vous n'ayez une solution pour empêcher les chargements en cascade, nous vous conseillons d'opter pour des API qui favorisent voire exigent un déclenchement du chargement des données en amont du rendu. Pour un exemple concret, vous pouvez regarder comment [l'API Suspense de Relay](#) garantit le pré-chargement. Par le passé, nous n'avons pas communiqué de façon très cohérente sur ce sujet. Suspense pour le chargement de données reste expérimental, de sorte que nos recommandations sont susceptibles de changer avec le temps, au fur et à mesure que nous tirons de nouvelles leçons de notre utilisation en production et améliorons notre compréhension de cette typologie de problèmes.

Les approches traditionnelles vs. Suspense {#traditional-approaches-vs-suspense}

Nous pourrions introduire Suspense sans mentionner les approches répandues de chargement de données. Néanmoins, il serait alors plus difficile de bien percevoir les problèmes que Suspense résout, en quoi ces problèmes méritent une résolution, et ce qui différencie Suspense des solutions existantes.

Nous allons plutôt considérer Suspense comme l'étape suivante logique dans une chronologie d'approches :

- **Fetch-on-render (par exemple, `fetch` dans `useEffect`)** : on commence l'affichage des composants. Chacun d'eux est susceptible de déclencher un chargement de données au sein de ses effets ou méthodes de cycle de vie. Cette approche aboutit souvent à des « cascades ».
- **Fetch-then-render (par exemple, Relay sans Suspense)** : on commence par charger toutes les données pour le prochain écran aussitôt que possible. Quand les données sont prêtes, on affiche le nouvel écran. On ne peut rien faire avant que les données ne soient reçues.
- **Render-as-you-fetch (par exemple, Relay avec Suspense)** : on lance le chargement de toutes les données requises par le prochain écran aussitôt que possible, et on commence le rendu du nouvel écran *immédiatement, avant d'avoir la réponse du réseau*. Au fil de la réception des flux de données, React retente le rendu des composants qui ont encore besoin de données jusqu'à ce que tout soit disponible.

Remarque

Il s'agit là d'une légère simplification, et en pratique les solutions ont tendance à combiner plusieurs approches. Quoi qu'il en soit, nous les examinerons en isolation pour mieux mettre en lumière leurs avantages et inconvénients respectifs.

Pour comparer ces approches, nous allons implémenter une page de profil avec chacune d'entre elles.

Approche 1 : *fetch-on-render* (sans utiliser Suspense) {#approach-1-fetch-on-render-not-using-suspense}

Une façon courante de charger les données dans une application React aujourd'hui consiste à utiliser un effet :

```
// Dans une fonction composant :  
useEffect(() => {  
  fetchSomething();  
, []);  
  
// Ou dans un composant à base de classe :  
componentDidMount() {  
  fetchSomething();  
}
```

Nous appelons cette approche "*fetch-on-render*" parce qu'elle ne commence à charger *qu'après* que le composant s'est affiché. Elle entraîne un problème appelé « la cascade ».

Prenez ces composants `<ProfilePage>` et `<ProfileTimeline>` :

```
function ProfilePage() {  
  const [user, setUser] = useState(null);  
  
  useEffect(() => {  
    fetchUser().then(u => setUser(u));  
, []);  
  
  if (user === null) {  
    return <p>Changement du profil...</p>;  
  }  
  return (  
    <>  
      <h1>{user.name}</h1>  
      <ProfileTimeline />  
    </>  
  );  
}  
  
function ProfileTimeline() {  
  const [posts, setPosts] = useState(null);
```

```

useEffect(() => {
  fetchPosts().then(p => setPosts(p));
}, []);

if (posts === null) {
  return <h2>Chargement des publications...</h2>;
}
return (
  <ul>
    {posts.map(post => (
      <li key={post.id}>{post.text}</li>
    )));
  </ul>
);
}

```

Essayez sur CodeSandbox

Si vous exécutez ce code et examinez les logs dans la console, vous y verrez se dérouler la séquence suivante :

1. On commence à charger les détails de l'utilisateur
2. On attend...
3. On finit de charger les détails de l'utilisateur
4. On commence à charger les publications
5. On attend...
6. On finit de charger les publications

Si le chargement des détails de l'utilisateur prend trois secondes, nous ne *commencerons* à charger les publications qu'au bout de trois secondes ! C'est une « cascade » : une *séquence* involontaire qui aurait dû être parallélisée.

Les cascades sont courantes dans le code qui charge les données au sein du rendu. On peut les corriger, mais à mesure que le produit grandit, les gens préfèreront une solution qui évite carrément ce problème.

Approche 2 : *fetch-then-render* (sans utiliser Suspense) {#approach-2-fetch-then-render-not-using-suspense}

Les bibliothèques peuvent prévenir les cascades en offrant une approche plus centralisée du chargement de données. Par exemple, Relay résout ce problème en déplaçant les informations relatives aux données dont un composant a besoin dans des *fragments* analysables statiquement, qui sont ensuite composés en une seule requête.

Sur cette page, nous ne supposons aucune connaissance préalable de Relay, aussi nous ne l'utiliserons pas dans cet exemple. Nous écrirons plutôt manuellement quelque chose de similaire en combinant nos méthodes de chargement de données :

```

function fetchProfileData() {
  return Promise.all([
    fetchUser(),
    fetchPosts()
  ])
}

```

```
]).then(([user, posts]) => {
  return {user, posts};
})
}
```

Dans cet exemple, `<ProfilePage>` attend les deux requêtes mais les démarre en parallèle :

```
// Lance les chargements aussitôt que possible
const promise = fetchProfileData();

function ProfilePage() {
  const [user, setUser] = useState(null);
  const [posts, setPosts] = useState(null);

  useEffect(() => {
    promise.then(data => {
      setUser(data.user);
      setPosts(data.posts);
    });
  }, []);

  if (user === null) {
    return <p>Chargement du profil...</p>;
  }
  return (
    <>
      <h1>{user.name}</h1>
      <ProfileTimeline posts={posts} />
    </>
  );
}

// Le fils n'a plus besoin de déclencher une chargement
function ProfileTimeline({ posts }) {
  if (posts === null) {
    return <h2>Chargement des publications...</h2>;
  }
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      )));
    </ul>
  );
}
```

Essayez sur CodeSandbox

La séquence d'événements devient la suivante :

1. On commence à charger les détails de l'utilisateur

2. On commence à charger les publications
3. On attend...
4. On finit de charger les détails de l'utilisateur
5. On finit de charger les publications

On a résolu la « cascade » réseau de l'exemple précédent, mais introduit un autre souci pas inadvertance. Nous attendons à présent que *toutes* les données soient chargées, en raison du `Promise.all()` dans `fetchProfileData`, de sorte qu'on ne peut pas afficher les détails du profil tant qu'on n'a pas aussi les publications. On doit attendre les deux.

Naturellement, il est possible de corriger cet exemple spécifique. On pourrait retirer l'appel à `Promise.all()` et attendre chaque promesse séparément. Cependant, cette approche devient progressivement plus ardue au fur et à mesure que nos données et notre arborescence de composants gagnent en complexité. Il est difficile d'écrire des composants fiables lorsque des parties aléatoires de notre arbre de données peuvent manquer ou se périmber, de sorte qu'il est souvent plus pragmatique de charger toutes les données pour le nouvel écran *et ensuite* l'afficher.

Approche 3 : *render-as-you-fetch* (en utilisant Suspense) {#approach-3-render-as-you-fetch-using-suspense}

Dans l'approche précédente, nous chargions les données avant d'appeler `setState` :

1. Commencer le chargement
2. Finir le chargement
3. Commencer le rendu

Avec Suspense, nous déclencherons le chargement en premier, mais inverserons les deux dernières étapes :

1. Commencer le chargement
2. **Commencer le rendu**
3. **Finir le chargement**

Avec Suspense, nous n'attendons pas que la réponse nous parvienne pour commencer le rendu. En fait, nous commençons le rendu *presque immédiatement* après avoir déclenché la requête réseau :

```
// Ce n'est pas une `Promise`. C'est un objet spécial issu de l'intégration avec
Suspense.
const resource = fetchProfileData();

function ProfilePage() {
  return (
    <Suspense fallback={<h1>Chargement du profil...</h1>}>
      <ProfileDetails />
      <Suspense fallback={<h1>Chargement des publications...</h1>}>
        <ProfileTimeline />
      </Suspense>
    </Suspense>
  );
}
```

```

function ProfileDetails() {
  // Essaie de lire les infos utilisateur, bien qu'elles puissent ne pas être
  // encore chargées
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}

function ProfileTimeline() {
  // Essaie de lire les publications, bien qu'elles puissent ne pas être encore
  // chargées
  const posts = resource.posts.read();
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}

```

Essayez sur CodeSandbox

Voici ce qui se passe quand on affiche `<ProfilePage>` à l'écran :

1. Nous avons déjà déclenché les requêtes dans `fetchProfileData()`. Ça nous a fourni une « ressource » spéciale au lieu d'une `Promise`. En situation réelle, cet objet viendrait de l'intégration Suspense de notre bibliothèque, par exemple Relay.
2. React essaie d'afficher `<ProfilePage>`. Il renvoie `<ProfileDetails>` et `<ProfileTimeline>` comme composants enfants.
3. React essaie d'afficher `<ProfileDetails>`. Il appelle `resource.user.read()`. Aucune donnée n'étant disponible à ce stade, ce composant « se suspend ». React le saute, et essaie d'afficher les autres composants dans l'arborescence.
4. React essaie d'afficher `<ProfileTimeline>`. Il appelle `resource.posts.read()`. Là aussi, faute de données disponibles, le composant « se suspend ». React le saute à son tour, et essaie d'afficher les autres composants dans l'arborescence.
5. Il ne reste rien à afficher. Puisque `<ProfileDetails>` est suspendu, React affiche le contenu de repli (*fallback, NdT*) de l'ancêtre `<Suspense>` le plus proche, à savoir : `<h1>Chargement du profil...</h1>`. Et il en a fini pour l'instant.

Cet objet `resource` représente les données qui ne sont pas encore arrivées, mais devraient à terme être disponibles. Lorsqu'on appelle `read()`, soit on obtient les données, soit le composant « se suspend ».

Au fil de l'arrivée des données, React recommencera le rendu, et chaque fois il pourra peut-être progresser « plus loin ». Lorsque `resource.user` sera chargée, le composant `<ProfileDetails>` pourra être affiché correctement et nous n'aurons plus besoin du contenu de repli `<h1>Chargement du profil...</h1>`. À terme, quand nous aurons toutes les données, il n'y aura plus de contenus de repli à l'écran.

Ce fonctionnement a une conséquence intéressante. Même si nous utilisons un client GraphQL qui regroupe tous nos besoins en données dans une seule requête, *streamer la réponse nous permet d'afficher plus de contenu plus tôt*. Parce que nous faisons le rendu *pendant le chargement* (par opposition à un rendu *après*), si

`user` apparaît dans la réponse avant `posts`, nous serons à même de « déverrouiller » le périmètre `<Suspense>` extérieur avant même que la réponse n'ait été totalement reçue. On ne s'en était pas forcément rendu compte avant, mais même la solution *fetch-then-render* contenait une cascade : entre le chargement et le rendu. Suspense ne souffre pas intrinsèquement de ce type de cascade, et les bibliothèques comme Relay en tirent parti.

Remarquez que nous avons éliminé les vérifications `if (...) « si ça charge »` de nos composants. Il ne s'agit pas juste de retirer du code générique, mais ça facilite aussi les ajustements rapides d'expérience utilisateur. Par exemple, si nous voulions que les détails du profil et les publications « surgissent » toujours d'un bloc, il nous suffirait de retirer le périmètre `<Suspense>` entre eux. Ou nous pourrions les rendre complètement indépendants l'un de l'autre en leur donnant à chacun *leur propre* périmètre `<Suspense>`. Suspense nous permet d'ajuster la granularité de nos états de chargement et d'orchestrer leur séquencement sans avoir à réaliser des changements invasifs dans notre code.

Démarrer le chargement tôt {#start-fetching-early}

Si vous travaillez sur une bibliothèque de chargement de données, il y a un aspect crucial de *render-as-you-fetch* que vous devez bien intégrer. **On déclenche le chargement avant le rendu.** Examinez le code suivant de plus près :

```
// Commence à charger tôt !
const resource = fetchProfileData();

// ...

function ProfileDetails() {
  // Essaie de lire les infos utilisateur
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}
```

Essayez sur CodeSandbox

Remarquez que l'appel à `read()` dans cet exemple ne *déclenche* pas le chargement. Il essaie juste de lire les données qui sont déjà en cours de chargement. Cette distinction est cruciale pour produire des applications rapides avec Suspense. Nous ne voulons pas différer le chargement des données jusqu'au rendu d'un composant. En tant qu'auteur·e de bibliothèque, vous pouvez garantir ça en rendant impossible l'obtention d'un objet `resource` sans déclencher au passage le chargement. Toutes les démos sur cette page qui utilisent notre « API factice » garantissent cet aspect.

Vous pourriez objecter que charger les données « au niveau racine » comme ici n'est guère pratique. Que ferons-nous si nous naviguons vers une autre page de profil ? On pourrait vouloir charger sur base des props. La réponse est que **nous voulons plutôt commencer le chargement dans les gestionnaires d'événements**. Voici un exemple simplifié de la navigation entre des pages utilisateurs :

```
// Premier chargement : aussitôt que possible
const initialResource = fetchProfileData(0);
```

```

function App() {
  const [resource, setResource] = useState(initialResource);
  return (
    <>
      <button onClick={() => {
        const nextUserId = getNextId(resource.userId);
        // Chargement suivant : lorsque l'utilisateur clique
        setResource(fetchProfileData(nextUserId));
      }}>
        Suivant
      </button>
      <ProfilePage resource={resource} />
    </>
  );
}

```

Essayez sur [CodeSandbox](#)

Avec cette approche, on peut **charger le code et les données en parallèle**. Quand on navigue entre les pages, on n'a pas besoin d'attendre le code de la page pour commencer à charger ses données. On peut commencer à charger aussi bien le code que les données au même moment (lors du clic sur le lien), ce qui donne une bien meilleure expérience utilisateur.

La question qui se pose alors est : comment savons-nous *quoi* charger avant d'afficher le prochain écran ? Il y a plusieurs solutions possibles (par exemple, en intégrant le chargement des données au plus près de notre système de routage). Si vous travaillez sur une bibliothèque de chargement de données, [Construire des super expériences utilisateurs avec le mode concurrent et Suspense](#) explore en profondeur les moyens d'accomplir ça en expliquant pourquoi c'est important.

On expérimente encore {#were-still-figuring-this-out}

Suspense lui-même est un mécanisme flexible qui n'impose que peu de contraintes. Le code produit doit se contraindre un peu plus pour être sûr d'éviter les cascades, mais il y a différentes façons de fournir ces garanties. Voici quelques-unes des questions que nous explorons en ce moment :

- Le chargement anticipé peut être lourd à exprimer. Comment faciliter ça en évitant les cascades ?
- Quand on charge les données d'une page, l'API peut-elle encourager l'inclusion de données en vue de transitions instantanées *pour en sortir* ?
- Quel est le délai de péremption d'une réponse ? Le cache doit-il être global ou local ? Qui gère le cache ?
- Les Proxies (*au sens JS, NdT*) peuvent-ils aider à exprimer des API de chargement paresseux sans avoir à coller des appels `read()` partout ?
- À quoi ressemblerait l'équivalent de la composition de requêtes GraphQL pour des données Suspense quelconques ?

Relay a ses propres réponses à certaines de ces questions. Il y a certainement plusieurs façons de s'y prendre, et nous avons hâte de voir quelles nouvelles idées la communauté React va faire émerger.

Suspense et les situations de compétition {#suspense-and-race-conditions}

Les situations de compétition (*race conditions*, *NdT*) sont des bugs qui surviennent suite à des suppositions incorrectes sur l'ordre d'exécution de notre code. On en rencontre souvent lorsqu'on charge des données dans un Hook `useEffect` ou une méthode de cycle de vie comme `componentDidUpdate`. Suspense peut là aussi nous être d'une aide précieuse ; voyons comment.

Pour illustrer le problème, nous allons ajouter un composant racine `<App>` qui affiche notre `<ProfilePage>` avec un bouton nous permettant de **basculer entre différents profils** :

```
function getNextId(id) {
  // ...
}

function App() {
  const [id, setId] = useState(0);
  return (
    <>
      <button onClick={() => setId(getNextId(id))}>
        Suivant
      </button>
      <ProfilePage id={id} />
    </>
  );
}
```

Voyons ensemble la façon dont les différentes stratégies de chargement de données traitent ce besoin.

Compétitions avec `useEffect` {#race-conditions-with-useeffect}

Commençons par une variation de notre exemple antérieur « chargement depuis un effet ». Nous allons le modifier pour passer un paramètre `id` depuis les props de `<ProfilePage>` vers `fetchUser(id)` et `fetchPosts(id)` :

```
function ProfilePage({ id }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetchUser(id).then(u => setUser(u));
  }, [id]);

  if (user === null) {
    return <p>Chargement du profil...</p>;
  }
  return (
    <>
      <h1>{user.name}</h1>
      <ProfileTimeline id={id} />
    </>
  );
}
```

```

        </>
    );
}

function ProfileTimeline({ id }) {
  const [posts, setPosts] = useState(null);

  useEffect(() => {
    fetchPosts(id).then(p => setPosts(p));
  }, [id]);

  if (posts === null) {
    return <h2>Chargement des publications...</h2>;
  }
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}

```

Essayez sur CodeSandbox

Remarquez que nous avons aussi ajusté les dépendances de l'effet, passant de `[]` à `[id]`, car nous voulons que l'effet s'exécute à nouveau si `id` change. Autrement, nous ne chargerions pas les nouvelles données.

Si nous essayons ce code, il peut sembler fonctionner au premier abord. Néanmoins, si nous introduisons un délai de réponse aléatoire dans l'implémentation de notre « API factice » et appuyons suffisamment en rafale sur le bouton « Suivant », nous verrons dans les logs de la console que quelque chose ne tourne pas rond du tout. **Les requêtes associées aux profils précédents répondent parfois après que nous avons changé à nouveau de profil, et du coup écrasent le nouvel état avec une réponse périmée associée à un ID différent.**

Ce problème peut être résolu (on pourrait utiliser la fonction de nettoyage de l'effet pour ignorer voire annuler les requêtes périmées), mais c'est contre-intuitif et difficile à déboguer.

Compétitions avec `componentDidUpdate` `{#race-conditions-with-componentdidupdate}`

On pourrait penser que c'est un problème spécifique à `useEffect` ou aux Hooks. Peut-être que si nous portions ce code vers des classes et utilisions des syntaxes confortables comme `async` / `await`, le problème serait résolu ?

Essayons ça :

```

class ProfilePage extends React.Component {
  state = {
    user: null,
  };

```

```

componentDidMount() {
  this.fetchData(this.props.id);
}
componentDidUpdate(prevProps) {
  if (prevProps.id !== this.props.id) {
    this.fetchData(this.props.id);
  }
}
async fetchData(id) {
  const user = await fetchUser(id);
  this.setState({ user });
}
render() {
  const { id } = this.props;
  const { user } = this.state;
  if (user === null) {
    return <p>Chargement du profil...</p>;
  }
  return (
    <>
      <h1>{user.name}</h1>
      <ProfileTimeline id={id} />
    </>
  );
}
}

class ProfileTimeline extends React.Component {
state = {
  posts: null,
};
componentDidMount() {
  this.fetchData(this.props.id);
}
componentDidUpdate(prevProps) {
  if (prevProps.id !== this.props.id) {
    this.fetchData(this.props.id);
  }
}
async fetchData(id) {
  const posts = await fetchPosts(id);
  this.setState({ posts });
}
render() {
  const { posts } = this.state;
  if (posts === null) {
    return <h2>Chargement des publications...</h2>;
  }
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      )))
    </ul>
  );
}
}

```

```
    );
}
}
```

Essayez sur [CodeSandbox](#)

Ce code est faussement simple à lire.

Malheureusement, ni le recours aux classes ni la syntaxe `async / await` ne nous ont aidés à résoudre le problème. Cette version souffre exactement du même problème de situations de compétition, pour les mêmes raisons.

Le problème {#the-problem}

Les composants React ont leur propre « cycle de vie ». Ils sont susceptibles de recevoir des props ou de mettre à jour leur état à n'importe quel moment. Mais hélas, chaque requête asynchrone a *aussi* son propre « cycle de vie ». Il démarre quand on déclenche le traitement, et se termine quand nous obtenons une réponse. La difficulté que nous rencontrons vient de la « synchronisation » entre différents processus au fil du temps, qui dépendent les uns des autres. Il est difficile d'y réfléchir correctement.

Résoudre les situations de compétition avec Suspense {#solving-race-conditions-with-suspense}

Reprendons à nouveau notre exemple, mais en utilisant seulement Suspense :

```
const initialResource = fetchProfileData(0);

function App() {
  const [resource, setResource] = useState(initialResource);
  return (
    <>
      <button onClick={() => {
        const nextUserId = getNextId(resource.userId);
        setResource(fetchProfileData(nextUserId));
      }}>
        Suivant
      </button>
      <ProfilePage resource={resource} />
    </>
  );
}

function ProfilePage({ resource }) {
  return (
    <Suspense fallback={<h1>Chargement du profil...</h1>}>
      <ProfileDetails resource={resource} />
      <Suspense fallback={<h1>Chargement des publications...</h1>}>
        <ProfileTimeline resource={resource} />
      </Suspense>
    </Suspense>
  );
}
```

```

}

function ProfileDetails({ resource }) {
  const user = resource.user.read();
  return <h1>{user.name}</h1>;
}

function ProfileTimeline({ resource }) {
  const posts = resource.posts.read();
  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.text}</li>
      ))}
    </ul>
  );
}

```

Essayez sur [CodeSandbox](#)

Dans l'exemple Suspense précédent, nous n'avions qu'une `resource`, aussi la placions-nous dans une variable de la portée racine. À présent que nous avons plusieurs ressources, nous les avons déplacées dans l'état local du composant `<App>` racine :

```

const initialResource = fetchProfileData(0);

function App() {
  const [resource, setResource] = useState(initialResource);

```

Quand on clique sur « Suivant », le composant `<App>` déclenche une requête pour le prochain profil, et passe cet *objet-là* au composant `<ProfilePage>` :

```

<>
  <button onClick={() => {
    const nextUserId = getNextId(resource.userId);
    setResource(fetchProfileData(nextUserId));
  }}>
    Suivant
  </button>
  <ProfilePage resource={resource} />
</>

```

Là aussi, remarquez que **nous n'attendons pas la réponse pour modifier l'état. C'est tout l'inverse : nous définissons l'état (et commençons le rendu) immédiatement après avoir déclenché une requête.** Dès que nous aurons plus de données, React « remplira » le contenu dans les composants `<Suspense>`.

Ce code est très lisible, mais contrairement aux exemples précédents, la version Suspense ne souffre pas de situations de compétition. Vous vous demandez peut-être pourquoi. La réponse est que dans la version Suspense, nous n'avons pas besoin de penser aussi intensément au *temps* dans notre code. Le code original, avec ses situations de compétition, avait besoin de modifier l'état *au bon moment ultérieur*, faute de quoi l'état devenait incorrect. Mais avec Suspense, nous définissons l'état *immédiatement* : c'est plus dur d'en corrompre la valeur.

Gérer les erreurs {#handling-errors}

Quand nous écrivons du code à base de promesses ([Promise](#)), nous pouvons recourir à [catch\(\)](#) pour gérer les erreurs. Comment faire avec Suspense, dans la mesure où nous *n'attendons pas* après des promesses pour commencer le rendu ?

Avec Suspense, gérer les erreurs de chargement fonctionne comme la gestion des erreurs de rendu : vous pouvez utiliser un [périmètre d'erreur](#) où bon vous semble pour « attraper » les erreurs dans les composants qu'il enrobe.

Commençons par définir un périmètre d'erreur utilisable dans tout notre projet :

```
// Les périmètres d'erreur exigent pour le moment une définition à base de classe.
class ErrorBoundary extends React.Component {
  state = { hasError: false, error: null };
  static getDerivedStateFromError(error) {
    return {
      hasError: true,
      error
    };
  }
  render() {
    if (this.state.hasError) {
      return this.props.fallback;
    }
    return this.props.children;
  }
}
```

Après quoi nous pouvons le placer où nous voulons dans l'arbre pour attraper les erreurs :

```
function ProfilePage() {
  return (
    <Suspense fallback={<h1>Chargement du profil...</h1>}>
      <ProfileDetails />
      <ErrorBoundary fallback={<h2>La récupération des publications a échoué.</h2>}>
        <Suspense fallback={<h1>Chargement des publications...</h1>}>
          <ProfileTimeline />
        </Suspense>
      </ErrorBoundary>
    </Suspense>
  )
}
```

```
 );  
 }
```

Essayez sur [CodeSandbox](#)

Il attraperait à la fois les erreurs de rendu *et* les erreurs du chargement de données Suspense. Nous pouvons avoir autant de périmètres d'erreur que nous le souhaitons, mais il vaut mieux [bien réfléchir](#) à leurs emplacements.

Prochaines étapes {#next-steps}

Et voilà, nous avons couvert les bases de Suspense pour le chargement de données ! Mais surtout, nous comprenons désormais mieux *pourquoi* Suspense fonctionne comme il le fait, et comment il s'inscrit dans la problématique du chargement de données.

Suspense apporte des réponses, mais pose aussi ses propres questions :

- Si un composant « se suspend », l'appli gèle-t-elle ? Comment éviter ça ?
- Comment faire pour afficher un *spinner* à un endroit autre que « au-dessus » du composant prévu dans l'arbre ?
- Supposons que nous *voulions* explicitement afficher une UI incohérente pendant un bref instant, est-ce possible ?
- Au lieu d'afficher un *spinner*, peut-on ajouter un effet visuel, comme « griser » l'écran en cours ?
- Pourquoi notre [dernier exemple Suspense](#) affiche-t-il un avertissement quand on clique sur le bouton « Suivant » ?

Pour répondre à ces questions, nous vous invitons à lire la prochaine section sur les [Approches pour une UI concurrente](#).

Attention

Cette page décrit **des fonctionnalités expérimentales qui ne sont pas encore disponibles dans une version stable**. Ne vous basez pas sur les builds expérimentaux de React pour vos applis en production. Ces fonctionnalités sont susceptibles d'évoluer de façon significative et sans avertissement avant d'intégrer officiellement React.

Cette documentation est destinée aux personnes curieuses ou habituées à adopter les nouvelles technologies très tôt. **Si vous débutez en React, ne vous préoccupez pas de ces fonctionnalités :** vous n'avez pas besoin de les apprendre pour le moment.

Cette page est une référence de l'API du [mode concurrent](#) de React. Si vous cherchez plutôt un guide introductif, jetez un coup d'œil à [Approches pour une UI concurrente](#).

Remarque : ceci est un Aperçu pour la Communauté et ne constitue pas la version stable finale. Ces API changeront probablement à l'avenir. Ne les utilisez qu'à vos risques et périls !

- [Activer le mode concurrent](#)
 - `createRoot`
 - `createBlockingRoot`
- [API de Suspense](#)
 - `<Suspense>`
 - `<SuspenseList>`
 - `useTransition`
 - `useDeferredValue`

Activer le mode concurrent {#concurrent-mode}

`createRoot` {#createroot}

```
ReactDOM.createRoot(rootNode).render(<App />);
```

Remplace `ReactDOM.render(<App />, rootNode)` et active le mode concurrent.

Pour en savoir plus sur le mode concurrent, consultez la [documentation du mode concurrent](#).

`createBlockingRoot` {#createblockingroot}

```
ReactDOM.createBlockingRoot(rootNode).render(<App />)
```

Remplace `ReactDOM.render(<App />, rootNode)` et active le [mode bloquant](#).

Choisir le mode concurrent introduit des modifications sémantiques dans le fonctionnement de React. Ça signifie que vous ne pouvez pas utiliser le mode concurrent sur seulement certains composants. Pour cette raison, certaines applis risquent de ne pas pouvoir migrer directement vers le mode concurrent.

Le mode bloquant fournit une petite partie des fonctionnalités du mode concurrent, et constitue une étape de migration intermédiaire pour les applis qui ne peuvent malheureusement pas migrer directement.

API de Suspense {#suspense}

<Suspense> {#suspensecomponent}

```
<Suspense fallback={<h1>Chargement...</h1>}>
  <ProfilePhoto />
  <ProfileDetails />
</Suspense>
```

Suspense permet à vos composants « d'attendre » que quelque chose ait lieu avant qu'ils procèdent à leur rendu, en affichant dans l'intervalle une interface utilisateur (UI) de repli.

Dans cet exemple, **ProfileDetails** attend qu'un appel API asynchrone charge des données. Pendant que nous attendons **ProfileDetails** et **ProfilePhoto**, nous affichons le repli **Chargement...** à leur place. Il faut bien comprendre que jusqu'à ce que tous les enfants de **<Suspense>** soient chargés, nous continuerons à afficher l'UI de repli.

Suspense prend deux props :

- **fallback** fournit un indicateur de chargement. Cette UI de repli est affichée jusqu'à ce que les enfants du composant **Suspense** aient fini leur rendu.
- **unstable_avoidThisFallback** prend un booléen. Elle indique à React s'il doit « sauter » la révélation de cette limite (c'est-à-dire le comportement d'attente) lors du chargement initial. Cette API sera probablement retirée dans une version à venir.

<SuspenseList> {#suspenselist}

```
<SuspenseList revealOrder="forwards">
  <Suspense fallback={'Chargement...'}>
    <ProfilePicture id={1} />
  </Suspense>
  <Suspense fallback={'Chargement...'}>
    <ProfilePicture id={2} />
  </Suspense>
  <Suspense fallback={'Chargement...'}>
    <ProfilePicture id={3} />
  </Suspense>
  ...
</SuspenseList>
```

SuspenseList aide à orchestrer la révélation progressive de composants susceptibles d'être suspendus.

Lorsque plusieurs composants ont besoin de charger des données, celles-ci peuvent arriver dans un ordre imprévisible. Cependant, si vous enveloppez ces éléments dans un **SuspenseList**, React ne montrera un élément

de la liste qu'une fois que tous les éléments qui le précédent auront été affichés (ce comportement est d'ailleurs ajustable).

`SuspenseList` prend deux props :

- `revealOrder ('forwards', 'backwards', 'together')` indique dans quel ordre les enfants de la `SuspenseList` doivent être révélés.
 - `'together'` les révèle *tous* d'un coup une fois qu'ils sont prêts, au lieu de le faire individuellement.
- `tail ('collapsed', 'hidden')` indique comment afficher les éléments non chargés dans une `SuspenseList`.
 - Par défaut, `SuspenseList` affichera toutes les UI de repli dans la liste.
 - `'collapsed'` affiche uniquement le repli du prochain élément dans la liste.
 - `'hidden'` n'affiche aucun élément non chargé.

Remarquez que `SuspenseList` n'opère que sur les composants enfants `Suspense` et `SuspenseList` les plus proches d'elle. Elle ne recherche pas les périmètres à plus d'un niveau de profondeur. Ceci dit, il est possible d'imbriquer plusieurs composants `SuspenseList` pour construire des grilles.

`useTransition` {#usetransition}

```
const SUSPENSE_CONFIG = { timeoutMs: 2000 };

const [startTransition, isPending] = useTransition(SUSPENSE_CONFIG);
```

`useTransition` permet aux composants d'éviter des états de chargement indésirables en attendant que le contenu soit chargé avant de **transiter vers le prochain écran**. Il permet aussi aux composants de différer des chargements de données plus lents vers des rendus ultérieurs afin que les mises à jour les plus cruciales puissent être affichées immédiatement.

Le hook `useTransition` renvoie deux valeurs dans un tableau.

- `startTransition` est une fonction qui prend une fonction de rappel. Nous pouvons l'utiliser pour indiquer à React quel état nous souhaitons différer.
- `isPending` est un booléen, grâce auquel React nous indique si nous sommes en train d'attendre la fin de la transition.

Si une mise à jour donnée de l'état entraîne la suspension d'un composant, cette mise à jour devrait être enrobée dans une transition.

```
const SUSPENSE_CONFIG = { timeoutMs: 2000 };

function App() {
  const [resource, setResource] = useState(initialResource);
  const [startTransition, isPending] = useTransition(SUSPENSE_CONFIG);
  return (
    <>
      <button>
```

```

disabled={isPending}
onClick={() => {
  startTransition(() => {
    const nextUserId = getNextId(resource.userId);
    setResource(fetchProfileData(nextUserId));
  });
}}
>
  Suivant
</button>
{isPending ? "Chargement..." : null}
<Suspense fallback={<Spinner />}>
  <ProfilePage resource={resource} />
</Suspense>
</>
);
}

```

Dans ce code, nous avons enrobé notre chargement de données avec `startTransition`. Ça nous permet de commencer immédiatement à charger les données du profil, tout en différant le rendu de la prochaine page de profil et de son `Spinner` associé pendant 2 secondes (le temps indiqué par `timeoutMs`).

Le booléen `isPending` est fourni par React pour nous indiquer que notre composant est en cours de transition, ce qui nous permet d'avertir l'utilisateur en affichant un texte de chargement au sein de la précédente page de profil.

Pour une exploration en profondeur des transitions, vous pouvez lire les [Approches pour une UI concurrente](#).

Configuration de `useTransition` {#usetransition-config}

```
const SUSPENSE_CONFIG = { timeoutMs: 2000 };
```

`useTransition` accepte une **configuration Suspense optionnelle** avec un champ `timeoutMs`. Ce délai d'expiration (en millisecondes) indique à React combien de temps attendre avant d'afficher le prochain état (dans l'exemple ci avant, ce serait la prochaine page de profil).

Remarque : nous vous conseillons de partager une unique configuration Suspense entre vos différents modules.

`useDeferredValue` {#usedeferredvalue}

```
const deferredValue = useDeferredValue(value, { timeoutMs: 2000 });
```

Renvoie une version différée de la valeur qui est susceptible d'être « en retard » pour un temps maximum de `timeoutMs`.

On utilise couramment ça pour préserver la réactivité de l'interface, avec des affichages immédiats suite à des saisies par l'utilisateur·rice malgré le besoin d'attendre un chargement de données.

La saisie de texte constitue un bon exemple :

```
function App() {
  const [text, setText] = useState("bonjour");
  const deferredText = useDeferredValue(text, { timeoutMs: 2000 });

  return (
    <div className="App">
      {/* Continue à passer le texte actuel au champ */}
      <input value={text} onChange={handleChange} />
      ...
      {/* Mais la liste des résultats est autorisée à « être en retard » si nécessaire */}
      <MySlowList text={deferredText} />
    </div>
  );
}
```

Ça nous permet de commencer à afficher le nouveau texte du `input` immédiatement, ce qui donne un sentiment de réactivité pour la page web. Dans le même temps, la mise à jour de `MySlowList` peut « retarder » à hauteur de 2 secondes en vertu du `timeoutMs`, ce qui lui permet de réaliser son rendu adapté au texte courant en arrière-plan.

Vous trouverez une exploration en profondeur des valeurs différées dans les [Approches pour une UI concurrente](#).

Configuration de `useDeferredValue` {#usedeferredvalue-config}

```
const SUSPENSE_CONFIG = { timeoutMs: 2000 };
```

`useDeferredValue` accepte une **configuration Suspense optionnelle** avec un champ `timeoutMs`. Ce délai d'expiration (en millisecondes) indique à React pendant combien de temps la valeur différée est autorisée à retarder.

React essaiera toujours de minimiser le retard lorsque le réseau et l'appareil le permettent.



Fromulaires

Contrôle des composants de formulaire.

Les éléments de formulaire HTML fonctionnent un peu différemment des autres éléments du DOM dans React, parce que les éléments de formulaire gardent naturellement un état interne.

Par exemple, ce formulaire en HTML accepte un seul nom:

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Un formulaire HTML possède un comportement par défaut (soumission).

Composants contrôlés

En HTML, les éléments de formulaire tels que `<input>`, `<textarea>` et `<select>` maintiennent généralement leur propre état et le mettent à jour en automatiquement.

Dans React, l'état mutable est généralement maintenu dans la propriété de `state` des composants, et seulement mis à jour avec `setState()`.

Un élément de formulaire dont la valeur est contrôlée par React de cette façon est appelé «composant contrôlé».

```

class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

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

```

[Essayez-le sur CodePen.](#)

La balise textarea

En HTML, un élément `<textarea>` définit son texte par ses enfants:

```

<textarea>
  Hello there, this is some text in a text area
</textarea>

```

Dans React, un `<textarea>` utilise un attribut `value` à la place. De cette façon, une formulaire utilisant un `<textarea>` peut être écrit:

```

class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}

```

Notez que `this.state.value` est initialisée dans le constructeur, de sorte que la zone de texte commence avec un texte définit.

La balise select

`<select>` crée une liste déroulante. Par exemple, ce code HTML crée une liste déroulante de saveurs:

```

<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>

```

Notez que l'option de `coconut` est initialement sélectionnée, en raison de l'attribut `selected`. React, au lieu d'utiliser cet attribut `selected`, utilise un attribut `value` sur la balise `select` racine.

Ceci est plus commode dans un composant contrôlé parce que vous avez seulement besoin de le mettre à jour en un seul endroit.

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite La Croix flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

[Essayez-le sur CodePen.](#)

Manipulation Entrées multiples

Lorsque vous avez besoin de gérer plusieurs éléments `input`, vous pouvez ajouter un attribut `name` à chaque élément et laisser la fonction de gestionnaire choisir ce qu'il faut faire sur la base de la valeur de `event.target.name`:

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

[Essayez-le sur CodePen.](#)

Notez la syntaxe ES6 pour mettre à jour la clé de l'état correspondant au nom d'entrée donné :

```
this.setState({  
  [name]: value  
});
```

Il est équivalent à ce code ES5:

```
var partialState = {};  
partialState[name] = value;  
this.setState(partialState);
```

Alternatives aux composants contrôlés

Il peut parfois être fastidieux d'utiliser des composants contrôlés, car vous avez besoin d'écrire un gestionnaire d'événements pour tout changement de données.

Vous pouvez utiliser une `ref` pour lier une donnée provenant du DOM

```
class NameForm extends React.Component {  
  constructor(props) {  
    super(props);  
    this.handleSubmit = this.handleSubmit.bind(this);  
  }  
  
  handleSubmit(event) {  
    alert('A name was submitted: ' + this.input.value);  
    event.preventDefault();  
  }  
  
  render() {  
    return (  
      <form onSubmit={this.handleSubmit}>  
        <label>  
          Name:  
          <input type="text"  
            defaultValue="Bob" //specifies un default value  
            ref={(input) => this.input = input} />  
        </label>  
        <input type="submit" value="Submit" />  
      </form>  
    );  
  }  
}
```

Manipulation du DOM.

React met en œuvre un système de DOM indépendant du navigateur pour la compatibilité des performances et cross-browser. Nous en avons profité pour nettoyer quelques bords rugueux dans les implémentations navigateur DOM.

Dans React, toutes les propriétés DOM et attributs (y compris les gestionnaires d'événements) devraient être en notation CamelCase.

Par exemple, l'attribut HTML `tabindex` correspond à l'attribut `tabIndex` dans React. A l'exception des attributs `aria-*` et `data-*`, qui devraient être minuscule.

Différences dans les attributs

Il y a un certain nombre d'attributs qui fonctionnent différemment entre React et HTML:

Checked

L'attribut `checked` est pris en charge par les composants `<input>` de type `checkbox` ou `radio`. Vous pouvez l'utiliser pour définir si le composant est coché.

Ceci est utile pour la construction de composants contrôlés. `defaultChecked` est l'équivalent non contrôlé, qui définit si le composant est coché quand il est monté en premier.

className

Pour spécifier une classe CSS, utilisez l'attribut `className`. Cela vaut pour tous les éléments DOM et SVG réguliers comme `<div>`, `<a>`, et d'autres.

Si vous utilisez React avec des composants Web (ce qui est rare), utilisez `class` à la place.

dangerouslySetInnerHTML

`dangerouslySetInnerHTML` est de le remplacement pour l'utilisation de `innerHTML`.

Ainsi, vous pouvez définir HTML directement à partir React, mais vous devez taper `dangerouslySetInnerHTML` et passer un objet avec une clé `__html`, pour vous rappeler qu'il est dangereux.

```

function createMarkup() {
  return {__html: 'First &nbsp; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}

```

htmlFor

Puisque `for` est un mot réservé en JavaScript, les éléments React utilisent `htmlFor` place.

style

L'attribut de `style` accepte un objet JavaScript avec des propriétés écrits en notation CamelCase plutôt que d'une chaîne de CSS.

```

const divStyle = {
  color: 'blue',
  backgroundImage: `url(${ imgUrl })`,
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}

```

Notez que les styles ne sont pas autoprefixed. Pour prendre en charge les navigateurs plus anciens, vous devez fournir des propriétés de style correspondantes:

```

const divStyle = {
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix
};

function ComponentWithTransition() {
  return <div style={divStyle}>This should work cross-browser</div>;
}

```

suppressContentEditableWarning

Normalement, il y a un avertissement quand un élément avec les enfants est également marqué comme `contentEditable`, parce que cela ne fonctionnera pas. Cet attribut supprime cet avertissement.

value

L'attribut `value` est pris en charge par les balises `<input>` et `<textarea>`. Vous pouvez l'utiliser pour définir la valeur du composant. Ceci est utile pour la construction de composants contrôlés.

`defaultValue` est équivalent non contrôlé, qui fixe la valeur du composant lorsqu'il est monté en premier.

Tous les attributs HTML pris en charge

React supporte les attributs `data-*` et `aria-*`, [ainsi que ces attributs](#)

En outre, les attributs non standard suivants sont supportés:

- `autoCapitalize` `autoCorrect` Mobile Safari.
- `color` pour `<link rel="mask-icon"/>` dans Safari.
- `itemProp` `itemScope` `itemType` `itemRef` `itemID` pour les [micro data](#).
- `security` pour les anciennes versions d'Internet Explorer.
- `unselectable` pour Internet Explorer.
- `results` `autoSave` pour les champs de type `search` de WebKit/Blink.

Tous les attributs SVG pris en charge



Code Isomorphique

Application isomorphe

La performance

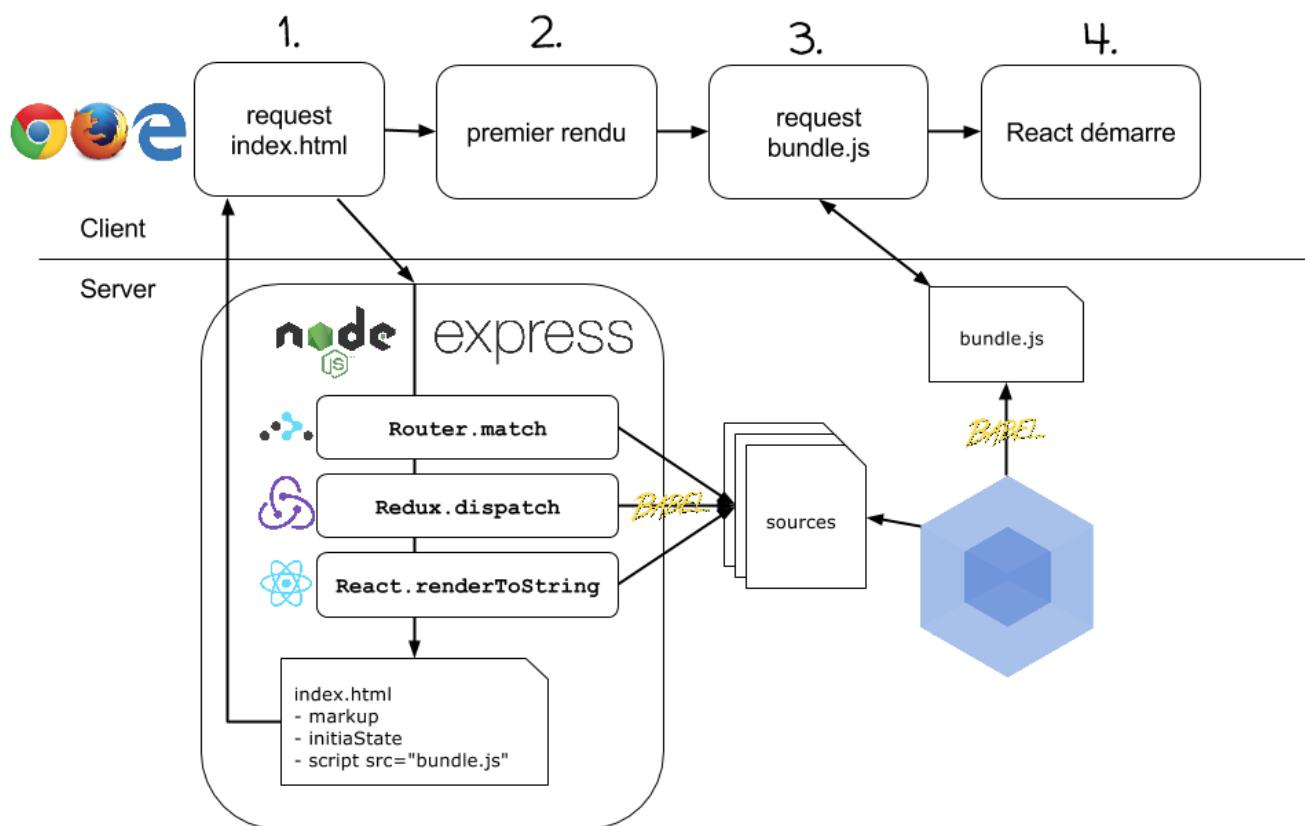
Cycle de d'initialisation d'une SPA coté client :

1. Chargement du fichier HTML

Chargement des différents Assets (Css, image, scripts JS application et librairies/frameworks)

Délai : Parsing / Execution.

* Délai : Démarrage et configuration de l'état.



Principe et bénéfices du développement isomorphe.

Le mot *isomorphisme* vient des racines grecques *isos* pour égal et *morph* pour forme.

L'isomorphisme désigne donc **deux entités de même forme dans un contexte différent**.

En informatique, et plus particulièrement dans le domaine du développement web, on dit qu'une **application est isomorphe lorsqu'elle partage le même code coté client (navigateur) et coté serveur**.

Concrètement, les avantages de l'isomorphisme sont multiples:

- Un seul code, pour toute une application (serveur et client)
- Les moteurs de recherche voient le contenu (plus totalement vrai: Google interprète le JavaSscript)
- Rapide, plus nécessaire d'attendre le téléchargement du JS pour voir la page.
- Plus facile à maintenir (une seule codebase)
- Un état identique (partagé) entre client et serveur = un debug plus simple.

Cependant, mettre en place de l'isomorphisme “from-scratch” n'est pas simple, c'est pourquoi il est souvent nécessaire d'utiliser un framework qui supporte ce mode de fonctionnement.

React et l'isomorphisme

Voici un exemple d'un code en JSX et de son équivalent en utilisant la notation classique de React (en ES6).

Il est possible d'utiliser 3 notations bien distinctes.

```

// Création de l'élément en JSX
class Button extends React.Component {
    render() {
        return (
            <button className={"button"}>
                <b>OK!</b>
            </button>
        );
    }
}

// Création de l'élément à l'aide des "helpers" React
class Button extends React.Component {
    render() {
        return React.createElement("button", { className: "button" },
            React.createElement("b", {}, "OK!")
        )
    }
}

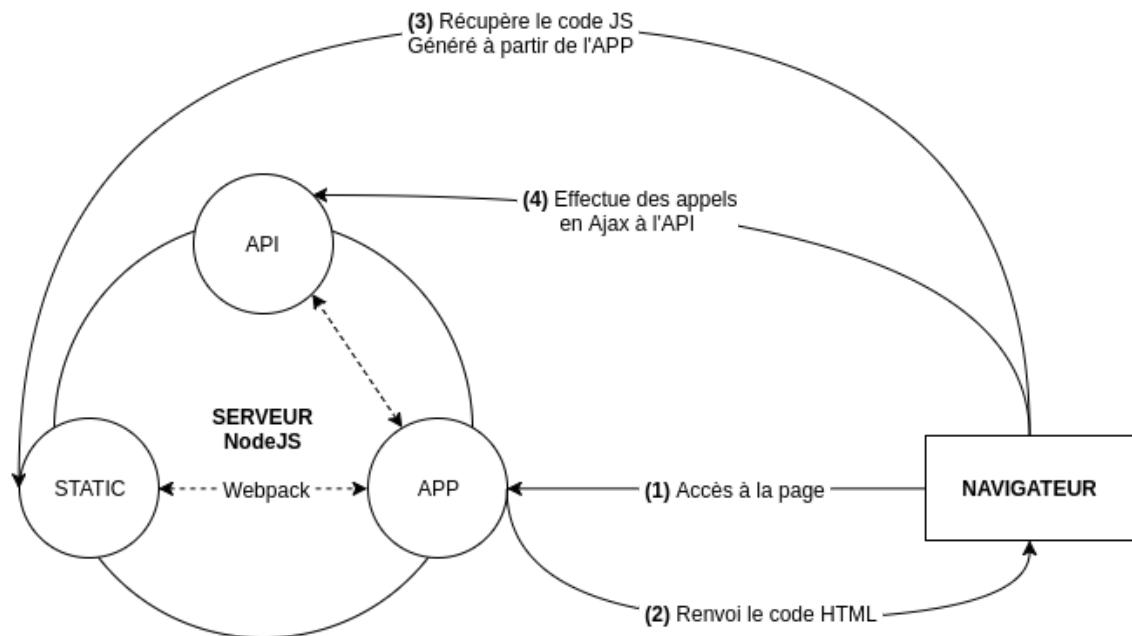
// Création de l'élément en "brut" (objet JS)
class Button extends React.Component {
    render() {
        return {
            type: 'button',
            props: {
                className: 'button',
                children: {
                    type: 'b',
                    props: {
                        children: 'OK!'
                    }
                }
            }
        };
    }
}

```

Ecosystème du JavaScript côté serveur.

L'utilisation d'un “DOM” virtuel permet à React d'être totalement “context agnostic”, ce qui lui permet de générer un rendu coté serveur

Workflow d'application isomorphique.



Examen du code source

L'objet `ReactDOMServer` vous permet de produire sous forme de texte statique le balisage nécessaire à l'affichage de composants. En règle générale, on l'utilise avec un serveur Node :

```
// Modules ES
import ReactDOMServer from 'react-dom/server';
// CommonJS
var ReactDOMServer = require('react-dom/server');
```

Aperçu {#overview}

Les méthodes suivantes peuvent être utilisées aussi bien dans des environnements navigateurs que serveurs :

- `renderToString()`
- `renderToStaticMarkup()`

Les méthodes suivantes dépendent d'un module (`stream`) **disponible uniquement côté serveur**, elles ne fonctionneront donc pas dans un navigateur.

- `renderToNodeStream()`
- `renderToStaticNodeStream()`

Référence de l'API {#reference}

`renderToString()` {#rendertostring}

```
ReactDOMServer.renderToString(element)
```

Produit le HTML initial d'un élément React, sous forme d'une chaîne de caractères. Vous pouvez utiliser cette méthode pour générer du HTML côté serveur et le renvoyer en réponse à la requête initiale, afin d'accélérer le chargement des pages et de permettre aux moteurs de recherche d'analyser vos pages dans une optique de référencement naturel (*SEO, NdT*).

Si vous appelez `ReactDOM.hydrate()` sur un nœud dont le balisage a déjà été généré par le serveur, React le conservera et se contentera d'y attacher les gestionnaires d'événements, ce qui vous permettra d'avoir une expérience de chargement initial des plus performantes.

`renderToStaticMarkup()` {#rendertostaticmarkup}

```
ReactDOMServer.renderToStaticMarkup(element)
```

Similaire à `renderToString`, si ce n'est qu'elle ne crée pas d'attributs supplémentaires utilisés par React en interne, tels que `data-reactroot`. Ça peut être pratique si vous souhaitez utiliser React comme simple

générateur de pages statiques, car supprimer les attributs supplémentaires économise quelques octets.

N'utilisez pas cette méthode si vous envisagez d'utiliser React côté client pour rendre le contenu interactif.

Préférez `renderToString` côté serveur, et `ReactDOM.hydrate()` côté client.

`renderToNodeStream()` {#rendertonodestream}

```
ReactDOMServer.renderToNodeStream(element)
```

Produit le HTML initial d'un élément React. Renvoie un [flux en lecture \(Readable\)](#) qui génère une chaîne de caractères HTML. La sortie HTML de ce flux est identique à ce que `ReactDOMServer.renderToString` renverrait. Vous pouvez utiliser cette méthode pour générer du HTML côté serveur et le renvoyer en réponse à la requête initiale, afin d'accélérer le chargement des pages et de permettre aux moteurs de recherche d'analyser vos pages dans une optique de référencement naturel.

Si vous appelez `ReactDOM.hydrate()` sur un nœud dont le balisage a déjà été généré par le serveur, React le conservera et se contentera d'y attacher les gestionnaires d'événements, ce qui vous permettra d'avoir une expérience de chargement initial des plus performantes.

Remarque

Côté serveur uniquement. Cette API n'est pas disponible côté navigateur.

Le flux renvoyé par cette méthode est encodé en UTF-8. Si vous avez besoin d'un autre encodage, jetez un œil au projet [iconv-lite](#), qui fournit des flux de transformation pour le transcodage de texte.

`renderToStaticNodeStream()` {#rendertostaticnodestream}

```
ReactDOMServer.renderToStaticNodeStream(element)
```

Similaire à `renderToNodeStream`, si ce n'est qu'elle ne crée pas d'attributs supplémentaires utilisés par React en interne, tels que `data-reactroot`. Ça peut être pratique si vous souhaitez utiliser React comme simple générateur de pages statiques, car supprimer les attributs supplémentaires économise quelques octets.

La sortie HTML de ce flux est identique à ce que `ReactDOMServer.renderToStaticMarkup` renverrait.

N'utilisez pas cette méthode si vous envisagez d'utiliser React côté client pour rendre le contenu interactif. Préférez `renderToNodeStream` côté serveur, et `ReactDOM.hydrate()` côté client.

Remarque

Côté serveur uniquement. Cette API n'est pas disponible côté navigateur.

Le flux renvoyé par cette méthode est encodé en UTF-8. Si vous avez besoin d'un autre encodage, jetez un œil au projet [iconv-lite](#), qui fournit des flux de transformation pour le transcodage de texte.

