

REACT Docs - Français

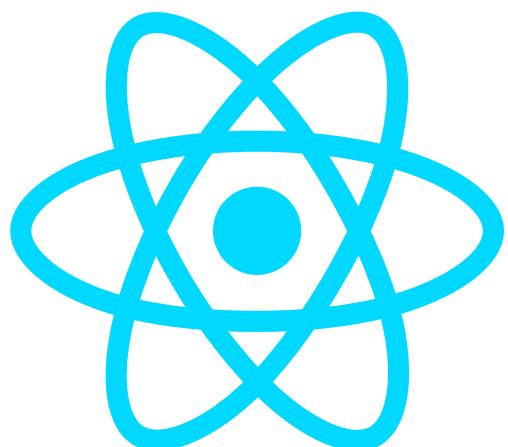


Table of contents

• Accessibility	5
• Add react /to /a /website	18
• Addons animation	23
• Addons create /fragment	30
• Addons perf	32
• Addons pure /render /mixin	36
• Addons shallow /compare	37
• Addons shallow /renderer	38
• Addons test /utils	40
• Addons two /way /binding /helpers	47
• Addons update	50
• Addons	53
• Cdn links	55
• Code splitting	56
• Codebase overview	61
• Components and /props	66
• Composition vs /inheritance	71
• Concurrent mode /adoption	75
• Concurrent mode /intro	79
• Concurrent mode /patterns	83
• Concurrent mode /reference	104
• Concurrent mode /suspense	109
• Conditional rendering	126
• Context	131
• Create a /new /react /app	137
• Cross origin /errors	140
• Design principles	142
• Error boundaries	149
• Error decoder	154
• Faq ajax	155
• Faq build	158
• Faq functions	159
• Faq internals	167
• Faq state	168

• Faq structure	171
• Faq styling	173
• Faq versioning	175
• Forms	178
• Forwarding refs	184
• Fragments	187
• Getting started	190
• Handling events	194
• Hello world	197
• Higher order /components	199
• Hooks custom	208
• Hooks effect	214
• Hooks faq	226
• Hooks intro	248
• Hooks overview	253
• Hooks reference	260
• Hooks rules	273
• Hooks state	276
• How to /contribute	283
• Implementation notes	289
• Integrating with /other /libraries	307
• Introducing jsx	317
• Jsx in /depth	321
• Legacy context	330
• Lifting state /up	335
• Lists and /keys	343
• Optimizing performance	349
• Portals	359
• React without /es6	362
• React without /jsx	367
• Reconciliation	369
• Reference dom /elements	374
• Reference events	379
• Reference glossary	386
• Reference javascript /environment /requirements	391
• Reference profiler	392

• Reference pure /render /mixin	395
• Reference react /component	397
• Reference react /dom /server	413
• Reference react /dom	416
• Reference react	420
• Reference test /renderer	429
• Refs and /the /dom	435
• Release channels	442
• Render props	446
• Rendering elements	453
• State and /lifecycle	456
• Static type /checking	465
• Strict mode	473
• Testing environments	478
• Testing recipes	481
• Testing	493
• Thinking in /react	495
• Typechecking with /proptypes	501
• Uncontrolled components	505
• Web components	507

Accessibilité

Pourquoi l'accessibilité ?

L'accessibilité du web (aussi désignée par la contraction [a11y](#)) concerne le design et la création de sites web qui peuvent être utilisés par tout le monde. La prise en charge de l'accessibilité est nécessaire pour permettre aux technologies d'assistance d'interpréter les pages web.

React prend totalement en charge la création de sites web accessibles, en s'appuyant souvent sur des techniques HTML standard.

Standards et lignes directrices

WCAG

Les [directives pour l'accessibilité au contenu web](#) (*Web Content Accessibility Guidelines, ou WCAG, NdT*) proposent des lignes directrices pour créer des sites web accessibles.

Les listes de contrôle du WCAG ci-dessous en fournissent un aperçu :

- [La liste de contrôle WCAG de Wuhcag](#) (en anglais).
- [La liste de contrôle WCAG de WebAIM](#) (en anglais, une traduction est disponible [ici](#)).
- [La liste de contrôle de The A11Y Project](#) (en anglais).

WAI-ARIA

Le document de [l'Initiative d'Accessibilité du Web - Applications Internet Riches Accessibles](#) (*Web Accessibility Initiative - Accessible Rich Internet Applications, ou WAI-ARIA, NdT*) contient les techniques nécessaires à la création d'éléments d'interface JavaScript complètement accessibles.

Remarquez que tous les attributs HTML `aria-*` sont autorisés dans JSX. Là où la plupart des propriétés DOM en React utilisent la casse *camelCase*, ces attributs doivent être en minuscules avec des tirets (casse connue sous les noms *hyphen-case*, *kebab-case*, *lisp-case*, etc.), comme ils le sont en HTML brut :

```
<input
  type="text"
  aria-label={labelText}
  aria-required="true"
  onChange={onchangeHandler}
  value={inputValue}
  name="name"
/>
```

HTML sémantique

Le HTML sémantique est la base de l'accessibilité dans une application web. L'utilisation des différents éléments HTML pour renforcer la signification de l'information de nos sites web améliore généralement l'accessibilité à peu de frais.

- [La référence des éléments HTML sur MDN](#).

Il nous arrive de casser la sémantique HTML en ajoutant des balises `<div>` dans notre JSX afin de faire fonctionner notre code React, en particulier lorsqu'on travaille avec des listes (``, `` et `<dl>`) ou des tableaux HTML `<table>`. Dans ces cas-là, nous devrions plutôt utiliser les [fragments React](#) pour regrouper de multiples éléments.

Par exemple,

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

function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
  );
}

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <ListItem item={item} key={item.id} />
      ))}
    </dl>
  );
}
```

Vous pouvez représenter une liste d'éléments comme un tableau de fragments, comme vous le feriez avec tout autre type d'élément :

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Les fragments doivent aussi disposer de la propriété `key`
        // lors de l'itération sur les listes.
        <Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}
```

Lorsque vous n'avez pas besoin de définir de propriété sur la balise fragment, vous pouvez utiliser la [syntaxe courte](#), si votre outillage la prend en charge :

```
function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}
```

Pour plus d'informations, consultez [la documentation sur les fragments](#).

Formulaires accessibles

L'étiquetage

Tous les champs de formulaire HTML, tels que `<input>` et `<textarea>`, doivent être étiquetés de manière accessible. Nous devons fournir des étiquettes descriptives, qui sont également exposées aux lecteurs d'écran.

Les ressources ci-dessous nous montrent comment procéder :

- [Le W3C nous montre comment étiqueter les éléments](#) (en anglais).
- [Le WebAIM nous montre comment étiqueter les éléments](#) (en anglais).
- [Le groupe Paciello explique ce qu'est un nom accessible](#) (en anglais).

Bien que ces pratiques HTML standard soient directement utilisables dans React, il faut noter que l'attribut `for` est écrit `htmlFor` en JSX :

```
<label htmlFor="namedInput">Nom :</label>
<input id="namedInput" type="text" name="name"/>
```

Notifier des erreurs à l'utilisateur

Les situations d'erreur doivent être comprises par tous les utilisateurs. Les liens ci-dessous montrent comment exposer les textes d'erreur aux lecteurs d'écran :

- [Le W3C présente les notifications utilisateur](#) (en anglais).
- [Le WebAIM se penche sur la validation de formulaire](#) (en anglais).

Contrôle du focus

Assurez-vous que votre application web peut être complètement utilisable avec le clavier seul :

- [Le WebAIM parle de l'accessibilité depuis le clavier](#) (en anglais).

Focus clavier et contour de focus

Le focus clavier fait référence à l'élément courant dans le DOM qui est sélectionné pour accepter les saisies au clavier. Nous le voyons partout comme un contour similaire à ce qu'on voit sur l'image ci-dessous :



N'utilisez CSS pour retirer ce contour, par exemple en définissant `outline: 0`, que si vous le remplacez par une autre implémentation de contour de focus.

Mécanismes pour sauter au contenu désiré

Fournissez un mécanisme permettant aux utilisateurs de sauter les sections de navigation dans votre application, car ça facilite et accélère la navigation au clavier.

Les liens d'évitement (*skiplinks*, *NdT*) sont des liens de navigation cachés qui ne sont visibles que lorsque l'utilisateur interagit au clavier avec la page. Ils sont très faciles à mettre en œuvre avec les ancrées de pages internes et un peu de style :

- [WebAIM - Liens d'évitement](#) (en anglais).

Utilisez également les éléments et rôles de repérage, tels que `<main>` et `<aside>`, afin de délimiter les régions de la page, car les technologies d'assistance permettent à l'utilisateur de naviguer rapidement vers ces sections.

Pour en apprendre davantage sur l'utilisation de ces éléments afin d'améliorer l'accessibilité, rendez-vous sur :

- [Repères d'accessibilité](#) (en anglais).

Gérer le focus programmatiquement

Nos applications React modifient en continu le DOM HTML au cours de l'exécution, ce qui entraîne parfois la perte du focus clavier ou le positionne sur un élément inattendu. Pour corriger ça, nous devons déplacer le focus clavier programmatiquement dans la bonne direction. On peut par exemple redonner le focus clavier à un bouton qui ouvre une fenêtre modale, lorsque cette dernière se referme.

La documentation web du MDN se penche sur ça et décrit comment nous pouvons construire [des éléments d'interface JavaScript permettant la navigation au clavier](#).

Afin de définir le focus en React, nous pouvons utiliser les [Refs aux éléments du DOM](#).

Nous créons d'abord une ref à un élément du JSX de la classe du composant :

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // Créez une référence pour conserver l'élément textInput du DOM.
```

```

    this.TextInput = React.createRef();
}
render() {
  // Utilisez la prop `ref` pour définir la valeur courante
  // de la ref `textInput` à l'élément DOM
  return (
    <input
      type="text"
      ref={this.TextInput}
    />
  );
}
}

```

Ensuite, nous pouvons lui donner le focus depuis n'importe où dans notre composant en cas de besoin :

```

focus() {
  // Donnez explicitement le focus au champ de saisie en utilisant l'API du DOM.
  // Remarque : nous utilisons "current" pour accéder au nœud du DOM.
  this.TextInput.current.focus();
}

```

Il arrive parfois qu'un composant parent ait besoin de définir le focus sur un élément au sein d'un composant enfant. Pour ce faire, nous [exposons des refs DOM aux composants parents](#) via une propriété spéciale sur le composant enfant qui transfère la ref du parent au nœud DOM de l'enfant.

```

function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.inputElement = React.createRef();
  }
  render() {
    return (
      <CustomTextInput inputRef={this.inputElement} />
    );
  }
}

// Maintenant, vous pouvez définir le focus quand vous en avez besoin.
this.inputElement.current.focus();

```

Quand vous enrobez des composants à l'aide d'un composant d'ordre supérieur (*Higher-Order Component, ou HOC, NdT*), il est recommandé de [transférer la référence](#) vers l'élément enrobé grâce à la fonction `forwardRef` de React. Si un HOC tiers n'implémente pas le transfert de référence, le modèle ci-dessus peut être utilisé comme solution de secours.

Le composant [react-aria-modal](#) est un excellent exemple de la gestion du focus. Il s'agit de l'un des rares exemples de fenêtre modale complètement accessible. Non seulement il définit le focus initial sur le bouton d'annulation (empêchant l'utilisateur du clavier d'activer accidentellement l'action de succès), mais il restreint bien le focus clavier à l'intérieur de la fenêtre modale et il remet à terme le focus sur l'élément qui a originellement déclenché la fenêtre modale.

Remarque

Bien qu'il s'agisse d'une caractéristique d'accessibilité très importante, c'est également une technique qui doit être utilisée à bon escient. Utilisez-la pour corriger le comportement du focus clavier lorsqu'il est perturbé, et non pour essayer d'anticiper la manière dont les utilisateurs souhaitent utiliser les applications.

Événements de souris et de pointeur

Assurez-vous que toutes les fonctionnalités exposées via un événement de souris ou de pointeur sont également accessibles avec le clavier seul. Ne dépendre que du pointeur peut aboutir à de nombreuses situations où les utilisateurs de clavier ne pourront pas utiliser votre application.

Pour illustrer ça, examinons un exemple courant où l'accessibilité est cassée par les événements de clics. Il s'agit du modèle de clic extérieur dans lequel un utilisateur peut désactiver une liste déroulante en cliquant à l'extérieur de l'élément.

Select an option

Load the option

Remove the option

C'est généralement implémenté en écoutant l'événement `click` de l'objet `window` pour fermer le menu déroulant :

```
class OuterClickExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }
```

```

componentDidMount() {
  window.addEventListener('click', this.onClickOutsideHandler);
}

componentWillUnmount() {
  window.removeEventListener('click', this.onClickOutsideHandler);
}

onClickHandler() {
  this.setState(currentState => ({
    isOpen: !currentState.isOpen
  }));
}

onClickOutsideHandler(event) {
  if (this.state.isOpen && !this.toggleContainer.current.contains(event.target))
  {
    this.setState({ isOpen: false });
  }
}

render() {
  return (
    <div ref={this.toggleContainer}>
      <button onClick={this.onClickHandler}>Choisissez une option</button>
      {this.state.isOpen && (
        <ul>
          <li>Option 1</li>
          <li>Option 2</li>
          <li>Option 3</li>
        </ul>
      )}
    </div>
  );
}
}

```

Ça fonctionne peut-être pour les utilisateurs ayant des dispositifs de pointage, tels qu'une souris, mais le fait de n'utiliser que le clavier entraîne des dysfonctionnements lors de la tabulation sur l'élément suivant, car l'objet `window` ne reçoit jamais d'événement `click`. Ça peut finir par masquer des fonctionnalités, ce qui empêche les utilisateurs d'utiliser votre application.

Select an option

Load the option

Remove the option

La même fonctionnalité peut être obtenue en utilisant les gestionnaires d'événements appropriés, tels que `onBlur` et `onFocus` :

```

class BlurExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.timeOutId = null;

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  // Nous fermons le menu déroulant au prochain tick en utilisant setTimeout.
  // C'est nécessaire car nous devons d'abord vérifier si un
  // autre enfant de l'élément a reçu le focus car l'événement
  // `blur` se déclenche avant le nouvel événement de focus.
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({
        isOpen: false
      });
    });
  }

  // Si un enfant reçoit le focus, alors on ne ferme pas le menu déroulant.
  onFocusHandler() {
    clearTimeout(this.timeOutId);
  }

  render() {
    // React nous aide en assurant la propagation des
    // événements `blur` et `focus` vers le parent.
    return (
      <div onBlur={this.onBlurHandler}
            onFocus={this.onFocusHandler}>
        <button onClick={this.onClickHandler}
                  aria-haspopup="true"
                  aria-expanded={this.state.isOpen}>
          Choisissez une option
        </button>
        {this.state.isOpen && (
          <ul>
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        )}
      </div>
    );
  }
}

```

Ce code expose la fonctionnalité aussi bien aux utilisateurs de dispositifs de pointage qu'aux utilisateurs de clavier. Remarquez également les propriétés `aria-*` ajoutées afin de prendre en charge les lecteurs d'écran. Par souci de simplicité, les événements clavier permettant l'interaction avec les options du menu déroulant via les touches de curseur n'ont pas été implémentés.

Select an option

Load the option

Remove the option

C'est un exemple des nombreux cas où le fait de ne dépendre que des événements de souris et de pointeur casse les fonctionnalités pour les utilisateurs de clavier. Toujours tester avec le clavier mettra immédiatement en évidence les problèmes qui peuvent ensuite être résolus à l'aide des gestionnaires d'événements clavier.

Éléments d'interface plus complexes

Une expérience utilisateur plus complexe ne doit pas signifier une expérience moins accessible. Alors que l'accessibilité est plus facile à réaliser en codant au plus près du HTML, même les éléments d'interface les plus complexes peuvent être codés de manière accessible.

Nous avons besoin ici de connaître les [rôles ARIA](#), ainsi que les [états et propriétés ARIA](#) (liens en anglais). Ce sont des boîtes à outils pleines d'attributs HTML pris en charge par JSX et qui nous permettent de construire des composants React pleinement accessibles et hautement fonctionnels.

Chaque type d'élément d'interface a son modèle de conception spécifique et devrait fonctionner de la même manière avec les utilisateurs et les agents utilisateurs (notamment les navigateurs et les lecteurs d'écran) :

- [Pratiques de création WAI-ARIA - Modèles de conception et éléments d'interface](#) (en anglais).
- [Heydon Pickering - Exemples ARIA](#) (en anglais).
- [Composants inclusifs](#) (en anglais).

Autres points à considérer

Définir la langue

Indiquez la langue des pages de texte de façon à ce que les logiciels de lecture d'écran puissent sélectionner les paramètres de voix appropriés :

- [WebAIM - Langue du document](#) (en anglais).

Définir le titre du document

Définissez la balise `<title>` du document pour décrire correctement le contenu de la page courante, afin de garantir que l'utilisateur est au courant du contexte de la page en cours :

- [WCAG - Comprendre l'exigence du titre du document](#) (en anglais).

Nous pouvons le définir dans React en utilisant le [composant React Document Title](#).

Contraste des couleurs

Assurez-vous que tous les textes lisibles sur votre site web ont un contraste des couleurs suffisant pour rester lisibles de manière optimale par les utilisateurs malvoyants :

- [WCAG - Comprendre l'exigence du contraste des couleurs](#) (en anglais).
- [Tout savoir sur le contraste des couleurs et pourquoi vous devriez le repenser](#) (en anglais).
- [A11yProject - Qu'est-ce que le contraste des couleurs ?](#) (en anglais).

Il peut être fastidieux de calculer manuellement les combinaisons de couleurs appropriées pour toutes les situations sur votre site web, aussi vous est-il possible de [calculer une palette entière de couleurs accessible avec Colorable](#) (en anglais).

Les outils aXe et WAVE mentionnés ci-dessous incluent également des tests de contraste des couleurs et signalent les erreurs de contraste.

Si vous souhaitez étendre vos capacités de test de contraste, vous pouvez utiliser ces outils :

- [WebAIM - Vérification de contraste des couleurs](#) (en anglais).
- [The Paciello Group - Analyseur de contraste des couleurs](#) (en anglais).

Outils de développement et de test

Il existe de nombreux outils que nous pouvons utiliser pour nous assister durant la création d'applications web accessibles.

Le clavier

La vérification de loin la plus simple, mais aussi l'une des plus importantes, consiste à tester si l'ensemble de votre site web est accessible et utilisable avec le clavier seul. Procédez ainsi :

1. Débranchez votre souris.
2. Utilisez `Tab` et `Shift + Tab` pour naviguer.
3. Utilisez `Entrée` pour activer des éléments.
4. Le cas échéant, utilisez les touches de curseur du clavier pour interagir avec certains éléments, tels que les menus et les listes déroulantes.

Assistance au développement

Nous pouvons tester certaines fonctionnalités d'accessibilité directement dans notre code JSX. Souvent des contrôles automatiques sont déjà présents dans les EDI qui prennent en charge JSX pour vérifier les rôles, états et propriétés ARIA. Nous avons également accès à l'outil suivant :

eslint-plugin-jsx-a11y

L'extension [eslint-plugin-jsx-a11y](#) pour ESLint fournit des informations concernant les problèmes d'accessibilité dans votre JSX. De nombreux EDI vous permettent d'intégrer ces résultats directement dans leurs fenêtres d'analyse de code ou de code source.

[Create React App](#) pré-configurera cette extension avec un sous-ensemble de règles activées. Si vous souhaitez activer encore plus de règles d'accessibilité, vous pouvez créer un fichier `.eslintrc` à la racine de votre projet avec ce contenu :

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

Tester l'accessibilité dans le navigateur

Il existe un certain nombre d'outils pour exécuter des audits d'accessibilité sur des pages web dans votre navigateur. Veuillez les utiliser conjointement avec d'autres contrôles d'accessibilité mentionnés ici car ils ne peuvent tester que l'accessibilité technique de votre HTML.

aXe, aXe-core et react-axe

Deque Systems propose [aXe-core](#) pour automatiser les tests d'accessibilité de bout en bout de vos applications. Ce module comprend des intégrations pour Selenium.

[The Accessibility Engine](#) (en anglais) ou aXe, est une extension du navigateur qui fournit un inspecteur d'accessibilité en se basant sur `aXe-core`.

Vous pouvez aussi utiliser le module [react-axe](#) pour signaler directement ces résultats d'accessibilité dans la console durant le développement et le débogage.

WebAIM WAVE

L'[outil Web Accessibility Evaluation](#) (en anglais) est une autre extension du navigateur pour l'accessibilité.

Inspecteurs d'accessibilité et arbre d'accessibilité

[L'arbre d'accessibilité](#) (en anglais) est un sous-ensemble de l'arbre DOM qui contient des objets d'accessibilité pour chaque élément du DOM devant être exposé aux technologies d'assistance, telles que les lecteurs d'écrans.

Sur certains navigateurs, nous pouvons facilement consulter les informations d'accessibilité pour chaque élément de l'arbre d'accessibilité :

- Utiliser l'inspecteur d'accessibilité de Firefox.
- Utiliser l'inspecteur d'accessibilité de Chrome (en anglais).
- Utiliser l'inspecteur d'accessibilité d'OS X Safari (en anglais).

Lecteurs d'écran

Tester avec un lecteur d'écran devrait faire partie de vos tests d'accessibilité.

Veuillez noter que les combinaisons navigateur / lecteur d'écran ont leur importance. Il est recommandé de tester votre application dans le navigateur le plus adapté au lecteur d'écran de votre choix.

Lecteurs d'écran fréquemment utilisés

NVDA dans Firefox

NonVisual Desktop Access (en anglais) ou NVDA est un logiciel libre de lecteur d'écran qui est largement utilisé.

Reportez-vous aux guides suivants pour savoir comment utiliser au mieux NVDA :

- [WebAIM - Utiliser NVDA pour évaluer l'accessibilité web](#) (en anglais).
- [Deque - Raccourcis clavier NVDA](#) (en anglais).

VoiceOver dans Safari

VoiceOver est un lecteur d'écran intégré dans les appareils d'Apple.

Reportez-vous aux guides suivants pour savoir comment activer et utiliser VoiceOver :

- [WebAIM - Utiliser VoiceOver pour évaluer l'accessibilité web](#) (en anglais).
- [Deque - Raccourcis clavier pour VoiceOver sur OS X](#) (en anglais).
- [Deque - Raccourcis pour VoiceOver sur iOS](#) (en anglais).

JAWS dans Internet Explorer

Job Access With Speech (en anglais) ou JAWS, est un lecteur d'écran très largement utilisé sur Windows.

Reportez-vous aux guides suivants pour utiliser au mieux JAWS :

- [WebAIM - Utiliser JAWS pour évaluer l'accessibilité web](#) (en anglais).
- [Deque - Raccourcis clavier pour JAWS](#) (en anglais).

Autres lecteurs d'écran

ChromeVox dans Google Chrome

ChromeVox est un lecteur d'écran intégré aux Chromebooks et est disponible [en tant qu'extension](#) pour Google Chrome.

Reportez-vous aux guides suivants pour utiliser au mieux ChromeVox :

- [Aide de Google Chromebook - Utiliser le lecteur d'écran intégré](#).

- Référence des raccourcis clavier pour ChromeVox Classic (en anglais).

Ajouter React à un site web

Utiliser React en fonction de vos besoins

React a été conçu dès le départ pour une adoption progressive, et **vous pouvez utiliser React à minima ou autant que nécessaire**. Peut-être souhaitez-vous juste ajouter un peu d'interactivité à une page existante. Les composants React sont une bonne solution pour ça.

La majorité des sites web ne sont pas, et n'ont pas besoin d'être, des applications web monopages (*Single-Page Application ou plus simplement SPA, NdT*). **Avec juste quelques lignes de code et sans outillage particulier**, vous pouvez essayer React sur une petite partie de votre site web. Vous pourrez par la suite ajouter des composants supplémentaires ou limiter React à quelques éléments dynamiques.

- [Ajouter React en une minute](#)
- Optionnel : [Essayer React avec JSX](#) (aucun *bundler* nécessaire !)

Ajouter React en une minute

Dans cette partie, nous allons voir comment ajouter un composant React à une page HTML existante. Vous pouvez suivre les étapes avec votre propre site web, ou avec une page HTML vide pour vous entraîner.

Pas besoin d'outils compliqués ni de faire des installations : **pour suivre cette section, vous avez juste besoin d'une connexion à internet et d'une minute de votre temps**.

Optionnel : [Télécharger l'exemple complet \(2 Ko zippé\)](#)

Étape 1 : ajouter un conteneur DOM à votre HTML

Pour commencer, ouvrez la page HTML que vous souhaitez modifier. Ajoutez une balise `<div>` vide pour indiquer l'endroit dans lequel vous souhaitez afficher quelque chose avec React. Par exemple :

```
<!-- ... code HTML existant ... -->
<div id="like_button_container"></div>
<!-- ... code HTML existant ... -->
```

L'attribut `id` de votre `<div>`, qui est unique, va vous permettre de manipuler cette balise depuis le code JavaScript afin d'afficher un composant React dans celle-ci.

Astuce

Vous pouvez placer une telle `<div>` « conteneur » **où vous le souhaitez** dans la balise `<body>`. Vous pouvez d'ailleurs avoir autant de conteneurs DOM indépendants que vous le souhaitez dans la même page. Il sont généralement vides, car React remplacera leur contenu existant de toutes façons.

Étape 2 : ajouter les balises de script

Ensuite, ajoutez trois balises `<script>` à votre page HTML, juste avant la fermeture de la balise `</body>` :

```
<!-- ... autres contenus HTML ... -->
<!-- Charge React -->
<!-- Remarque : pour le déploiement, remplacez "development.js"
     par "production.min.js" -->
<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>
<!-- Charge notre composant React -->
<script src="like_button.js"></script>
</body>
```

Les deux premières balises permettent de charger React, alors que la troisième charge le code de votre composant.

Étape 3 : créer un composant React

Créez un fichier nommé `like_button.js` dans le même dossier que votre page HTML.

Ouvrez et copiez [le code de démarrage](#) et collez son contenu dans votre fichier.

Astuce

Ce code définit un composant appelé `LikeButton`. Ne vous en faites pas si vous ne comprenez pas son code tout de suite, nous aborderons les facettes de React ultérieurement dans notre [tutoriel pratique](#) et dans le [guide des fondamentaux](#). Pour le moment, essayons simplement de l'afficher à l'écran !

Ajoutez deux lignes à la fin de `like_button.js`, après [le code de démarrage](#) :

```
// ... le code de démarrage que vous avez collé ...
const domContainer = document.querySelector('#like_button_container');
ReactDOM.render(e(LikeButton), domContainer);
```

Ces deux lignes de code vont remplacer le contenu de la `<div>` que nous avons ajoutée dans la première étape. Ainsi elles afficheront notre composant React sous forme de bouton « J'aime ».

Et voilà !

Il n'y a pas de quatrième étape. **Vous venez tout juste d'ajouter votre premier composant React à votre site web.**

Pour plus d'astuces concernant l'intégration de React, n'hésitez pas à consulter les prochaines sections.

[Voir le code source intégral de l'exemple](#)[Télécharger le code source intégral de l'exemple \(2 Ko zippé\)](#)

Astuce : réutiliser un composant

Généralement, on souhaite afficher des composants React à plusieurs endroits d'une page HTML. Voici un exemple qui affiche le bouton « J'aime » à trois reprises et lui fournit quelques données :

[Voir le code source intégral de l'exemple](#)[Télécharger le code source intégral de l'exemple \(2 Ko zippé\)](#)

Remarque

Cette solution est souvent utilisée lorsque les éléments de React doivent être isolés les uns des autres. Il est cependant plus facile d'utiliser le principe de [composition de composants](#) au sein de votre code React.

Astuce : minifier votre JavaScript pour la production

Avant de déployer votre site web en production, gardez à l'esprit qu'un JavaScript non-minifié risque de ralentir significativement le chargement de la page pour vos utilisateurs.

Si vous minifiez déjà vos scripts applicatifs et si vous vous assurez de référencer les versions de React dont les noms se terminent en `production.min.js`, **votre site sera prêt pour la production** :

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

Si vous n'avez pas de minification déjà en place pour vos scripts, voici [une façon de faire](#).

Optionnel : essayer React avec JSX

Dans les exemples ci-dessus, nous utilisions uniquement les fonctionnalités prises en charge nativement par les navigateurs. C'est pourquoi nous appelions une fonction JavaScript pour indiquer à React ce qu'il fallait afficher :

```
const e = React.createElement;

// Affiche un bouton « J'aime »
return e(
  'button',
  { onClick: () => this.setState({ liked: true }) },
  'J'aime'
);
```

Néanmoins, React nous offre aussi la possibilité d'utiliser plutôt **JSX**.

```
// Affiche un bouton « J'aime »
return (
  <button onClick={() => this.setState({ liked: true })}>
    J'aime
  </button>
);
```

Ces deux extraits de code sont équivalents. Même si la syntaxe **JSX est complètement optionnelle**, la plupart des développeur·euse·s React la trouvent très pratique pour écrire du code (avec React ou même avec d'autres bibliothèques).

Vous pouvez tester JSX en utilisant un [convertisseur en ligne](#).

Essayer JSX en un clin d'œil

La façon la plus simple et rapide de tester JSX dans votre projet est d'ajouter la balise `<script>` ci-dessous à votre page :

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

Vous pouvez désormais utiliser JSX dans les fichiers chargés par n'importe quelle balise `<script>` simplement en lui ajoutant l'attribut `type="text/babel"`. Vous pouvez télécharger et tester cet exemple contenant un [fichier HTML utilisant JSX](#).

Cette approche est acceptable pour se former ou réaliser des démos simples. Cependant, elle va ralentir l'affichage de votre site, elle n'est donc **pas adaptée pour la production**. Lorsque vous serez prêt·e à aller plus loin, supprimez la balise `<script>` et l'attribut `type="text/babel"` que vous venez d'ajouter. Dans la section suivante, nous verrons plutôt comment configurer le préprocesseur JSX afin de convertir automatiquement toutes les balises `<script>`.

Ajouter JSX à votre projet

L'ajout de JSX à votre projet ne nécessite pas d'outils compliqués comme un *bundler* ou un serveur de développement. Cela ressemble plus à **l'ajout d'un préprocesseur CSS**. Le seul outil indispensable est [Node.js](#), qui doit être installé sur votre ordinateur.

Dans un terminal (invite de commande), déplacez-vous dans le dossier de votre projet, et copiez-collez ces deux commandes :

1. **Étape 1 :** exécutez `npm init -y` (si ça ne fonctionne pas, [voici un correctif](#))
2. **Étape 2 :** exécutez `npm install babel-cli@6 babel-preset-react-app@3`

Astuce

Nous **utilisons npm uniquement pour installer le préprocesseur JSX** (vous n'en aurez besoin pour rien d'autre). React et le code applicatif peuvent rester sous la forme de balises `<script>` sans modification.

Bravo ! Vous venez d'ajouter **une configuration JSX de production** à votre projet.

Lancer le préprocesseur JSX

Créez un dossier nommé `src` et lancez la commande suivante dans le terminal :

```
npx babel --watch src --out-dir . --presets react-app/prod
```

Remarque

`npx` n'est pas une faute de frappe, il s'agit d'un [outil d'exécution de paquet fourni avec npm 5.2+](#) ("package runner tool", NdT).

Si le message d'erreur "*You have mistakenly installed the `babel` package*" apparaît, vous avez peut-être loupé [l'étape précédente](#). Dans le même dossier, recommencez l'opération et essayez de nouveau.

Cette commande surveille votre JSX en continu, inutile donc d'attendre qu'elle se termine.

Dorénavant, si à partir du [code de démarrage JSX](#), vous créez un fichier nommé `src/like_button.js`, le préprocesseur va produire un fichier traduit `like_button.js`, avec du code JavaScript exploitable par le navigateur. Ce fichier sera mis à jour à chaque modification de votre fichier JSX original.

Et en bonus, ça vous permet aussi d'utiliser des syntaxes JavaScript modernes comme les classes, sans vous soucier de la compatibilité avec d'anciens navigateurs. L'outil que nous venons d'utiliser se nomme Babel, et vous pouvez en savoir plus sur lui en allant sur [sa documentation](#).

Si vous vous sentez à l'aise avec les outils de compilation et souhaitez en apprendre plus, la [partie suivante](#) décrit les outils les plus populaires et les plus accessibles de la chaîne de compilation. Si ce n'est pas le cas, les balises décrites précédemment fonctionneront à merveille !

[Go to TOC](#)

Animation Add-Ons

Note:

`ReactTransitionGroup` and `ReactCSSTransitionGroup` have been moved to the `react-transition-group` package that is maintained by the community. Its 1.x branch is completely API-compatible with the existing addons. Please file bugs and feature requests in the [new repository](#).

The `ReactTransitionGroup` add-on component is a low-level API for animation, and `ReactCSSTransitionGroup` is an add-on component for easily implementing basic CSS animations and transitions.

High-level API: ReactCSSTransitionGroup

`ReactCSSTransitionGroup` is a high-level API based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It's inspired by the excellent [ng-animate](#) library.

Importing

```
import ReactCSSTransitionGroup from 'react-transition-group'; // ES6
var ReactCSSTransitionGroup = require('react-transition-group'); // ES5 with npm
```

```
class TodoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: ['hello', 'world', 'click', 'me']};
    this.handleAdd = this.handleAdd.bind(this);
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter some text')
    ]);
    this.setState({items: newItems});
  }

  handleRemove(i) {
    let newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  }

  render() {
    const items = this.state.items.map((item, i) => (
      <div key={item} onClick={() => this.handleRemove(i)}>
        {item}
      </div>
    ));
    return (
      <div>
```

```

    <button onClick={this.handleAdd}>Add Item</button>
    <ReactCSSTransitionGroup
      transitionName="example"
      transitionEnterTimeout={500}
      transitionLeaveTimeout={300}>
      {items}
    </ReactCSSTransitionGroup>
  </div>
);
}
}

```

Note:

You must provide the `key` attribute for all children of `ReactCSSTransitionGroup`, even when only rendering a single item. This is how React will determine which children have entered, left, or stayed.

In this component, when a new item is added to `ReactCSSTransitionGroup` it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

You can use these classes to trigger a CSS animation or transition. For example, try adding this CSS and adding a new list item:

```

.example-enter {
  opacity: 0.01;
}

.example-enter.example-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}

.example-leave {
  opacity: 1;
}

.example-leave.example-leave-active {
  opacity: 0.01;
  transition: opacity 300ms ease-in;
}

```

You'll notice that animation durations need to be specified in both the CSS and the render method; this tells React when to remove the animation classes from the element and -- if it's leaving -- when to remove the element from the DOM.

Animate Initial Mounting

`ReactCSSTransitionGroup` provides the optional prop `transitionAppear`, to add an extra transition phase at the initial mount of the component. There is generally no transition phase at the initial mount as the default value of `transitionAppear` is `false`. The following is an example which passes the prop `transitionAppear` with the value `true`.

```

render() {
  return (
    <ReactCSSTransitionGroup
      transitionName="example"
      transitionAppear={true}
      transitionAppearTimeout={500}
      transitionEnter={false}
      transitionLeave={false}>
      <h1>Fading at Initial Mount</h1>
    </ReactCSSTransitionGroup>
  );
}

```

During the initial mount `ReactCSSTransitionGroup` will get the `example-appear` CSS class and the `example-appear-active` CSS class added in the next tick.

```

.example-appear {
  opacity: 0.01;
}

.example-appear.example-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}

```

At the initial mount, all children of the `ReactCSSTransitionGroup` will `appear` but not `enter`. However, all children later added to an existing `ReactCSSTransitionGroup` will `enter` but not `appear`.

Note:

The prop `transitionAppear` was added to `ReactCSSTransitionGroup` in version `0.13`. To maintain backwards compatibility, the default value is set to `false`.

However, the default values of `transitionEnter` and `transitionLeave` are `true` so you must specify `transitionEnterTimeout` and `transitionLeaveTimeout` by default. If you don't need either enter or leave animations, pass `transitionEnter={false}` or `transitionLeave={false}`.

Custom Classes

It is also possible to use custom class names for each of the steps in your transitions. Instead of passing a string into `transitionName` you can pass an object containing either the `enter` and `leave` class names, or an object containing the `enter`, `enter-active`, `leave-active`, and `leave` class names. If only the enter and leave classes are provided, the enter-active and leave-active classes will be determined by appending '-active' to the end of the class name. Here are two examples using custom classes:

```

// ...
<ReactCSSTransitionGroup
  transitionName={{ 
    enter: 'enter',
    enterActive: 'enterActive',
    leave: 'leave',
    leaveActive: 'leaveActive',
  }}

```

```

        appear: 'appear',
        appearActive: 'appearActive'
    } }>
{item}
</ReactCSSTransitionGroup>

<ReactCSSTransitionGroup
  transitionName={ {
    enter: 'enter',
    leave: 'leave',
    appear: 'appear'
  } }>
{item2}
</ReactCSSTransitionGroup>
// ...

```

Animation Group Must Be Mounted To Work

In order for it to apply transitions to its children, the `ReactCSSTransitionGroup` must already be mounted in the DOM or the prop `transitionAppear` must be set to `true`.

The example below would **not** work, because the `ReactCSSTransitionGroup` is being mounted along with the new item, instead of the new item being mounted within it. Compare this to the [Getting Started](#) section above to see the difference.

```

render() {
  const items = this.state.items.map((item, i) =>
    <div key={item} onClick={() => this.handleRemove(i)}>
      <ReactCSSTransitionGroup transitionName="example">
        {item}
      </ReactCSSTransitionGroup>
    </div>
  );
}

return (
  <div>
    <button onClick={this.handleAdd}>Add Item</button>
    {items}
  </div>
);
}

```

Animating One or Zero Items

In the example above, we rendered a list of items into `ReactCSSTransitionGroup`. However, the children of `ReactCSSTransitionGroup` can also be one or zero items. This makes it possible to animate a single element entering or leaving. Similarly, you can animate a new element replacing the current element. For example, we can implement a simple image carousel like this:

```

import ReactCSSTransitionGroup from 'react-transition-group';

function ImageCarousel(props) {
  return (
    <div>
      <ReactCSSTransitionGroup
        transitionName="carousel"
        transitionEnterTimeout={300}

```

```

        transitionLeaveTimeout={300}>
      <img src={props.imageSrc} key={props.imageSrc} />
    </ReactCSSTransitionGroup>
  </div>
);
}

```

Disabling Animations

You can disable animating `enter` or `leave` animations if you want. For example, sometimes you may want an `enter` animation and no `leave` animation, but `ReactCSSTransitionGroup` waits for an animation to complete before removing your DOM node. You can add `transitionEnter={false}` or `transitionLeave={false}` props to `ReactCSSTransitionGroup` to disable these animations.

Note:

When using `ReactCSSTransitionGroup`, there's no way for your components to be notified when a transition has ended or to perform any more complex logic around animation. If you want more fine-grained control, you can use the lower-level `ReactTransitionGroup` API which provides the hooks you need to do custom transitions.

Low-level API: `ReactTransitionGroup`

Importing

```

import ReactTransitionGroup from 'react-addons-transition-group' // ES6
var ReactTransitionGroup = require('react-addons-transition-group') // ES5 with
npm

```

`ReactTransitionGroup` is the basis for animations. When children are declaratively added or removed from it (as in the [example above](#)), special lifecycle methods are called on them.

- `componentWillAppear()`
- `componentDidAppear()`
- `componentWillEnter()`
- `componentDidEnter()`
- `componentWillLeave()`
- `componentDidLeave()`

Rendering a Different Component

`ReactTransitionGroup` renders as a `span` by default. You can change this behavior by providing a `component` prop. For example, here's how you would render a ``:

```

<ReactTransitionGroup component="ul">
  {/* ... */}
</ReactTransitionGroup>

```

Any additional, user-defined, properties will become properties of the rendered component. For example, here's how you would render a `` with CSS class:

```
<ReactTransitionGroup component="ul" className="animated-list">
  {/* ... */}
</ReactTransitionGroup>
```

Every DOM component that React can render is available for use. However, `component` does not need to be a DOM component. It can be any React component you want; even ones you've written yourself! Just write `component={List}` and your component will receive `this.props.children`.

Rendering a Single Child

People often use `ReactTransitionGroup` to animate mounting and unmounting of a single child such as a collapsible panel. Normally `ReactTransitionGroup` wraps all its children in a `span` (or a custom `component` as described above). This is because any React component has to return a single root element, and `ReactTransitionGroup` is no exception to this rule.

However if you only need to render a single child inside `ReactTransitionGroup`, you can completely avoid wrapping it in a `` or any other DOM component. To do this, create a custom component that renders the first child passed to it directly:

```
function FirstChild(props) {
  const childrenArray = React.Children.toArray(props.children);
  return childrenArray[0] || null;
}
```

Now you can specify `FirstChild` as the `component` prop in `<ReactTransitionGroup>` props and avoid any wrappers in the result DOM:

```
<ReactTransitionGroup component={FirstChild}>
  {someCondition ? <MyComponent /> : null}
</ReactTransitionGroup>
```

This only works when you are animating a single child in and out, such as a collapsible panel. This approach wouldn't work when animating multiple children or replacing the single child with another child, such as an image carousel. For an image carousel, while the current image is animating out, another image will animate in, so `<ReactTransitionGroup>` needs to give them a common DOM parent. You can't avoid the wrapper for multiple children, but you can customize the wrapper with the `component` prop as described above.

Reference

`componentWillAppear()` **#componentwillappear**

```
componentWillAppear(callback)
```

This is called at the same time as `componentDidMount()` for components that are initially mounted in a `TransitionGroup`. It will block other animations from occurring until `callback` is called. It is only called on the initial render of a `TransitionGroup`.

`componentDidAppear()` **{#componentdidappear}**

`componentDidAppear()`

This is called after the `callback` function that was passed to `componentWillAppear()` is called.

`componentWillEnter()` **{#componentwillenter}**

`componentWillEnter(callback)`

This is called at the same time as `componentDidMount()` for components added to an existing `TransitionGroup`. It will block other animations from occurring until `callback` is called. It will not be called on the initial render of a `TransitionGroup`.

`componentDidEnter()` **{#componentdidenter}**

`componentDidEnter()`

This is called after the `callback` function that was passed to `componentWillEnter()` is called.

`componentWillLeave()` **{#componentwillleave}**

`componentWillLeave(callback)`

This is called when the child has been removed from the `ReactTransitionGroup`. Though the child has been removed, `ReactTransitionGroup` will keep it in the DOM until `callback` is called.

`componentDidLeave()` **{#componentdidleave}**

`componentDidLeave()`

This is called when the `willLeave` `callback` is called (at the same time as `componentWillUnmount()`).

[Go to TOC](#)

Keyed Fragments

Note:

`React.addons` entry point is deprecated as of React v15.5. We now have first class support for fragments which you can read about [here](#).

Importing

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

Overview

In most cases, you can use the `key` prop to specify keys on the elements you're returning from `render`. However, this breaks down in one situation: if you have two sets of children that you need to reorder, there's no way to put a key on each set without adding a wrapper element.

That is, if you have a component such as:

```
function Swapper(props) {
  let children;
  if (props.swapped) {
    children = [props.rightChildren, props.leftChildren];
  } else {
    children = [props.leftChildren, props.rightChildren];
  }
  return <div>{children}</div>;
}
```

The children will unmount and remount as you change the `swapped` prop because there aren't any keys marked on the two sets of children.

To solve this problem, you can use the `createFragment` add-on to give keys to the sets of children.

```
Array<ReactNode> createFragment(object children) {#arrayreactnode-createfragmentobject-children}
```

Instead of creating arrays, we write:

```
import createFragment from 'react-addons-create-fragment';

function Swapper(props) {
  let children;
  if (props.swapped) {
    children = createFragment({
      right: props.rightChildren,
      left: props.leftChildren
    });
  }
  return <div>{children}</div>;
}
```

```
    } else {
      children = createFragment({
        left: props.leftChildren,
        right: props.rightChildren
      });
    }
    return <div>{children}</div>;
}
```

The keys of the passed object (that is, `left` and `right`) are used as keys for the entire set of children, and the order of the object's keys is used to determine the order of the rendered children. With this change, the two sets of children will be properly reordered in the DOM without unmounting.

The return value of `createFragment` should be treated as an opaque object; you can use the `React.Children` helpers to loop through a fragment but should not access it directly. Note also that we're relying on the JavaScript engine preserving object enumeration order here, which is not guaranteed by the spec but is implemented by all major browsers and VMs for objects with non-numeric keys.

Performance Tools

Note:

As of React 16, `react-addons-perf` is not supported. Please use [your browser's profiling tools](#) to get insight into which components re-render.

Importing

```
import Perf from 'react-addons-perf'; // ES6
var Perf = require('react-addons-perf'); // ES5 with npm
```

Overview

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a `shouldComponentUpdate()` method where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, `Perf` is a profiling tool that tells you exactly where you need to put these methods.

See these articles for an introduction to React performance tooling:

- ["How to Benchmark React Components"](#)
- ["Performance Engineering with React"](#)
- ["A Deep Dive into React Perf Debugging"](#)

Development vs. Production Builds

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the [minified production build](#). The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

However, the perf tools described on this page only work when using the development build of React. Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

Using Perf

The `Perf` object can be used with React in development mode only. You should not include this bundle when building your app for production.

Getting Measurements

- `start()`
- `stop()`

- `getLastMeasurements()`

Printing Results

The following methods use the measurements returned by `Perf.getLastMeasurements()` to pretty-print the result.

- `printInclusive()`
 - `printExclusive()`
 - `printWasted()`
 - `printOperations()`
 - `printDOM()`
-

Reference

`start() {#start}`

`stop() {#stop}`

```
Perf.start()  
// ...  
Perf.stop()
```

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` to get the measurements.

`getLastMeasurements() {#getlastmeasurements}`

```
Perf.getLastMeasurements()
```

Get the opaque data structure describing measurements from the last start-stop session. You can save it and pass it to the other print methods in `Perf` to analyze past measurements.

Note

Don't rely on the exact format of the return value because it may change in minor releases. We will update the documentation if the return value format becomes a supported part of the public API.

`printInclusive() {#printinclusive}`

```
Perf.printInclusive(measurements)
```

Prints the overall time taken. When no arguments are passed, `printInclusive` defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:

(index)	Owner > component	Inclusive time (ms)	Instances
0	"<root> > App"	15.31	37
1	"App > Box"	6.47	37
2	"Box > Box2"	1.96	37

printExclusive() `{#printexclusive}`

`Perf.printExclusive(measurements)`

"Exclusive" times don't include the times taken to mount the components: processing props, calling `componentWillMount` and `componentDidMount`, etc.

(index)	Component c...	Total inclus...	Exclusive m...	Exclusive r...	Mount time ...	Render time...	Instances
0	"App"	15.31	1.76	1.23	0.04	0.03	37

printWasted() `{#printwasted}`

`Perf.printWasted(measurements)`

The most useful part of the profiler.

"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.

(index)	Owner > component	Wasted time (ms)	Instances
0	"<root> > App"	15.317999947001226	37
1	"App > Box"	6.479999952716753	37
2	"Box > Box2"	1.9630000460892916	37

printOperations() `{#printoperations}`

`Perf.printOperations(measurements)`

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".

(index)	data-reactid	type	args
0	"0"	"set innerHTML"	"<div data-reactid=".0\..."

printDOM() `{#printdom}`

`Perf.printDOM(measurements)`

This method has been renamed to `printOperations()`. Currently `printDOM()` still exists as an alias but it prints a deprecation warning and will eventually be removed.

PureRenderMixin

Note:

`PureRenderMixin` is a legacy add-on. Use `React.PureComponent` instead.

Importing

```
import PureRenderMixin from 'react-addons-pure-render-mixin'; // ES6
var PureRenderMixin = require('react-addons-pure-render-mixin'); // ES5 with npm
```

Overview

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
const createReactClass = require('create-react-class');

createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

[Go to TOC](#)

Shallow Compare

Note:

`shallowCompare` is a legacy add-on. Use `React.memo` or `React.PureComponent` instead.

Importing

```
import shallowCompare from 'react-addons-shallow-compare'; // ES6
var shallowCompare = require('react-addons-shallow-compare'); // ES5 with npm
```

Overview

Before `React.PureComponent` was introduced, `shallowCompare` was commonly used to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

```
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.

It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update.

`shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

[Go to TOC](#)

Moteur de rendu superficiel

Importation

```
import ShallowRenderer from 'react-test-renderer/shallow'; // ES6
var ShallowRenderer = require('react-test-renderer/shallow'); // ES5 avec npm
```

Aperçu

Quand on écrit des tests unitaires pour React, le rendu superficiel peut être utile. Le rendu superficiel vous permet de réaliser le rendu d'un composant « à un seul niveau de profondeur » afin de pouvoir vérifier ce que renvoie sa fonction de rendu, sans vous préoccuper des composants enfants, qui ne sont pas sollicités. Ça ne nécessite donc pas de DOM.

Par exemple, si vous avez le composant suivant :

```
function MyComponent() {
  return (
    <div>
      <span className="heading">Titre</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

Vous pouvez alors faire les vérifications suivantes :

```
import ShallowRenderer from 'react-test-renderer/shallow';

// dans votre test :
const renderer = new ShallowRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Titre</span>,
  <Subcomponent foo="bar" />
]);
```

Le rendu superficiel a pour le moment quelques limites, notamment l'absence de prise en charge des refs.

Note:

Nous vous conseillons par ailleurs de regarder [l'API de rendu superficiel](#) (en anglais) d'Enzyme. Elle propose une meilleure API de haut niveau pour ce type de fonctionnalité.

Référence de l'API

`shallowRenderer.render() {#shallowrendererrender}`

Vous pouvez voir le `shallowRenderer` comme un « endroit » dans lequel faire le rendu du composant que vous testez, et depuis lequel vous pouvez extraire la sortie que ce composant produit.

`shallowRenderer.render()` est similaire à `ReactDOM.render()`, à ceci près qu'elle n'a pas besoin du DOM et n'effectue le rendu qu'à un seul niveau de profondeur. Ça signifie que vous pouvez tester des composants indépendamment de l'implémentation de leurs enfants.

`shallowRenderer.getRenderOutput() {#shallowrenderergetrenderoutput}`

Après avoir appelé `shallowRenderer.render()`, vous pouvez utiliser `shallowRenderer.getRenderOutput()` pour récupérer le rendu superficiel obtenu.

Vous pouvez alors vérifier ce que vous souhaitez sur le résultat attendu.

Utilitaires de test

Importation

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6  
var ReactTestUtils = require('react-dom/test-utils'); // ES5 avec npm
```

Aperçu

`ReactTestUtils` facilite le test de composants React quel que soit votre framework de test. Chez Facebook, nous utilisons [Jest](#) pour réaliser facilement nos tests JavaScript. Pour bien démarrer avec Jest, vous pouvez lire le [Tutoriel React](#) du site web de Jest.

Remarque

Nous vous conseillons d'utiliser [React Testing Library](#). Cette bibliothèque est conçue pour encourager l'écriture de tests utilisant vos composants de façon similaire aux utilisateurs finaux.

Par ailleurs, Airbnb propose un utilitaire de test appelé [Enzyme](#), qui permet de facilement exprimer des assertions, manipuler vos composants React et en explorer le résultat.

- `act()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`
- `renderIntoDocument()`
- `Simulate`

Référence de l'API

act() {#act}

Afin de préparer un composant aux assertions, il suffit d'enrober le code qui se charge de son rendu et de ses mises à jour au sein d'une fonction de rappel qui sera passée à `act()`. De cette manière, votre test s'exécutera dans un environnement proche de celui de React dans le navigateur.

Remarque

Si vous utilisez `react-test-renderer`, ce dernier propose un export de `act` qui se comporte de la même façon.

Prenons l'exemple d'un composant `Counter` :

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
    this.handleClick = this.handleClick.bind(this);
  }
  componentDidMount() {
    document.title = `Vous avez cliqué ${this.state.count} fois`;
  }
  componentDidUpdate() {
    document.title = `Vous avez cliqué ${this.state.count} fois`;
  }
  handleClick() {
    this.setState(state => ({
      count: state.count + 1,
    }));
  }
  render() {
    return (
      <div>
        <p>Vous avez cliqué {this.state.count} fois</p>
        <button onClick={this.handleClick}>
          Cliquez ici
        </button>
      </div>
    );
  }
}
```

Voici comment nous pouvons le tester :

```
import React from 'react';
import ReactDOM from 'react-dom';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
```

```

    container = document.createElement('div');
    document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('peut afficher et mettre à jour un compteur', () => {
  // Teste le premier affichage et l'appel à componentDidMount
  act(() => {
    ReactDOM.render(<Counter />, container);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('Vous avez cliqué 0 fois');
  expect(document.title).toBe('Vous avez cliqué 0 fois');

  // Teste un second affichage et l'appel à componentDidUpdate
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('Vous avez cliqué 1 fois');
  expect(document.title).toBe('Vous avez cliqué 1 fois');
});

```

N'oubliez pas que l'émission d'événements DOM ne fonctionne que lorsque le conteneur DOM a été ajouté au `document`. Vous pouvez utiliser un utilitaire comme `React Testing Library` afin de réduire le volume de code générique.

La page des `recettes` contient de plus amples détails sur le fonctionnement de `act()`, avec des exemples d'utilisation.

`mockComponent()` [\[#mockcomponent\]](#)

```

mockComponent(
  componentClass,
  [mockTagName]
)

```

Cette méthode prend un module de composant simulé et lui ajoute des méthodes utiles pour lui permettre d'être utilisé comme un composant React factice. Plutôt que de réaliser un rendu classique, le composant va simplement devenir une balise `<div>` (ou toute autre balise si `mockTagName` est renseigné) contenant les enfants qui lui sont fournis.

Remarque

`mockComponent()` est une API obsolète. Nous recommandons plutôt de recourir à `jest.mock()`.

isElement() {#iselement}

```
isElement(element)
```

Renvoie `true` si `element` est un élément React.

isElementOfType() {#iselementoftype}

```
isElementOfType(
  element,
  componentClass
)
```

Renvoie `true` si `element` est un élément React dont le type est un `componentClass` React.

isDOMComponent() {#isdomcomponent}

```
isDOMComponent(instance)
```

Renvoie `true` si `instance` est un composant DOM (tel que `<div>` ou ``).

isCompositeComponent() {#iscompositecomponent}

```
isCompositeComponent(instance)
```

Renvoie `true` si `instance` est un composant défini par l'utilisateur, via une classe ou une fonction.

isCompositeComponentWithType() {#iscompositecomponentwithtype}

```
isCompositeComponentWithType(
  instance,
  componentClass
)
```

Renvoie `true` si `instance` est un composant dont le type est un `componentClass` React.

findAllInRenderedTree() {#findallinrenderedtree}

```
findAllInRenderedTree(
  tree,
  test
)
```

Parcourt tous les composants présents dans `tree` en ne retenant que les composants pour lesquels `test(component)` renvoie `true`. Ce n'est guère utile en soi, mais ça sert de base à d'autres utilitaires de test.

scryRenderedDOMComponentsWithClass() {#scryrendereddomcomponentswithclass}

```
scryRenderedDOMComponentsWithClass(  
  tree,  
  className  
)
```

Trouve tous les éléments DOM des composants de l'arbre de rendu qui sont des composants DOM dont la classe CSS correspond à `className`.

findRenderedDOMComponentWithClass() {#findrendereddomcomponentwithclass}

```
findRenderedDOMComponentWithClass(  
  tree,  
  className  
)
```

Fonctionne comme `scryRenderedDOMComponentsWithClass()`, mais cette méthode s'attend à ne trouver qu'un seul résultat, qu'elle renverra. Si elle trouve un nombre d'éléments différent de un, elle lèvera une exception.

scryRenderedDOMComponentsWithTag() {#scryrendereddomcomponentswithtag}

```
scryRenderedDOMComponentsWithTag(  
  tree,  
  tagName  
)
```

Trouve tous les éléments DOM parmi les composants de l'arbre de rendu qui sont des composants DOM dont le nom de balise correspond à `tagName`.

findRenderedDOMComponentWithTag() {#findrendereddomcomponentwithtag}

```
findRenderedDOMComponentWithTag(  
  tree,  
  tagName  
)
```

Fonctionne comme `scryRenderedDOMComponentsWithTag()`, mais cette méthode s'attend à ne trouver qu'un seul résultat, qu'elle renverra. Si elle trouve un nombre d'éléments différent de un, elle lèvera une exception.

scryRenderedComponentsWithType() {#scryrenderedcomponentswithtype}

```
scryRenderedComponentsWithType(
  tree,
  componentClass
)
```

Trouve toutes les instances des composants dont le type est `componentClass`.

findRenderedComponentWithType() {#findrenderedcomponentwithtype}

```
findRenderedComponentWithType(
  tree,
  componentClass
)
```

Fonctionne comme `scryRenderedComponentsWithType()`, mais cette méthode s'attend à ne trouver qu'un seul résultat, qu'elle renverra. Si elle trouve un nombre d'éléments différent de un, elle lèvera une exception.

renderIntoDocument() {#renderintodocument}

```
renderIntoDocument(element)
```

Réalise le rendu d'un élément React au sein d'un nœud du DOM détaché du document. **Cette fonction nécessite un DOM**. C'est en fait équivalent à :

```
const domContainer = document.createElement('div');
ReactDOM.render(element, domContainer);
```

Remarque

Vous aurez besoin d'avoir `window`, `window.document` et `window.document.createElement` disponibles globalement **avant** votre import de `React`. Dans le cas contraire, React pensera qu'il ne peut accéder au DOM, et des méthodes telles que `setState` ne fonctionneront pas.

Autre utilitaires

Simulate {#simulate}

```
Simulate.{eventName}(
  element,
  [eventData]
)
```

Simule l'envoi d'un événement sur un nœud du DOM avec des données optionnelles dans `eventData`.

`Simulate` dispose d'une méthode pour chaque événement que React comprend.

Cliquer sur un élément

```
// <button ref={(node) => this.button = node}>...</button>
const node = this.button;
ReactTestUtils.Simulate.click(node);
```

Changer la valeur d'un champ de saisie puis presser ENTRÉE.

```
// <input ref={(node) => this.textInput = node} />
const node = this.textInput;
node.value = 'girafe';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

Remarque

Vous devrez fournir toute les propriétés dédiées aux événements que vous utilisez dans votre composant (par exemple keyCode, which, etc.) car React ne les créera pas pour vous.

[Go to TOC](#)

Two-way Binding Helpers

Note:

`LinkedStateMixin` is deprecated as of React v15. The recommendation is to explicitly set the value and change handler, instead of using `LinkedStateMixin`.

Importing

```
import LinkedStateMixin from 'react-addons-linked-state-mixin'; // ES6
var LinkedStateMixin = require('react-addons-linked-state-mixin'); // ES5 with npm
```

Overview

`LinkedStateMixin` is an easy way to express two-way binding with React.

In React, data flows one way: from owner to child. We think that this makes your app's code easier to understand. You can think of it as "one-way data binding."

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you'll often want to update some React `state` when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a "change" event, read from your data source (usually the DOM) and call `setState()` on one of your components. "Closing the data flow loop" explicitly leads to more understandable and easier-to-maintain programs. See [our forms documentation](#) for more information.

Two-way binding -- implicitly enforcing that some value in the DOM is always consistent with some React `state` -- is concise and supports a wide variety of applications. We've provided `LinkedStateMixin`: syntactic sugar for setting up the common data flow loop pattern described above, or "linking" some data source to React `state`.

Note:

`LinkedStateMixin` is just a thin wrapper and convention around the `onChange / setState()` pattern. It doesn't fundamentally change how data flows in your React application.

LinkedStateMixin: Before and After

Here's a simple form example without using `LinkedStateMixin`:

```
var createReactClass = require('create-react-class');

var NoLink = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleChange}>;
  }
});
```

This works really well and it's very clear how data is flowing, however, with a lot of form fields it could get a bit verbose. Let's use `LinkedStateMixin` to save us some typing:

```
var createReactClass = require('create-react-class');

var WithLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')} />;
  }
});
```

`LinkedStateMixin` adds a method to your React component called `linkState()`. `linkState()` returns a `valueLink` object which contains the current value of the React state and a callback to change it.

`valueLink` objects can be passed up and down the tree as props, so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

Note that checkboxes have a special behavior regarding their `value` attribute, which is the value that will be sent on form submit if the checkbox is checked (defaults to `on`). The `value` attribute is not updated when the checkbox is checked or unchecked. For checkboxes, you should use `checkedLink` instead of `valueLink`:

```
<input type="checkbox" checkedLink={this.linkState('booleanValue')} />
```

Under the Hood

There are two sides to `LinkedStateMixin`: the place where you create the `valueLink` instance and the place where you use it. To prove how simple `LinkedStateMixin` is, let's rewrite each side separately to be more explicit.

valueLink Without LinkedStateMixin

```
var createReactClass = require('create-react-class');

var WithoutMixin = createReactClass({
```

```

getInitialState: function() {
  return {message: 'Hello!'};
},
handleChange: function(newValue) {
  this.setState({message: newValue});
},
render: function() {
  var valueLink = {
    value: this.state.message,
    requestChange: this.handleChange
  };
  return <input type="text" valueLink={valueLink} />;
}
);

```

As you can see, `valueLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkedStateMixin` is similarly simple: it just populates those fields with a value from `this.state` and a callback that calls `this.setState()`.

LinkedStateMixin Without valueLink

```

var LinkedStateMixin = require('react-addons-linked-state-mixin');
var createReactClass = require('create-react-class');

var WithoutLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={handleChange} />;
  }
});

```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

Immutability Helpers

Note:

`update` is a legacy add-on. Use `immutability-helper` instead.

Importing

```
import update from 'react-addons-update'; // ES6
var update = require('react-addons-update'); // ES5 with npm
```

Overview

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like [Clojure](#). However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's [Immutable-js](#) and the [Advanced Performance](#) section for more detail on Immutable-js.

The Main Idea

If you mutate data like this:

```
myData.x.y.z = 7;
// or...
myData.a.b.push(9);
```

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

```
const newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

```
const newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
```

```

}),
  a: extend(myData.a, {b: myData.a.b.concat(9)})
);

```

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

update() {#update}

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

```

import update from 'react-addons-update';

const newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});

```

While the syntax takes a little getting used to (though it's inspired by MongoDB's query language) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

Available Commands

- `{$push: array} push()` all the items in `array` on the target.
- `{$unshift: array} unshift()` all the items in `array` on the target.
- `{$splice: array of arrays}` for each item in `arrays` call `splice()` on the target with the parameters provided by the item.
- `{$set: any}` replace the target entirely.
- `{$merge: object}` merge the keys of `object` with the target.
- `{$apply: function}` passes in the current value to the function and updates it with the new returned value.

Examples

Simple push

```

const initialArray = [1, 2, 3];
const newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]

```

`initialArray` is still `[1, 2, 3]`.

Nested collections

```

const collection = [1, 2, {a: [12, 17, 15]}];
const newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]

```

This accesses `collection`'s index `2`, key `a`, and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14`.

Updating a value based on its current one

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
const newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

(Shallow) Merge

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

Add-Ons

Note:

`React.addons` entry point is deprecated as of React v15.5. The add-ons have moved to separate modules, and some of them have been deprecated.

The React add-ons are a collection of useful utility modules for building React apps. **These should be considered experimental** and tend to change more often than the core.

- `createFragment`, to create a set of externally-keyed children.

The add-ons below are in the development (unminified) version of React only:

- `Perf`, a performance profiling tool for finding optimization opportunities.
- `ReactTestUtils`, simple helpers for writing test cases.

Legacy Add-ons

The add-ons below are considered legacy and their use is discouraged. They will keep working in observable future, but there is no further development.

- `PureRenderMixin`. Use `React.PureComponent` instead.
- `shallowCompare`, a helper function that performs a shallow comparison for props and state in a component to decide if a component should update. We recommend using `React.PureComponent` instead.
- `update`. Use `kolodny/immutability-helper` instead.
- `ReactDOMFactories`, pre-configured DOM factories to make React easier to use without JSX.

Deprecated Add-ons

- `LinkedStateMixin` has been deprecated.
- `TransitionGroup` and `CSSTransitionGroup` have been deprecated in favor of [their drop-in replacements](#).

Using React with Add-ons

You can install the add-ons individually from npm (e.g. `npm install react-addons-create-fragment`) and import them:

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

When using React 15 or earlier from a CDN, you can use `react-with-addons.js` instead of `react.js`:

```
<script src="https://unpkg.com/react@15/dist/react-with-addons.js"></script>
```

The add-ons will be available via the `React.addons` global (e.g. `React.addons.TestUtils`).

[Go to TOC](#)

Liens CDN

React et ReactDOM sont tous deux accessibles depuis un CDN.

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

Les versions ci-dessus ne doivent être utilisées qu'à des fins de développement et ne sont pas adaptées à l'utilisation en production. Les versions minifiées et optimisées pour la production sont disponibles ici :

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

Pour charger une version spécifique de `react` et `react-dom`, il suffit de remplacer le `16` par la version souhaitée.

Pourquoi utiliser l'attribut `crossorigin` ?

Si vous chargez React depuis un CDN, nous vous recommandons de conserver l'attribut `crossorigin` :

```
<script crossorigin src="... "></script>
```

Nous vous conseillons également de vérifier que le CDN que vous utilisez définit bien l'en-tête HTTP `Access-Control-Allow-Origin: *` :

	Headers	Preview	Response	Timing
x	General			
	Request URL:	https://unpkg.com/react@16/umd/react.development.js		
	Request Method:	GET		
	Status Code:	304		
	Remote Address:	[2400:cb00:2048:1::681c]		
	Referrer Policy:	no-referrer-when-downgrade		
	Response Headers			
	access-control-allow-origin:	*		
	cache-control:	public, max-age=31536000		

`Access-Control-Allow-Origin: *`

Cela améliore l'[expérience de gestion des erreurs](#) avec React 16 et des versions ultérieures.

Découpage dynamique de code

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

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

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 intégrez 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

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

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>Chargement...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
);
```

Exports nommés

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"));
```

Aperçu du code source

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

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

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

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

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

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

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

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

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 [React-DOM 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

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

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

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

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 ?

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.

Composants et props

Les composants vous permettent de découper l'interface utilisateur en éléments indépendants et réutilisables, vous permettant ainsi de considérer chaque élément de manière isolée. Cette page fournit une introduction au concept de composants. Vous trouverez une [référence détaillée de l'API des composants ici](#).

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

Fonctions composants et composants à base de classes

Le moyen le plus simple de définir un composant consiste à écrire une fonction JavaScript :

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

Cette fonction est un composant React valide car elle accepte un seul argument « props » (qui signifie « propriétés ») contenant des données, et renvoie un élément React. Nous appelons de tels composants des « fonctions composants », car ce sont littéralement des fonctions JavaScript.

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

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

Les deux composants ci-dessus sont équivalents du point de vue de React.

Les composants (aussi bien fonctions que classes) possèdent quelques fonctionnalités supplémentaires dont nous discuterons dans les [prochaines sections](#).

Produire le rendu d'un composant

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

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

Mais ces éléments peuvent également représenter des composants définis par l'utilisateur :

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

Lorsque React rencontre un élément représentant un composant défini par l'utilisateur, il transmet les attributs JSX et les enfants à ce composant sous la forme d'un objet unique. Nous appelons cet objet « props ».

Par exemple, ce code affiche « Bonjour, Sara » sur la page :

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

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

[Essayer sur CodePen](#)

Récapitulons ce qui se passe dans cet exemple :

1. On appelle `ReactDOM.render()` avec l'élément `<Welcome name="Sara" />`.
2. React appelle le composant `Welcome` avec comme props `{name: 'Sara'}`.
3. Notre composant `Welcome` retourne un élément `<h1>Bonjour, Sara</h1>` pour résultat.
4. React DOM met à jour efficacement le DOM pour correspondre à `<h1>Bonjour, Sara</h1>`.

Remarque

React considère les composants commençant par des lettres minuscules comme des balises DOM. Par exemple, `<div />` représente une balise HTML div, mais `<Welcome />` représente un composant, et exige que l'identifiant `Welcome` existe dans la portée courante.

Pour en apprendre davantage sur le raisonnement qui sous-tend cette convention, lisez donc [JSX en profondeur](#).

Composition de composants

Les composants peuvent faire référence à d'autres composants dans leur sortie. Ça nous permet d'utiliser la même abstraction de composants pour n'importe quel niveau de détail. Un bouton, un formulaire, une boîte de dialogue, un écran : dans React, ils sont généralement tous exprimés par des composants.

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

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

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

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

[Essayer sur CodePen](#)

En règle générale, les nouvelles applications React ont un seul composant `App` à la racine. En revanche, si vous intégrez React à une application existante, vous pouvez commencer tout en bas par un petit composant comme `Button` et remonter progressivement la hiérarchie des vues.

Extraire des composants

N'ayez pas peur de scinder des composants en composants plus petits.

Prenons par exemple 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>
  );
}
```

[Essayer sur CodePen](#)

Il accepte comme props `author` (un objet), `text` (une chaîne de caractères) et `date` (une date), et décrit un commentaire sur un réseau social en ligne.

Les nombreuses imbrications au sein du composant le rendent difficile à maintenir, et nous empêchent d'en réutiliser des parties individuelles. Essayons donc d'en extraire quelques composants.

Pour commencer, 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 figure dans un composant `Comment`. C'est pourquoi nous avons donné à sa prop un nom plus générique : `user` plutôt que `author`.

Nous vous recommandons de nommer les props du point de vue du composant plutôt que de celui du contexte dans lequel il est utilisé.

On peut maintenant simplifier un poil `Comment` :

```

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>
    );
}

```

Ensuite, nous allons extraire un composant `UserInfo` qui affiche 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>
  );
}

```

Ce qui nous permet de simplifier encore davantage `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>
  );
}

```

[Essayer sur CodePen](#)

Au début, extraire des composants peut vous sembler fastidieux, mais disposer d'une palette de composants réutilisables s'avère rentable sur des applications de plus grande taille. En règle générale, si une partie de votre interface utilisateur est utilisée plusieurs fois (`Button`, `Panel`, `Avatar`) ou si elle est suffisamment complexe en elle-même (`App`, `FeedStory`, `Comment`), c'est un bon candidat pour une extraction en tant que composant séparé.

Les props sont en lecture seule

Que vous déclariez un composant [sous forme de fonction ou de classe](#), il ne doit jamais modifier ses propres props. Considérons cette fonction `sum` :

```

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

```

Ces fonctions sont dites « [pures](#) » parce qu'elles ne tentent pas de modifier leurs entrées et retournent 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 plutôt flexible mais applique une règle stricte :

Tout composant React doit agir comme une fonction pure vis-à-vis de ses props.

Bien entendu, les interfaces utilisateurs des applications sont dynamiques et évoluent dans le temps. Dans la [prochaine section](#), nous présenterons un nouveau concept « d'état local ». L'état local permet aux composants React de modifier leur sortie au fil du temps en fonction des actions de l'utilisateur, des réponses réseau et de n'importe quoi d'autre, mais sans enfreindre cette règle.

[Go to TOC](#)

Composition ou héritage ?

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

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

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 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 ?

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.

Adopter le mode concurrent (expérimental)

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.

- Installation
 - À qui s'adresse cette version expérimentale ?
 - Activer le mode concurrent
- À quoi s'attendre ?
 - Étape de migration : le mode bloquant
 - Pourquoi tant de modes ?
 - Comparaison des fonctionnalités

Installation

Le mode concurrent est disponible uniquement dans les [builds expérimentaux](#) de React. Pour les installer, exécutez :

```
npm install react@experimental react-dom@experimental
```

Les builds expérimentaux n'offrent aucune des garanties de la gestion sémantique des versions. Nous sommes susceptibles d'ajouter, modifier ou retirer des API dans n'importe quelle version `@experimental`.

Les versions expérimentales rompront souvent la compatibilité ascendante.

Vous pouvez essayer ces builds sur des projets personnels ou dans une branche, mais nous déconseillons leur utilisation en production. Chez Facebook, nous les utilisons *effectivement* en production, mais uniquement parce que nous sommes à même d'en corriger les bugs immédiatement. Vous voilà averti·e !

À qui s'adresse cette version expérimentale ?

Cette version est surtout destinée aux personnes habituées à adopter les nouvelles technologies très tôt, aux mainteneurs de bibliothèques et, de façon plus générale, aux personnes curieuses.

Nous utilisons ce code en production (et ça fonctionne pour nous) mais il reste quelques bugs, des fonctionnalités manquantes, et des lacunes dans la documentation. Nous sommes avides de vos retours sur ce qui casse en mode concurrent, afin que nous puissions mieux le préparer pour sa sortie prochaine au sein d'une version stable.

Activer le mode concurrent

En temps normal, quand nous ajoutons des fonctionnalités à React, vous pouvez vous en servir immédiatement. Les fragments, les Contextes ou même les Hooks sont autant d'exemples récents. Vous pouvez les utiliser dans du nouveau code sans avoir à changer quoi que ce soit au code existant.

Il en va différemment pour le mode concurrent. Il introduit des changements sémantiques dans le fonctionnement de React. Si ce n'était pas le cas, les [nouvelles fonctionnalités](#) qu'il permet *ne seraient pas possibles*. C'est pourquoi nous les avons regroupées dans un « mode » au lieu de les sortir, une à une, en isolation.

Vous ne pouvez pas activer le mode concurrent seulement pour une partie de l'arborescence React. Au lieu de ça, pour l'activer, vous devez le faire à l'endroit où, aujourd'hui, vous appelez `ReactDOM.render()`.

Voici comment activer le mode concurrent pour toute l'arborescence de `<App />` :

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

// Si vous aviez auparavant :
// ReactDOM.render(<App />, document.getElementById('root'));
// Vous pouvez désormais activer le mode concurrent en écrivant :

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

Remarque

Les API du mode concurrent, telles que `createRoot`, n'existent que dans les builds expérimentaux de React.

En mode concurrent, les méthodes de cycle de vie qui étaient [auparavant désignées](#) comme « dangereuses » (*"unsafe"*, NdT) sont *effectivement* dangereuses, et peuvent entraîner des bugs encore plus souvent que dans le code React habituel. Nous vous déconseillons de tester le mode concurrent tant que votre appli n'est pas compatible avec le [mode strict](#).

À quoi s'attendre ?

Si vous avez une grosse appli existante, ou si votre appli dépend de nombreux modules tiers, ne vous attendez pas à pouvoir utiliser le mode concurrent immédiatement. **Par exemple, chez Facebook nous utilisons le mode concurrent sur le nouveau site web, mais nous n'avons pas l'intention de l'activer**

sur l'ancien site. C'est parce que notre ancien site utilise encore de nombreuses méthodes de cycle de vie classées dangereuses, dans son code produit comme dans des bibliothèques tierces, ainsi que diverses approches qui ne fonctionnent pas bien avec le mode concurrent.

L'expérience nous indique que la manière la plus simple de fonctionner en mode concurrent, c'est d'avoir du code qui utilise des approches React idiomatiques et ne repose pas sur des solutions externes de gestion de l'état. Dans les prochaines semaines, nous documenterons séparément les problèmes courants que nous avons rencontrés et leurs solutions.

Étape de migration : le mode bloquant

Le mode concurrent est sans doute une fausse bonne idée pour les bases de code anciennes. C'est pourquoi nous fournissons aussi un nouveau « mode bloquant » dans nos builds expérimentaux. Vous pouvez l'essayer en remplaçant `createRoot` par `createBlockingRoot`. Il ne fournit qu'un *petit sous-ensemble* des fonctionnalités du mode concurrent, mais il est plus proche de la façon dont React fonctionne aujourd'hui et peut vous faciliter la transition.

En résumé :

- **Mode historique :** `ReactDOM.render(<App />, rootNode)`. C'est le fonctionnement actuel de React. Nous n'avons pas l'intention de retirer le mode historique dans un avenir proche, mais il ne prendra pas en charge ces nouvelles fonctionnalités.
- **Mode bloquant :** `ReactDOM.createBlockingRoot(rootNode).render(<App />)`. Il est expérimental pour le moment. Il est pensé comme une première étape de migration pour les applis qui veulent bénéficier d'au moins certaines fonctionnalités du mode concurrent.
- **Mode concurrent :** `ReactDOM.createRoot(rootNode).render(<App />)`. Il est expérimental pour le moment. À l'avenir, une fois qu'il sera stabilisé, nous comptons en faire le mode par défaut de React. Ce mode active *toutes* les nouvelles fonctionnalités.

Pourquoi tant de modes ?

Nous estimons qu'il est préférable de proposer une [stratégie de migration graduelle](#) plutôt que de faire d'énormes ruptures de compatibilité ascendante—qui scléroseraient React jusqu'à le rendre hors-sujet.

En pratique, nous pensons que la plupart des applis utilisant aujourd'hui le mode historique devraient pouvoir migrer vers au moins le mode bloquant (voire le mode concurrent). Cette fragmentation peut être irritante pour les bibliothèques qui essaient de prendre en charge l'ensemble des modes sur le court terme. Toutefois, éloigner progressivement l'écosystème du mode historique va aussi résoudre des problèmes qui affectent des bibliothèques de premier plan dans l'écosystème React, telles que [des comportements déroutants de Suspense lorsqu'on lit la mise en page](#) et [le manque de garanties stables de traitement par lot](#). Un certain nombre de bugs ne peuvent pas être corrigés en mode historique, sans changement de sémantique, mais n'existent pas dans les modes bloquant et concurrent.

Pensez au mode bloquant comme à une version en « gracieusement dégradée » du mode concurrent. **Résultat, sur le long terme nous devrions pouvoir converger et totalement cesser de nous préoccuper des différents modes.** Mais pour le moment, les modes constituent une importante stratégie de migration. Ils permettent à chacun·e de décider si la migration vaut le coup, et de réaliser la mise à jour à leur propre rythme.

Comparaison des fonctionnalités

	Mode histo.	Mode bloquant	Mode concurrent
Refs de type string	✓	🚫 **	🚫 **
API historique de Contexte	✓	🚫 **	🚫 **
<code>findDOMNode</code>	✓	🚫 **	🚫 **
Suspense	✓	✓	✓
<code>SuspenseList</code>	🚫	✓	✓
Suspense côté serveur + Hydratation	🚫	✓	✓
Hydratation progressive	🚫	✓	✓
Hydratation sélective	🚫	🚫	✓
Multitâches coopératif	🚫	🚫	✓
Regroupement automatique de multiples <code>setState</code>	🚫 *	✓	✓
Rendu basé sur priorités	🚫	🚫	✓
Prérendu interruptible	🚫	🚫	✓
<code>useTransition</code>	🚫	🚫	✓
<code>useDeferredValue</code>	🚫	🚫	✓
« Train » de révélations de Suspense	🚫	🚫	✓

* : le mode historique regroupe automatiquement les événements gérés par React, mais il est limité à une tâche navigateur. Les événements non-React doivent le demander explicitement en appelant `unstable_batchedUpdates`. Dans les modes bloquant et concurrent, tous les `setState`s sont traités par lot par défaut.

** : affiche des avertissements en développement.

Introduction au mode concurrent (expérimental)

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 ?

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

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

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 lisant 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

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

Résumons les deux exemples ci-dessus 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

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

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 distinctes 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.

[Go to TOC](#)

Approches pour une UI concurrente (expérimental)

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

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

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

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

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 ?

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 [démonstration](#), 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

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

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>
  );
}

```

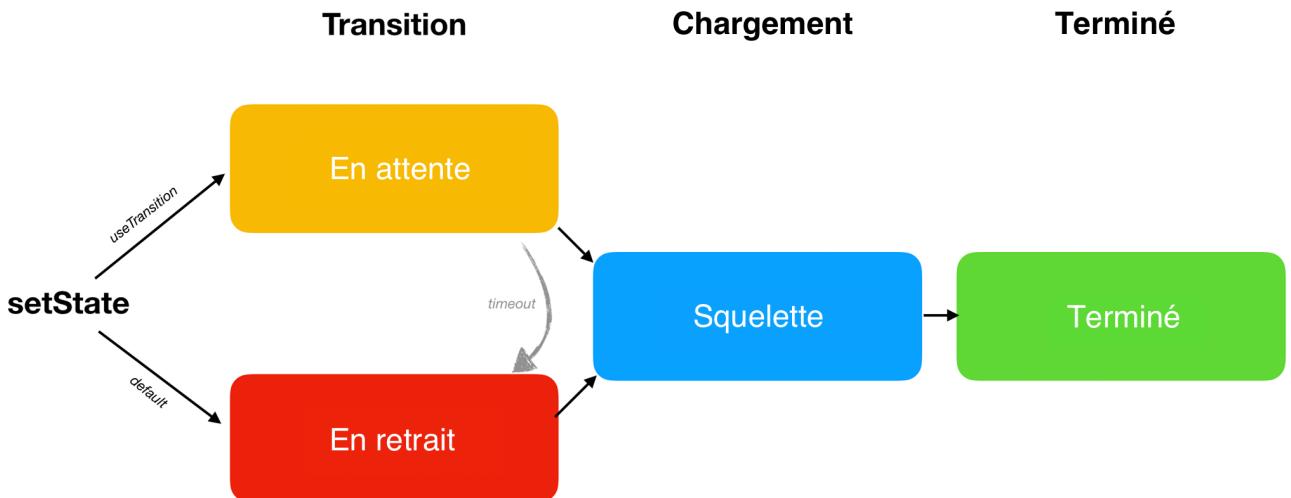
[Essayez sur CodeSandbox](#)

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

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

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

Quand nous utilisons `useTransition`, React nous permet de « rester » sur l'écran précédent—et 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 préférera 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 garantissez 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

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

Notre composant `Button` affichera immédiatement l'indicateur d'attente une fois cliqué :

```
function Button({ children, onClick }) {
  const [startTransition, isPending] = useTransition({
    timeoutMs: 1000
  });

  // ...

  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

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

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

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>Chargement...</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>Chargement...</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

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 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 :{" "}
        <input value={text} onChange={handleChange} />
      </label>
    </div>
  );
}

```

```

    ...
    <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 nœuds 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 nœuds `<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

Le mode concurrent offre un puissant modèle de programmation d'UI et un jeu de nouvelles primitives composites 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.

Référence de l'API du mode concurrent (expérimental)

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

`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> {#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 continuons 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 enrobez 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
necessaire */}
    <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 différés 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.

[Go to TOC](#)

Suspense pour le chargement de données (expérimental)

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 ?

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

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 utilisez 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

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

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 ?

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

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

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 re-tente 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)

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>Chargement 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**)

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)

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

ment 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

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

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 préemption 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

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>
  );
}
```

```

        })
    </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

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

Reprenons 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

Quand nous écrivons du code à base de promesses (`Promise`), nous pouvons recourir à `catch()` pour en 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

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](#).

Affichage conditionnel

En React, vous pouvez concevoir des composants distincts qui encapsulent le comportement voulu. Vous pouvez alors n'afficher que certains d'entre eux, suivant l'état de votre application.

L'affichage conditionnel en React fonctionne de la même façon que les conditions en Javascript. On utilise l'instruction Javascript `if` ou l'[opérateur ternaire](#) pour créer des éléments représentant l'état courant, et on laisse React mettre à jour l'interface utilisateur (UI) pour qu'elle corresponde.

Considérons ces deux composants :

```
function UserGreeting(props) {
  return <h1>Bienvenue !</h1>;
}

function GuestGreeting(props) {
  return <h1>Veuillez vous inscrire.</h1>;
}
```

Nous allons créer un composant `Greeting` qui affiche un de ces deux composants, selon qu'un utilisateur est connecté ou non :

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // Essayez de changer ça vers isLoggedIn={true} :
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

[Essayer sur CodePen](#)

Cet exemple affiche un message différent selon la valeur de la prop `isLoggedIn`.

Variables d'éléments

Vous pouvez stocker les éléments dans des variables. Ça vous aide à afficher de façon conditionnelle une partie du composant tandis que le reste ne change pas.

Prenons ces deux nouveaux composants, qui représentent les boutons de Déconnexion et de Connexion :

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Connexion
    </button>
  );
}
```

```

}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Déconnexion
    </button>
  );
}

```

Dans l'exemple ci-dessous, nous allons créer un [composant à état](#) appelé `LoginControl`.

Il affichera soit `<LoginButton />`, soit `<LogoutButton />`, selon son état courant. Il affichera aussi un `<Greeting />` de l'exemple précédent :

```

class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

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

```

[Essayer sur CodePen](#)

Même si déclarer une variable et utiliser une instruction `if` reste une bonne façon d'afficher conditionnellement un composant, parfois vous voudrez peut-être utiliser une syntaxe plus concise. Nous allons voir, ci-dessous, plusieurs façons d'utiliser des conditions à la volée en JSX.

Condition à la volée avec l'opérateur logique `&&` {#inline-if-with-logical--operator}

Vous pouvez utiliser n'importe quelle expression dans du JSX en l'enveloppant dans des accolades. Ça vaut aussi pour l'opérateur logique Javascript `&&`. Il peut être pratique pour inclure conditionnellement un élément :

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Bonjour !</h1>
      {unreadMessages.length > 0 &&
        <h2>
          Vous avez {unreadMessages.length} message(s) non-lu(s).
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

[Essayer sur CodePen](#)

Ça fonctionne parce qu'en JavaScript, `true && expression` est toujours évalué à `expression`, et `false && expression` est toujours évalué à `false`.

Du coup, si la condition vaut `true`, l'élément juste après `&&` sera affiché. Si elle vaut `false`, React va l'ignorer et le sauter.

Alternative à la volée avec opérateur ternaire

Une autre méthode pour l'affichage conditionnel à la volée d'éléments consiste à utiliser l'opérateur ternaire Javascript `condition ? trueValue : falseValue`.

Dans l'exemple ci-dessous, on l'utilise pour afficher conditionnellement un bloc de texte.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      L'utilisateur <b>{isLoggedIn ? 'est actuellement' : 'n'est pas'}</b>
    connecté.
    </div>
  );
}
```

On peut aussi l'utiliser pour des expressions plus longues, même si c'est moins clair :

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}
```

Tout comme en Javascript, c'est à vous de choisir un style approprié selon les préférences de lisibilité en vigueur pour vous et votre équipe. Souvenez-vous aussi que chaque fois que des conditions deviennent trop complexes, c'est peut-être le signe qu'il serait judicieux d'en [extraire un composant](#).

Empêcher l'affichage d'un composant

Dans de rares cas, vous voudrez peut-être qu'un composant se masque alors même qu'il figure dans le rendu d'un autre composant. Pour ce faire, il suffit de renvoyer `null` au lieu de son affichage habituel.

Dans l'exemple ci-dessous, `<WarningBanner />` s'affichera en fonction de la valeur de la prop `warn`. Si la valeur est `false`, le composant ne s'affiche pas :

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Attention !
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Masquer' : 'Afficher'}
        </button>
      </div>
    );
  }
}
```

```
        </div>
    );
}

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

Essayer sur [CodePen](#)

Renvoyer `null` depuis la méthode `render` d'un composant n'affecte pas l'appel aux méthodes du cycle de vie du composant. Par exemple, `componentDidUpdate` sera quand même appelée.

[Go to TOC](#)

Contexte

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

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

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

`React.createContext` [\[#reactcreatecontext\]](#)

```

const MyContext = React.createContext(defaultValue);

```

Crée un objet Contexte. 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

Contexte dynamique

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

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

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

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

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](#).

[Go to TOC](#)

Créer une nouvelle appli React

Utilisez une boîte à outils intégrée pour la meilleure expérience utilisateur et développeur possible.

Cette page décrit quelques boîtes à outils populaires qui facilitent les tâches telles que :

- La montée à l'échelle avec de nombreux fichiers et composants.
- L'utilisation de bibliothèques tierces depuis npm.
- La détection précoce des erreurs courantes.
- L'édition à la volée du CSS et du JS en développement.
- L'optimisation pour la production.

Les boîtes à outils recommandées sur cette page **ne nécessitent aucune configuration pour démarrer**.

You n'avez peut-être pas besoin d'une boîte à outils

Si vous ne rencontrez pas les problèmes décrits ci-dessus ou si vous n'êtes pas encore à l'aise avec l'utilisation d'outils JavaScript, envisagez [d'ajouter React comme une simple balise <script> sur une page HTML](#), éventuellement [avec du JSX](#).

C'est également **la façon la plus simple d'intégrer React au sein d'un site web existant**. Vous pourrez toujours étendre votre outillage si ça vous semble utile !

Boîtes à outils recommandées

L'équipe React recommande en premier lieu ces solutions :

- Si vous **apprenez React** ou **créez une nouvelle application web monopage**, alors utilisez [Create React App](#).
- Si vous construisez un **site web rendu côté serveur avec Node.js**, essayez [Next.js](#).
- Si vous construisez un **site web statique orienté contenu**, essayez [Gatsby](#).
- Si vous construisez une **bibliothèque de composants** ou une **intégration avec du code déjà existant**, essayez [des boîtes à outils plus flexibles](#).

Create React App

[Create React App](#) est un environnement confortable pour **apprendre React**, et constitue la meilleure option pour démarrer **une nouvelle application web monopage** en React.

Il configure votre environnement de développement de façon à vous permettre d'utiliser les dernières fonctionnalités de JavaScript, propose une expérience développeur agréable et optimise votre application pour la production. Vous aurez besoin de [Node >= 8.10 et de npm >= 5.6](#) sur votre machine. Pour créer un projet, exécutez :

```
npx create-react-app mon-app
cd mon-app
npm start
```

Remarque :

`npx` sur la première ligne n'est pas une faute de frappe -- c'est un [exécuteur de paquets qui est inclus dans npm 5.2+](#).

Create React App ne prend pas en charge la logique côté serveur ni les bases de données ; il crée simplement une chaîne de construction pour la partie frontale, de sorte que vous pouvez utiliser le serveur de votre choix. Sous le capot, il utilise [Babel](#) et [webpack](#), mais vous n'avez pas besoin de connaître ces outils.

Lorsque vous êtes prêt·e à déployer en production, exécutez `npm run build` pour créer une version optimisée de votre application dans le répertoire `build`. Vous pouvez en apprendre davantage sur Create React App [dans son README](#) et son [guide utilisateur](#).

Next.js

[Next.js](#) est un framework populaire et léger pour les **applications statiques rendues côté serveur** construites avec React. Il fournit des solutions prêtes à l'emploi pour **les styles et le routage**, et suppose que vous utilisez [Node.js](#) comme environnement serveur.

Apprenez Next.js grâce à [son guide officiel](#).

Gatsby

[Gatsby](#) est la meilleure option pour créer des **sites web statiques** avec React. Il vous permet d'utiliser des composants React, mais génère du HTML et du CSS pré-rendus afin de garantir le temps de chargement le plus rapide.

Apprenez Gatsby grâce à [son guide officiel](#) et à une [collection de kits de démarrage](#).

Boîtes à outils plus flexibles

Les boîtes à outils suivantes offrent plus de flexibilité et de choix. Nous les recommandons pour les utilisateurs expérimentés :

- [Neutrino](#) combine la puissance de [webpack](#) avec la simplicité des préréglages. Il inclut un préréglage pour les [applications React](#) et les [composants React](#).
- [nwb](#) est particulièrement utile pour [publier des composants React sur npm](#). Il peut également être utilisé pour créer des applications React.
- [Nx](#) est une boîte à outils pour le développements de monorepos full-stack, avec une prise en charge intégrée de React, next.js, Express et plus encore.
- [Parcel](#) est un *bundler* d'applications web rapide et sans configuration qui [fonctionne avec React](#).
- [Razzle](#) est un framework de rendu côté serveur qui ne requiert aucune configuration, mais offre plus de flexibilité que Next.js.

Créer une boîte à outils à partir de zéro

Une boîte à outils de construction en JavaScript comprend généralement :

- Un **gestionnaire de paquets**, tel que [Yarn](#) ou [npm](#). Il vous permet de tirer parti d'un vaste écosystème de paquets tiers, et de les installer ou les mettre à jour facilement.
- Un **bundler**, tel que [webpack](#) ou [Parcel](#). Il vous permet d'écrire du code modulaire et de le regrouper en petits paquets permettant d'optimiser le temps de chargement.
- Un **compilateur** tel que [Babel](#). Il vous permet d'écrire du JavaScript moderne qui fonctionnera quand même dans les navigateurs les plus anciens.

Si vous préférez configurer votre propre boîte à outils JavaScript à partir de zéro, [jetez un œil à ce guide](#) qui re-crée certaines des fonctionnalités de Create React App.

Pensez à vous assurer que votre outillage personnalisé [est correctement configuré pour la production](#).

Cross-origin Errors

Note:

The following section applies only to the development mode of React. Error handling in production mode is done with regular try/catch statements.

In [development mode](#), React uses a global `error` event handler to preserve the "pause on exceptions" behavior of browser DevTools. It also logs errors to the developer console.

If an error is thrown from a [different origin](#) the browser will mask its details and React will not be able to log the original error message. This is a security precaution taken by browsers to avoid leaking sensitive information.

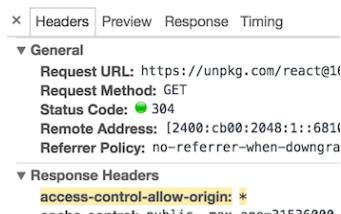
You can simplify the development/debugging process by ensuring that errors are thrown with a same-origin policy. Below are some common causes of cross-origin errors and ways to address them.

CDN

When loading React (or other libraries that might throw errors) from a CDN, add the `crossorigin` attribute to your `<script>` tags:

```
<script crossorigin src="..."></script>
```

Also ensure the CDN responds with the `Access-Control-Allow-Origin: *` HTTP header:



`Access-Control-Allow-Origin: *`

Webpack

Source maps

Some JavaScript bundlers may wrap the application code with `eval` statements in development. (For example Webpack will do this if `devtool` is set to any value containing the word "eval".) This may cause errors to be treated as cross-origin.

If you use Webpack, we recommend using the `cheap-module-source-map` setting in development to avoid this problem.

Code splitting

If your application is split into multiple bundles, these bundles may be loaded using JSONP. This may cause errors thrown in the code of these bundles to be treated as cross-origin.

To resolve this, use the `crossOriginLoading` setting in development to add the `crossorigin` attribute to the `<script>` tags generated for the JSONP requests.

Principes de conception

Nous avons écrit ce document afin que vous ayez une meilleure idée de la façon dont nous décidons ce que React fait et ne fait pas, et de notre philosophie de développement. Bien que nous adorions les contributions de la communauté, il est peu probable que nous choisissons un chemin qui enfreindrait un ou plusieurs de ces principes.

Remarque

Ce document suppose une solide compréhension de React. Il décrit les principes de conception de *React lui-même*, et non des composants ou des applications React.

Pour une introduction à React, consultez plutôt [Penser en React](#).

Composition

La caractéristique principale de React est la composition de composants. Les composants écrits par des personnes différentes devraient fonctionner correctement ensemble. Nous tenons à ce que vous puissiez ajouter des fonctionnalités à un composant sans que ça impacte le reste de la base de code.

Par exemple, il devrait être possible d'introduire un état local dans un composant sans changer les composants qui l'utilisent. Dans le même esprit, il devrait être possible d'ajouter du code d'initialisation et de nettoyage à n'importe quel composant lorsque c'est nécessaire.

Il n'y a rien de « mauvais » à utiliser les méthodes d'état ou de cycle de vie dans les composants. Comme n'importe quelle fonctionnalité puissante, elles devraient être utilisées avec modération, mais nous n'avons aucune intention de les retirer. Au contraire, nous pensons qu'elles font partie intégrante de ce qui rend React utile. Nous ajouterons peut-être [des approches plus fonctionnelles](#) (en anglais) à l'avenir, mais les méthodes d'état local et de cycle de vie feront partie de ce modèle.

Les composants sont souvent décrits comme de « simples fonctions » mais à nos yeux ils doivent être bien plus que ça pour être utiles. Avec React, les composants décrivent un comportement composable, ce qui inclut le rendu, le cycle de vie et l'état local. Certaines bibliothèques externes telles que [Relay](#) ajoutent d'autres responsabilités aux composants comme la description des dépendances de données. Il est possible que ces idées reviennent dans React sous une forme ou une autre.

Abstraction commune

En général, nous [évitons d'ajouter des fonctionnalités](#) (vidéo en anglais) pouvant être mises en œuvre par les utilisateurs. Nous ne voulons pas surcharger vos applications avec du code inutile. Il y a toutefois des exceptions à ça.

Par exemple, si React ne proposait pas des méthodes pour l'état local ou le cycle de vie, les utilisateurs créeraient leurs propres abstractions personnalisées pour ça. Quand plusieurs abstractions s'affrontent, React ne peut garantir ni bénéficier des propriétés de l'une ou l'autre. Il doit se contenter du plus petit dénominateur commun.

C'est la raison pour laquelle nous ajoutons parfois des fonctionnalités directement à React. Si nous constatons que de nombreux composants implémentent une certaine fonctionnalité de façons incompatibles ou inefficaces, nous préférons parfois l'intégrer à React. Nous ne le faisons pas à la légère. Lorsque nous le faisons, c'est parce que nous sommes convaincus qu'élever le niveau d'abstraction profite à l'ensemble de l'écosystème. L'état local, les méthodes de cycle de vie ou la normalisation des événements des navigateurs en sont de bons exemples.

Nous discutons toujours de ces propositions d'amélioration avec la communauté. Vous pouvez trouver certaines de ces discussions avec l'étiquette "[big picture](#)" dans le suivi des problèmes et questions de React.

Échappatoires

React est pragmatique. Il est piloté par les besoins des produits développés chez Facebook. Bien qu'il soit influencé par certains paradigmes qui ne sont pas tout à fait populaires comme la programmation fonctionnelle, le rendre accessible à un large public de développeurs aux compétences et expériences variées est un objectif affiché du projet.

Si nous voulons déprécier une approche que nous n'aimons pas, il est de notre responsabilité de considérer tous ses cas d'usage et d'[éduquer la communauté sur les alternatives](#) avant de la déprécier. Si une approche utile pour la création d'applications est difficile à exprimer de manière déclarative, nous lui [fournirons une API impérative](#). Si nous ne parvenons pas à trouver l'API parfaite pour une fonctionnalité que nous estimons nécessaire dans de nombreuses applications, nous [fournirons une API temporaire de moindre qualité](#) du moment qu'il reste possible de nous en débarrasser ultérieurement et qu'elle laisse la porte ouverte à de futures améliorations.

Stabilité

Nous accordons de l'importance la stabilité de l'API. Chez Facebook, nous avons plus de 50 000 composants utilisant React. De nombreuses autres sociétés, telles que [Twitter](#) ou [Airbnb](#), sont également de grandes utilisatrices de React. C'est pour ça que nous sommes généralement réticents à changer les API ou les comportements publics.

Cependant, nous pensons que la stabilité au sens où « rien ne change » est surfait. Ça finit vite par stagner. Nous préférons plutôt la stabilité au sens où « c'est largement utilisé en production et lorsque quelque chose change, il existe un chemin de migration clair (et de préférence automatisé) ».

Lorsque nous dépréciions une approche, nous étudions son utilisation interne chez Facebook et nous ajoutons des avertissements de dépréciation. Ils nous permettent de mesurer l'impact du changement. Parfois nous renonçons quand nous voyons que c'est encore trop tôt, et nous réfléchissons de manière plus stratégique sur la façon de préparer les bases de code à ce changement.

Si nous sommes convaincus que le changement n'est pas trop disruptif et que la stratégie de migration est viable pour tous les cas d'usage, nous livrons les avertissements de dépréciation à la communauté open source. Nous sommes en contact étroit avec de nombreux utilisateurs de React en dehors de Facebook, nous surveillons les projets open source populaires et les aidons à corriger ces dépréciations.

Compte tenu de la taille considérable de la base de code React chez Facebook, la réussite de la migration en interne est généralement un bon indicateur du fait que les autres sociétés n'auront pas de problèmes non plus. Néanmoins, il arrive que des personnes nous signalent des cas d'usage auxquels nous n'avons pas pensé, et nous ajoutons alors des solutions de contournement pour elles ou repensons notre approche.

Nous ne dépréciions rien sans une bonne raison. Nous reconnaissons que les avertissements de dépréciation sont parfois frustrants, mais nous les ajoutons car les dépréciations permettent de préparer le terrain à des améliorations ou de nouvelles fonctionnalités que la communauté et nous-mêmes estimons utiles.

Par exemple, nous avons ajouté un [avertissement concernant les props DOM inconnues](#) dans React 15.2.0. De nombreux projets en furent impactés. Cependant, corriger cet avertissement est important pour pouvoir introduire la prise en charge des [attributs personnalisés](#) dans React. Il y a une raison comme celle-ci derrière chaque dépréciation que nous ajoutons.

Lorsque nous ajoutons un avertissement de dépréciation, nous le conservons pour le reste de la version majeure en cours, et nous [changeons le comportement lors de la version majeure suivante](#). S'il y a beaucoup de travail manuel répétitif à la clé, nous publions un script [codemod](#) (vidéo en anglais) qui automatise la plus grande partie des ajustements nécessaires. Les *codemods* nous permettent d'avancer sans stagner sur une base de code importante, et nous vous encourageons à les utiliser également.

Vous trouverez les codemods que nous publions dans le dépôt [react-codemod](#).

Interopérabilité

Nous accordons une grande importance à l'interopérabilité avec les systèmes existants et à l'adoption progressive. Facebook a une importante base de code sans React. Son site web utilise à la fois un système de composants côté serveur appelé XHP, des bibliothèques d'interface utilisateur (*UI*) internes antérieures à React et React lui-même. Il est important pour nous que n'importe quelle équipe produis puisse [commencer à utiliser React pour une petite fonctionnalité](#) (vidéo en anglais) plutôt que de réécrire leur code pour pouvoir en bénéficier.

C'est pour ça que React propose des échappatoires pour fonctionner avec des modèles mutables, et essaie de fonctionner correctement avec d'autres bibliothèques d'*UI*. Vous pouvez enrober une *UI* impérative existante dans un composant déclaratif, et inversement. C'est crucial pour une adoption progressive.

Planification

Même si vos composants sont décrits comme des fonctions, en utilisant React vous ne les appelez pas directement. Chaque composant renvoie une [description de ce qui doit être affiché](#), et cette description peut inclure à la fois des composants écrits par l'utilisateur comme `<LikeButton>` et des composants spécifiques à la plate-forme tels que `<div>`. React doit se charger de « dérouler » `<LikeButton>` plus tard et d'appliquer récursivement à l'arbre de l'*UI* les modifications apportées par les rendus des composants.

Il s'agit d'une distinction subtile mais puissante. Puisque c'est React qui appelle cette fonction de composant à votre place, ça signifie que React peut différer son appel si nécessaire. Dans son implémentation actuelle, React parcourt l'arbre de façon récursive et appelle les fonctions de rendu de l'ensemble de l'arbre mis à jour en une seule passe. À l'avenir cependant, [certaines mises à jour pourraient être différées pour conserver un affichage fluide](#).

C'est un thème courant dans la conception de React. Certaines bibliothèques populaires implémentent une approche "*push*" dans laquelle les calculs sont exécutés dès que de nouvelles données sont disponibles. React, quant à lui, s'en tient à l'approche "*pull*" où les calculs peuvent être différés au besoin.

React n'est pas une bibliothèque de traitement de données générique. C'est une bibliothèque pour construire des interfaces utilisateurs. Nous pensons que React est particulièrement bien placé dans une application pour savoir quels calculs sont pertinents à un instant précis et lesquels ne le sont pas.

Si quelque chose est en dehors de la zone affichée à l'écran, nous pouvons différer toute la logique qui y est associée. Si les données arrivent plus vite que la fréquence de rafraîchissement, nous pouvons fusionner et regrouper les mises à jour. Nous pouvons donner la priorité au travail résultant des interactions utilisateurs (telles qu'une animation déclenchée par le clic d'un bouton) plutôt qu'à une tâche de fond de moindre importance (telle que l'affichage d'un nouveau contenu fraîchement chargé depuis le réseau) afin d'éviter de produire un affichage saccadé.

Pour être clairs, nous n'en sommes pas encore là. Cependant, c'est pour avoir la liberté de faire quelque chose comme ça que nous préférons avoir le contrôle de la planification et que `setState()` est asynchrone. Sur un plan conceptuel, nous voyons ça comme la « planification d'une mise à jour ».

Il nous serait plus difficile de contrôler la planification si nous permettions à l'utilisateur de composer directement des vues avec le paradigme "*push*" commun à certaines variations de la [programmation fonctionnelle réactive](#). Nous voulons garder le contrôle du code intermédiaire qui sert de « colle ».

React a pour objectif-clé de minimiser la quantité de code utilisateur qui s'exécute avant de lui rendre la main. Ça garantit que React conserve la possibilité de planifier et scinder le travail en segments selon sa connaissance de l'UI.

Il y a une blague interne dans l'équipe selon laquelle React aurait dû s'appeler "*Schedule*" parce que React ne veut pas être totalement « réactif ».

Expérience de développement

Nous tenons à offrir une bonne expérience de développement.

Par exemple, nous maintenons les [React DevTools](#) qui vous permettent d'inspecter l'arbre des composants React dans Chrome et Firefox. Nous avons entendu dire que ça améliore considérablement la productivité des ingénieur·e·s de Facebook et de la communauté.

Nous essayons également de faire un effort supplémentaire pour fournir des avertissements utiles aux développeurs. Par exemple, React vous avertit durant le développement si vous imbriquez des balises d'une façon que le navigateur ne comprend pas, ou si vous faites une faute de frappe courante dans l'API. Les avertissements aux développeurs et les vérifications associées sont la raison principale pour laquelle la version de développement de React est plus lente que celle de production.

Les schémas d'utilisation que nous voyons en interne chez Facebook nous aident à comprendre quelles sont les erreurs courantes et comment les prévenir rapidement. Lorsque nous ajoutons de nouvelles fonctionnalités, nous essayons d'anticiper les erreurs courantes et vous avertissons à leur sujet.

Nous cherchons en permanence des moyens d'améliorer l'expérience de développement. Nous aimons recueillir vos suggestions et accepter vos contributions visant à la rendre encore meilleure.

Débogage

Lorsque quelque chose se passe mal, il est important que vous ayez des pistes pour remonter jusqu'à la source de l'erreur dans votre base de code. Avec React, ces pistes se basent sur les props et l'état local.

Si quelque chose ne va pas à l'écran, vous pouvez ouvrir React DevTools, trouver le composant responsable du rendu, et ainsi voir si les props et l'état sont corrects. Si c'est le cas, vous savez que le problème se situe dans la fonction `render()` ou dans une fonction appelée par `render()`. Le problème est isolé.

Si l'état est incorrect, vous savez que le problème est dû à l'un des appels à `setState()` au sein de ce fichier. C'est là aussi facile à localiser et à corriger car il y a généralement peu d'appels à `setState()` dans un même fichier.

Si les props sont incorrectes, vous pouvez remonter le long de l'arbre dans l'inspecteur à la recherche du composant qui est à l'origine des mauvaises props.

Cette capacité à relier n'importe quelle UI aux données qui l'ont générée est très importante dans React. Un objectif explicite de conception impose de ne pas « emprisonner » l'état au sein de fermetures lexicales (*closures*, *NdT*) ou de combinatoires, mais de le rendre plutôt disponible directement dans React.

Alors que l'UI est dynamique, nous pensons que les fonctions synchrones `render()` basées entièrement sur les props et l'état local rendent le débogage certes ennuyeux mais déterministe et bien délimité, plutôt que de virer au jeu de devinettes. Nous aimerions conserver cette contrainte dans React même si ça complexifie certains cas d'usage, tels que des animations complexes.

Configuration

Nous considérons les options de configuration d'exécution comme problématiques.

Par exemple, on nous demande parfois d'implémenter une fonction du genre `React.configure(options)` ou `React.register(component)`. Ça pose toutefois de nombreux problèmes, et nous ne connaissons pas de bonne solution.

Que se passerait-il si cette fonction était appelée depuis une bibliothèque de composants tierce ? Et si une application React intégrait une autre application React et que leurs configurations respectives étaient incompatibles ? Comment un composant tiers peut-il spécifier qu'il nécessite une configuration spécifique ? Nous pensons que la configuration globale ne fonctionne pas correctement avec la composition. Puisque la composition est au cœur de React, nous ne fournissons aucune configuration globale dans le code.

Nous fournissons cependant une configuration globale au niveau de la construction. Par exemple, nous offrons des versions distinctes pour le développement et la production. Nous pourrions également [ajouter une version de profilage](#) à l'avenir, et sommes ouverts à l'examen d'autres paramètres de construction.

Au-delà du DOM

Nous voyons l'utilité de React dans la mesure où il nous permet d'écrire des composants qui ont moins de bugs et se composent facilement entre eux. React visait originellement le DOM, mais [React Native](#) est tout aussi important pour Facebook et pour la communauté.

Être indépendant du moteur de rendu constitue une contrainte de conception importante pour React. Ça alourdit un peu la représentation interne. D'un autre côté, les améliorations apportées au noyau bénéficient à toutes les plates-formes.

Avoir un seul modèle de programmation nous permet de former des équipes d'ingénieur·e·s autour de produits plutôt que de plates-formes. Jusqu'à présent le jeu en vaut la chandelle.

Implémentation

Nous essayons de fournir des API élégantes dans la mesure du possible. Nous sommes beaucoup moins préoccupés par l'élégance de l'implémentation. Le monde réel est loin d'être parfait et, dans une mesure raisonnable, nous préférons mettre du code laid dans la bibliothèque si ça signifie que l'utilisateur n'aura pas à l'écrire. Lorsque nous évaluons du nouveau code, nous visons d'abord une implémentation correcte, performante et offrant une bonne expérience de développement. L'élégance est secondaire.

Nous préférons le code ennuyeux au code trop « malin ». Le code est jetable et change souvent. Il est donc important qu'il [n'introduise pas de nouvelles abstractions internes sauf si c'est absolument nécessaire](#) (vidéo en anglais). Un code verbeux qui est facile à déplacer, modifier et retirer reste préférable à un code élégant qui est prématurément abstrait et difficile à modifier.

Optimisé pour l'outillage

Certaines API couramment utilisées ont des noms à rallonge. Par exemple, nous utilisons `componentDidMount()` plutôt que `didMount()` ou `onMount()`. C'est [volontaire](#). L'objectif est de bien mettre en avant les points d'interactions avec la bibliothèque.

Dans une base de code aussi vaste que celle de Facebook, il est très important de pouvoir rechercher les utilisations d'API spécifiques. Nous privilégions les noms verbeux et distinctifs, tout particulièrement pour les fonctionnalités qui doivent être utilisées avec parcimonie. Par exemple, il est difficile de rater `dangerouslySetInnerHTML` lors d'une revue de code.

L'optimisation pour la recherche est également importante du fait de notre dépendance aux [codemods](#) (vidéo en anglais) pour exécuter des modifications qui rompent la compatibilité ascendante. Nous voulons que les changements automatiques sur la base de code soient faciles et sûrs, et des noms verbeux nous aident à y parvenir. De même, des noms aisément identifiables facilitent l'écriture de [règles d'analyse statique](#) personnalisées pour React sans avoir à se soucier de potentiels faux positifs.

[JSX](#) joue un rôle similaire. Bien qu'il ne soit pas nécessaire à React, nous l'utilisons beaucoup chez Facebook à la fois pour des raisons esthétiques et pragmatiques.

Dans notre base de code, JSX indique clairement aux outils qu'ils travaillent avec un arbre d'éléments React. Ça rend possible l'ajout à la construction d'optimisations telles que [l'extraction \(hoisting, NdT\)](#) des [éléments constants](#) (en anglais), l'utilisation sûre d'analyseurs statiques et de [codemods](#) internes, et [l'inclusion de l'emplacement dans le source JSX](#) au sein des messages d'avertissements.

Dogfooding {#dogfooding}

Nous faisons de notre mieux pour résoudre les problèmes soulevés par la communauté. Néanmoins, nous sommes susceptibles de prioriser les problèmes rencontrés aussi par les personnes en interne chez Facebook. Ça peut sembler contre-intuitif, mais nous pensons que c'est la raison principale pour laquelle la communauté peut miser sur React.

L'utilisation importante faite en interne nous conforte dans l'idée que React n'est pas près de disparaître. React a été créé chez Facebook pour résoudre ses problèmes. Il apporte une valeur métier tangible à la société et est utilisé dans bon nombre de ses produits. Notre [dogfooding](#) de React signifie que notre vision reste claire et que nous avons une direction bien établie pour l'avenir.

Ça ne signifie pas que nous ignorons les problèmes soulevés par la communauté. Par exemple, nous avons ajouté la prise en charge des [Web Components](#) et de [SVG](#) à React alors que nous n'utilisions ni l'un ni l'autre en interne. Nous [écoutons attentivement vos difficultés](#) et nous [y répondons](#) au mieux de nos capacités. La communauté est ce qui rend React si spécial pour nous et nous sommes honorés d'y contribuer en retour.

Après avoir livré de nombreux projets open source chez Facebook, nous avons appris qu'essayer de satisfaire tout le monde en même temps aboutissait à des projets mal ciblés et qui ne grandissaient pas. Au lieu de ça, nous avons constaté que choisir un petit public et s'efforcer de le satisfaire avait un net effet positif. C'est exactement ce que nous avons fait avec React et, jusqu'à présent, la résolution des problèmes rencontrés par les équipes de produits chez Facebook s'est bien transposée à la communauté open source.

L'inconvénient de cette approche est que parfois nous ne mettons pas assez l'accent sur les problèmes auxquels les équipes de Facebook n'ont pas à faire face, telles que l'expérience de « démarrage ». Nous en sommes parfaitement conscients, et nous réfléchissons aux façons de nous améliorer qui profiteraient à toute la communauté sans commettre les mêmes erreurs que nous avons faites sur de précédents projets open source.

Périmètres d'erreurs

Auparavant, les erreurs JavaScript au sein des composants avaient l'habitude de corrompre l'état interne de React, et de causer des [erreurs assez incompréhensibles](#) lors des rendus suivants. Ces erreurs étaient toujours causées par une erreur antérieure dans le code applicatif et comme React ne proposait alors aucun moyen de les gérer correctement dans les composants, il n'avait pas la possibilité de se rétablir.

L'arrivée des périmètres d'erreurs

Une erreur JavaScript au sein d'une partie de l'interface utilisateur (UI) ne devrait pas casser l'ensemble de l'application. Pour résoudre ce problème, React 16 a introduit un nouveau concept appelé « Périmètres d'erreurs » (*Error Boundaries*, NdT).

Les périmètres d'erreurs sont des composants React qui **interceptent les erreurs JavaScript n'importe où au sein de leur arbre de composants enfants, enregistrent ces erreurs, et affichent une UI de repli** à la place de l'arbre de composants qui a planté. Les périmètres d'erreurs interceptent les erreurs survenant au rendu, dans les méthodes de cycle de vie, ainsi que dans les constructeurs de tous les éléments de leur arborescence.

Remarque

Les périmètres d'erreurs n'interceptent **pas** les erreurs qui surviennent dans :

- Les gestionnaires d'événements ([en savoir plus](#)).
- Le code asynchrone (par exemple les fonctions de rappel de `setTimeout` ou `requestAnimationFrame`).
- Le rendu côté serveur.
- Les erreurs levées dans le composant du périmètre d'erreur lui-même (plutôt qu'au sein de ses enfants).

Une classe de composant devient un périmètre d'erreur si elle définit au moins une des méthodes de cycle de vie `static getDerivedStateFromError()` ou `componentDidCatch()`. Utilisez `static getDerivedStateFromError()` pour afficher une UI de repli lorsqu'une erreur est levée. Utilisez `componentDidCatch()` pour enregistrer les informations relatives à l'erreur.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Mettez à jour l'état, de façon à montrer l'UI de repli au prochain rendu.
    return { hasError: true };
  }
}
```

```

componentDidCatch(error, errorThrown) {
  // Vous pouvez aussi enregistrer l'erreur au sein d'un service de rapport.
  logErrorToMyService(error, errorThrown);
}

render() {
  if (this.state.hasError) {
    // Vous pouvez afficher n'importe quelle UI de repli.
    return <h1>Something went wrong.</h1>;
  }

  return this.props.children;
}
}

```

Vous pouvez alors l'utiliser comme un composant classique :

```

<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>

```

Les périmètres d'erreurs fonctionnent comme un bloc JavaScript `catch {}`, mais pour les composants. Seuls les composants à base de classe peuvent être des périmètres d'erreurs. En pratique, vous voudrez généralement définir un seul composant de périmètre d'erreur puis l'utiliser partout dans votre application.

Notez bien que **les périmètres d'erreurs ne détectent que les erreurs présentes en dessous d'eux dans l'arbre des composants**. Un périmètre d'erreur ne peut intercepter une erreur survenant dans son propre code. Si un périmètre d'erreur plante en tentant d'afficher son message d'erreur, l'erreur se propagera alors au périmètre d'erreur le plus proche au-dessus de lui dans l'arbre. Là aussi, c'est similaire à la façon dont le bloc `catch {}` fonctionne en JavaScript.

Démonstration interactive

Jetez un coup d'œil sur [cet exemple de déclaration et d'utilisation d'un périmètre d'erreur avec React 16](#).

Où placer les périmètres d'erreurs ?

La granularité des périmètres d'erreurs est à votre discréction. Vous pourriez enrober les composants racines de routage pour afficher à l'utilisateur un message du type « Quelque chose s'est mal passé », à l'image de ce qui est souvent fait par les frameworks côté serveur. Vous pourriez aussi enrober des éléments d'interface précis avec un périmètre d'erreur afin de les empêcher de planter le reste de l'application.

Nouveau comportement pour les erreurs non-rattrapées

Ce changement a un impact important. **À compter de React 16, les erreurs qui ne sont pas intercceptées par un périmètre d'erreur entraîneront le démontage de l'intégralité de l'arbre des composants.**

Cette décision a été débattue, mais l'expérience nous a montré qu'il est bien pire de laisser en place une interface corrompue que de la supprimer complètement. Par exemple, dans un produit tel que Messenger, laisser visible une interface dégradée peut amener l'utilisateur à envoyer un message à la mauvaise personne. De la même façon, pour une application de paiement, afficher un mauvais montant est bien pire que de ne rien afficher du tout.

Cette modification signifie que lorsque vous migrez vers React 16, vous découvrirez probablement des plantages dans votre application qui étaient jusque-là passés inaperçus. L'ajout de périmètres d'erreurs permet d'offrir une meilleure expérience utilisateur en cas de problème.

Par exemple, Facebook Messenger enrobe le contenu de la barre latérale, du panneau d'information, du journal de conversation, ainsi que de la saisie de message dans des périmètres d'erreurs distincts. Si l'un des composants de ces zones d'interface plante, les autres continueront de fonctionner normalement.

Nous vous encourageons également à utiliser des services de rapport d'erreurs JavaScript (ou à construire le vôtre) afin de mieux connaître les exceptions non gérées dès qu'elles apparaissent en production, et donc de pouvoir les corriger.

Traces de piles des composants

En mode développement, React 16 affiche dans la console toutes les erreurs qui apparaissent durant le rendu, même si l'application les cache accidentellement. En plus du message d'erreur et de la trace de pile (*stack trace, NDT*) JavaScript, il fournit également la trace de pile du composant. Vous pouvez désormais voir exactement où l'erreur est apparue dans l'arbre des composants :

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.
```

```
Error: I crashed!
```

```
The error is located at:
  in BuggyCounter (created by App)
  in ErrorBoundary (created by App)
  in div (created by App)
  in App
```

Vous pouvez également voir les noms des fichiers et les numéros de lignes dans la trace de pile du composant. C'est le fonctionnement par défaut pour les projets créés avec [Create React App](#) :

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.
```

```
Error: I crashed!
```

```
The error is located at:
  in BuggyCounter (at App.js:26)
  in ErrorBoundary (at App.js:21)
  in div (at App.js:8)
  in App (at index.js:5)
```

Si vous n'utilisez pas Create React App, vous pouvez ajouter [cette extension](#) manuellement dans votre configuration Babel. Remarquez que c'est conçu pour le développement et **ne doit pas être activé en production**.

Remarque

Les noms des composants affichés dans la trace de pile dépendent de la propriété `Function.name`. Si vous devez prendre en charge des navigateurs ou des dispositifs plus anciens qui ne proposent pas ça nativement (par exemple IE 11), vous pouvez envisager d'inclure le polyfill `function.name-polyfill` dans votre application. Autrement, vous pouvez également définir explicitement la propriété `displayName` sur tous vos composants.

Et pourquoi pas try / catch ?

Les `try` / `catch` sont super, mais ne marchent qu'avec du code impératif :

```
try {
  showButton();
} catch (error) {
  // ...
}
```

Mais les composants React sont déclaratifs et spécifient ce *qui* doit être rendu :

`<Button />`

Les périmètres d'erreurs respectent la nature déclarative de React, et se comportent sans surprises. Par exemple, même si une erreur survient dans une méthode `componentDidUpdate` suite à un `setState` quelque part au fin fond de l'arbre des composants, elle se propagera correctement jusqu'au périmètre d'erreur le plus proche.

Et les gestionnaires d'événements ?

Les périmètres d'erreurs n'interceptent **pas** les erreurs qui surviennent au sein des gestionnaires d'événements.

React n'a pas besoin de périmètres d'erreurs pour gérer les erreurs dans les gestionnaires d'événements. Contrairement aux méthodes de rendu ou de cycle de vie, les gestionnaires d'événements ne sont pas appelés pendant le rendu. Du coup même si ces gestionnaires lèvent une erreur, React saura tout de même quoi afficher à l'écran.

Si vous avez besoin d'intercepter une erreur au sein d'un gestionnaire d'événements, il suffit d'utiliser un classique `try` / `catch` JavaScript :

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
```

```
// Faites ici quelque chose qui va lever une erreur
} catch (error) {
  this.setState({ error });
}

render() {
  if (this.state.error) {
    return <h1>Une erreur a été interceptée.</h1>
  }
  return <button onClick={this.handleClick}>Cliquez ici</button>
}
```

Remarquez que l'exemple ci-dessus illustre un comportement JavaScript classique et n'utilise aucun périmètre d'erreur.

Changements de nommage par rapport à React 15

React 15 disposait d'une prise en charge très limitée des périmètres d'erreurs sous un nom de méthode différent : `unstable_handleError`. Cette méthode ne fonctionne plus, et vous devrez la remplacer par `componentDidCatch` dans votre code à partir de la première version bêta de React 16.

Pour ce changement, nous fournissons un [codemod](#) qui vous permet de migrer automatiquement votre code.

Error Decoder

In the minified production build of React, we avoid sending down full error messages in order to reduce the number of bytes sent over the wire.

We highly recommend using the development build locally when debugging your app since it tracks additional debug info and provides helpful warnings about potential problems in your apps, but if you encounter an exception while using the production build, this page will reassemble the original text of the error.

AJAX et les API

Comment faire un appel AJAX ?

Vous pouvez utiliser n'importe quelle bibliothèque AJAX de votre choix avec React. Parmi les plus populaires, on trouve [Axios](#), [jQuery AJAX](#), et le standard `window.fetch` intégré au navigateur.

Où dois-je faire mon appel AJAX dans le cycle de vie du composant ?

Vous devriez obtenir vos données via des appels AJAX dans la méthode de cycle de vie `componentDidMount`. De cette façon, vous pourrez y utiliser `setState` pour mettre à jour votre composant lorsque les données seront récupérées.

Exemple : utiliser des résultats AJAX pour initialiser un état local

Le composant ci-dessous montre comment faire un appel AJAX dans `componentDidMount` pour peupler l'état local d'un composant.

Dans l'exemple, l'API renvoie un objet JSON avec la structure suivante :

```
{
  "items": [
    { "id": 1, "name": "Apples", "price": "$2" },
    { "id": 2, "name": "Peaches", "price": "$5" }
  ]
}
```

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      isLoading: false,
      items: []
    };
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          this.setState({
            isLoading: true,
            items: result.items
          });
        },
        // Remarque : il est important de traiter les erreurs ici
        // au lieu d'utiliser un bloc catch(), pour ne pas passer à la trappe
        // des exceptions provenant de réels bugs du composant.
        (error) => {
          this.setState({
            isLoading: true,
            error
          });
        }
      );
  }
}
```

```

        )
    }

    render() {
        const { error, isLoading, items } = this.state;
        if (error) {
            return <div>Erreur : {error.message}</div>;
        } else if (!isLoading) {
            return <div>Chargement...</div>;
        } else {
            return (
                <ul>
                    {items.map(item => (
                        <li key={item.name}>
                            {item.name} {item.price}
                        </li>
                    ))}
                </ul>
            );
        }
    }
}

```

Voici l'équivalent avec les Hooks :

```

function MyComponent() {
    const [error, setError] = useState(null);
    const [isLoading, setIsLoaded] = useState(false);
    const [items, setItems] = useState([]);

    // Remarque : le tableau vide de dépendances [] indique
    // que useEffect ne s'exécutera qu'une fois, un peu comme
    // componentDidMount()
    useEffect(() => {
        fetch("https://api.example.com/items")
            .then(res => res.json())
            .then(
                (result) => {
                    setIsLoaded(true);
                    setItems(result.items);
                },
                // Remarque : il faut gérer les erreurs ici plutôt que dans
                // un bloc catch() afin que nous n'avalions pas les exceptions
                // dues à de véritables bugs dans les composants.
                (error) => {
                    setError(error);
                }
            )
    }, [])

    if (error) {
        return <div>Erreur : {error.message}</div>;
    } else if (!isLoading) {
        return <div>Chargement...</div>;
    } else {
        return (
            <ul>
                {items.map(item => (
                    <li key={item.name}>

```

```
{item.name} {item.price}  
    </li>  
  )});  
  </ul>  
);  
}  
}
```

Babel, JSX, et étapes de construction

Ai-je besoin d'utiliser JSX avec React ?

Non ! Pour en apprendre davantage, consultez « [React sans JSX](#) ».

Ai-je besoin d'utiliser ES6 (+) avec React ?

Non ! Pour en apprendre davantage, consultez « [React sans ES6](#) ».

Comment écrire des commentaires en JSX ?

```
<div>
  {/* Placez un commentaire ici */}
  Bonjour, {name} !
</div>
```

```
<div>
  {/* Ça fonctionne aussi pour
     les commentaires sur plusieurs lignes */}
  Bonjour, {name} !
</div>
```

Passer des fonctions aux composants

Comment passer un gestionnaire d'événements (par exemple `onClick`) à un composant ?

On peut passer un gestionnaire d'événements et d'autres fonctions dans les props d'un composant, comme n'importe quelle autre valeur :

```
<button onClick={this.handleClick}>
```

Si vous avez besoin d'accéder au composant parent dans le gestionnaire d'événements, vous devrez lier la fonction à l'instance du composant (comme ci-dessous).

Comment lier une fonction à l'instance d'un composant ?

Il y'a plusieurs façons de s'assurer que des fonctions ont accès aux attributs du composant comme `this.-props` et `this.state`, qui dépendent de la syntaxe et de l'outillage éventuel que vous utilisez.

Lier la fonction dans le constructeur (ES2015)

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Cliqué');
  }
  render() {
    return <button onClick={this.handleClick}>Cliquez ici</button>;
  }
}
```

Propriété de classe (proposition stade 3)

```
class Foo extends Component {
  // Remarque : cette syntaxe est expérimentale et n'est pas encore standardisée
  handleClick = () => {
    console.log('Cliqué');
  }
  render() {
    return <button onClick={this.handleClick}>Cliquez ici</button>;
  }
}
```

Lier dans la méthode `render` `{#bind-in-render}`

```
class Foo extends Component {
  handleClick() {
    console.log('Cliqué');
  }
  render() {
```

```

    return <button onClick={this.handleClick.bind(this)}>Cliquez ici</button>;
}

```

Remarque

Utiliser `Function.prototype.bind` dans la méthode `render` crée une nouvelle fonction à chaque fois que le composant est affiché, ce qui peut impacter négativement les performances (voir plus bas).

Fonction fléchée dans le rendu

```

class Foo extends Component {
  handleClick() {
    console.log('Cliqué');
  }
  render() {
    return <button onClick={() => this.handleClick()}>Cliquez ici</button>;
  }
}

```

Remarque

Utiliser une fonction fléchée dans la fonction de rendu crée une nouvelle fonction à chaque fois que le composant est affiché, ce qui peut impacter négativement les optimisations basées sur une comparaison stricte d'identité.

Est-il acceptable d'utiliser une fonction fléchée à l'intérieur de la méthode `render` ?

C'est généralement acceptable, et c'est souvent la façon la plus facile de passer des arguments à une fonction de rappel.

Si vous avez des problèmes de performances, n'hésitez pas à optimiser !

Pourquoi est-il parfois nécessaire de lier une fonction ?

En JavaScript, ces deux extraits de code ne sont **pas** équivalents :

```
obj.method();
```

```

var method = obj.method;
method();

```

Lier les méthodes permet de s'assurer que le deuxième extrait de code fonctionne de la même manière que le premier.

Avec React, vous n'avez généralement besoin de lier que les méthodes que vous passez à d'autres composants. Par exemple, `<button onClick={this.handleClick}>` passe `this.handleClick`, vous devez donc la lier. Cependant, il n'est pas nécessaire de lier la méthode `render` ou les méthodes de cycle de vie : on ne les passe pas à d'autres composants.

Cet article de Yehuda Katz (en anglais) explique en détail ce qu'est la liaison de méthode et comment les fonctions marchent en JavaScript.

Pourquoi ma fonction est appelée à chaque affichage du composant ?

Vérifiez que vous *nappelez pas la fonction* en la passant au composant :

```
render() {
  // Erroné : handleClick est appelée au lieu d'être passée par référence !
  return <button onClick={this.handleClick()}>Cliquez ici</button>
}
```

Au lieu de ça, *passez la fonction* (sans les parenthèses) :

```
render() {
  // Correct : handleClick est passée par référence !
  return <button onClick={this.handleClick}>Cliquez ici</button>
}
```

Comment passer un argument à un gestionnaire d'événements ou une fonction de rappel ?

Vous pouvez utiliser une fonction fléchée pour enrober un gestionnaire d'événements et lui passer des arguments :

```
<button onClick={() => this.handleClick(id)} />
```

Le code ci-dessous est équivalent à une utilisation de `.bind` :

```
<button onClick={this.handleClick.bind(this, id)} />
```

Exemple : passer des arguments en utilisant une fonction fléchée

```
const A = 65 // Code de caractère ASCII

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }
  handleClick(letter) {
    this.setState({ justClicked: letter });
  }
  render() {
    return (
      <ul>
        {this.state.letters.map((letter) => (
          <li key={letter}>
            {letter}
            {letter === this.state.justClicked ? " (clicqué)" : null}
          </li>
        ))}
      </ul>
    );
  }
}
```

```

<div>
  Cliqué : {this.state.justClicked}
  <ul>
    {this.state.letters.map(letter =>
      <li key={letter} onClick={() => this.handleClick(letter)}>
        {letter}
      </li>
    )}
  </ul>
</div>
)
}
}

```

Exemple : passer des arguments en utilisant des attributs `data-*` [\[#example-passing-params-using-data-attributes}](#)

Une autre approche consiste à utiliser des API DOM pour stocker les données nécessaires aux gestionnaires d'événements. Pensez-y si vous avez besoin d'optimiser un grand nombre d'éléments ou si vous avez des composants basés sur `React.PureComponent`.

```

const A = 65 // Code de caractère ASCII

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }

  handleClick(e) {
    this.setState({
      justClicked: e.target.dataset.letter
    });
  }

  render() {
    return (
      <div>
        Cliqué : {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} data-letter={letter} onClick={this.handleClick}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}

```

Comment éviter qu'une fonction soit appelée trop tôt ou trop souvent ?

Si vous avez un gestionnaire d'événements comme `onClick` ou `onScroll` et que vous voulez éviter que la fonction de rappel soit appelée trop fréquemment, vous pouvez limiter sa fréquence d'exécution. Vous pouvez le faire en utilisant :

- **le *throttling*** : limitation de la fréquence de déclenchement (exemple : `_.throttle`)
- **le *debouncing*** : application des modifications après une période d'inactivité (exemple : `_.debounce`)
- **le *throttling basé sur requestAnimationFrame*** : limitation de la fréquence de déclenchement grâce à `requestAnimationFrame` (exemple : `raf-schd`)

Jetez un coup d'œil à [cette visualisation](#) pour une comparaison intuitive des fonctions `throttle` et `debounce`.

Remarque

`_.debounce`, `_.throttle` et `raf-schd` fournissent une méthode `cancel` pour annuler l'appel à la fonction de rappel différée. Il est recommandé d'appeler cette méthode dans `componentWillUnmount` ou de s'assurer dans la fonction différée que le composant est toujours monté.

Throttle

Le *throttling* évite qu'une fonction soit appelée plus d'une fois dans un certain laps de temps. L'exemple ci-dessous limite le gestionnaire d'événements de clic pour éviter qu'il soit appelé plus d'une fois par seconde.

```
import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleClickThrottled = throttle(this.handleClick, 1000);
  }

  componentWillUnmount() {
    this.handleClickThrottled.cancel();
  }

  render() {
    return <button onClick={this.handleClickThrottled}>Charger la suite</button>;
  }
}

handleClick() {
  this.props.loadMore();
}
```

Debounce

Le *debouncing* garantit qu'une fonction ne sera appelée qu'après qu'un certain temps a passé depuis le dernier appel à cette fonction : on garantit un intervalle minimal entre deux exécutions. Ça peut être utile quand vous avez un calcul complexe à faire en réponse à un événement susceptible d'être déclenché fréquemment (comme le défilement d'une page ou les frappes au clavier). L'exemple ci-dessous utilise cette méthode sur un champ texte avec un délai de 250 ms.

```
import debounce from 'lodash.debounce';

class Searchbox extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.emitChangeDebounced = debounce(this.emitChange, 250);
  }

  componentWillUnmount() {
    this.emitChangeDebounced.cancel();
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="Recherche..."
        defaultValue={this.props.value}
      />
    );
  }

  handleChange(e) {
    // React recycle les événements, on a donc besoin de lire la valeur avant le différé.
    // On aurait aussi pu appeler `event.persist()` et passer l'événement complet.
    // Pour en apprendre davantage, consultez
    // fr.reactjs.org/docs/events.html#event-pooling
    this.emitChangeDebounced(e.target.value);
  }

  emitChange(value) {
    this.props.onChange(value);
  }
}
```

Throttling basé sur `requestAnimationFrame` **{#requestAnimationFrame-throttling}**

`requestAnimationFrame` permet de différer une fonction pour qu'elle soit exécutée par le navigateur à un moment optimal pour les performances d'affichage. Une fonction différée avec `requestAnimationFrame` sera exécutée à la prochaine passe d'affichage (*frame, NdT*). Le navigateur fera de son mieux pour qu'il y ait toujours 60 passes par seconde (60 fps). Cependant, si le navigateur n'y arrive pas, il *limitera* naturellement le nombre de passes par seconde. Par exemple, un appareil pourrait n'être capable d'afficher que 30 fps, vous n'obtiendrez donc que 30 passes par seconde. Utiliser `requestAnimationFrame` pour limiter

l'exécution est une technique pratique afin d'éviter de faire plus de 60 mises à jour par seconde. Si vous faites 100 mises à jour en une seconde, vous créez une charge de travail supplémentaire pour le navigateur que l'utilisateur ne pourra de toutes façons pas voir.

Remarque

Utiliser cette technique ne capturera que la dernière valeur publiée à chaque passe. Vous pouvez voir un exemple du fonctionnement de cette optimisation sur le [MDN](#).

```
import rafSchedule from 'raf-schd';

class ScrollListener extends React.Component {
  constructor(props) {
    super(props);

    this.handleScroll = this.handleScroll.bind(this);

    // Crée une nouvelle fonction à planifier
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  handleScroll(e) {
    // Quand on reçoit un événement de défilement de la page, planifier une mise à jour.
    // Si on reçoit plusieurs mises à jour dans la même passe, on ne publiera que la
    // dernière valeur.
    this.scheduleUpdate({ x: e.clientX, y: e.clientY });
  }

  componentWillUnmount() {
    // Annule toute mise à jour en attente puisqu'on démonte le composant
    this.scheduleUpdate.cancel();
  }

  render() {
    return (
      <div
        style={{ overflow: 'scroll' }}
        onScroll={this.handleScroll}
      >
        
      </div>
    );
  }
}
```

Tester votre limitation de fréquence

Afin de tester que votre code de limitation de fréquence fonctionne correctement, il est utile de pouvoir manipuler le temps. Si vous utilisez `jest`, vous pouvez utiliser les [fausses horloges](#) (*mock timers*, *NdT*) pour accélérer le temps. Si vous utilisez `requestAnimationFrame`, vous pouvez utiliser `raf-stub` afin de contrôler la progression des passes d'animation.

react

[Go to TOC](#)

DOM virtuel et autres détails

Qu'est-ce que le DOM virtuel ?

Le DOM virtuel (VDOM) est un concept de programmation dans lequel une représentation idéale, ou « virtuelle », d'une interface utilisateur (UI) est conservée en mémoire et synchronisée avec le DOM « réel » par une bibliothèque telle que ReactDOM. Ce processus s'appelle [réconciliation](#).

Cette approche rend possible l'API déclarative de React : vous indiquez à React dans quel état vous souhaitez que l'UI se trouve, et il s'assure que le DOM correspond à cet état. Ça permet de faire abstraction de la manipulation des attributs, de la gestion des événements et de la mise à jour manuelle du DOM que vous auriez normalement dû faire vous-mêmes pour créer votre application.

Puisque le « DOM virtuel » est plus un modèle qu'une technologie spécifique, on l'emploie parfois pour désigner différentes choses. Dans le monde React, le terme « DOM virtuel » est généralement associé aux [éléments React](#), car il s'agit des objets représentant l'interface utilisateur. Cependant, React utilise également des objets internes appelés « fibres » (*fibers, NdT*) pour conserver des informations supplémentaires sur l'arbre des composants. Ils peuvent également être considérés comme faisant partie de l'implémentation du « DOM virtuel » dans React.

Est-ce que le Shadow DOM est identique au DOM virtuel ?

Non, ils sont différents. Le Shadow DOM est une technologie du navigateur conçue principalement pour limiter la portée des variables et du CSS dans les Web Components. Le DOM virtuel est un concept implémenté par les bibliothèques en JavaScript en plus des API des navigateurs.

Qu'est-ce que « React Fiber » ?

Fiber est le nouveau moteur de réconciliation de React 16. Son objectif principal est de permettre le rendu incrémentiel du DOM virtuel. [En savoir plus](#).

État local de composant

Que fait `setState` ?

`setState()` planifie la mise à jour de l'objet `state` du composant. Quand l'état local change, le composant répond en se rafraîchissant.

Quelle est la différence entre `state` et `props` ?

`props` (diminutif de « propriétés ») et `state` sont tous les deux des objets JavaScript bruts. Même s'ils contiennent tous les deux des informations qui influencent le résultat produit, ils présentent une différence majeure : `props` est passé *au* composant (à la manière des arguments d'une fonction) tandis que `state` est géré *dans* le composant (comme le sont les variables déclarées à l'intérieur d'une fonction).

Voici quelques ressources utiles pour mieux comprendre selon quels critères choisir entre `props` et `state` :

- [Props vs State](#)
- [ReactJS: Props vs. State](#)

Pourquoi `setState` me renvoie-t-elle une valeur incorrecte ?

En React, `this.props` et `this.state` représentent l'un comme l'autre les valeurs du rendu, c'est-à-dire ce qui est actuellement affiché.

Les appels à `setState` sont asynchrones : ne comptez pas sur `this.state` pour refléter la nouvelle valeur immédiatement après avoir appelé `setState`. Passez une fonction de mise à jour à la place d'un objet si vous devez calculer des valeurs en fonction de l'état actuel (voir ci-dessous pour plus de détails).

Voici un exemple de code qui ne se comporte *pas* comme attendu :

```
incrementCount() {
  // Attention : ça ne va *pas* fonctionner comme prévu.
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // Disons que `this.state.count` commence à 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // Lorsque React rafraîchira le composant, `this.state.count` sera à 1,
  // pourtant, on s'attendait à 3.

  // C'est parce que la fonction `incrementCount()` ci-dessus lit
  `this.state.count`,
  // mais React ne met pas à jour `this.state.count` tant que le composant n'est
  // pas rafraîchi.
  // Du coup `incrementCount()` lit `this.state.count` qui est égal à 0 à chaque
  // fois,
  // et le définit à 1.
```

```
// Le correctif est décrit ci-dessous !
}
```

Voir ci-dessous pour savoir comment résoudre ce problème.

Comment mettre à jour l'état avec des valeurs qui dépendent de l'état actuel ?

Passez une fonction au lieu d'un objet à `setState` pour vous assurer que l'appel utilise toujours la version la plus récente de l'état (voir ci-dessous).

Quelle est la différence entre passer un objet ou une fonction à `setState` ?

Passer une fonction de mise à jour vous permet d'accéder à la valeur à jour de l'état actuel au sein de cette fonction. Comme les appels `setState` sont groupés par lots, ça vous permet d'enchaîner les mises à jour et de vous assurer qu'elles sont effectuées les unes après les autres au lieu d'entrer en conflit :

```
incrementCount() {
  this.setState((state) => {
    // Important : lisez `state` au lieu de `this.state` lors de la mise à jour.
    return {count: state.count + 1}
  });
}

handleSomething() {
  // Disons que `this.state.count` commence à 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();

  // Si vous lisiez `this.state.count` maintenant, il serait toujours à 0.
  // Mais quand React rafraîchira le composant, il vaudra bien 3.
}
```

[En apprendre davantage sur `setState`](#)

Quand `setState` est-elle asynchrone ?

Actuellement, `setState` est asynchrone à l'intérieur des gestionnaires d'événements.

Ça permet de garantir, par exemple, que si `Parent` et `Child` appellent tous les deux `setState` lors d'un clic, `Child` ne sera pas rafraîchi deux fois. Au lieu de ça, React « apure » les mises à jour de l'état à la fin de l'événement du navigateur. Ça permet une amélioration significative des performances pour les applications de grande ampleur.

Il s'agit d'un détail d'implémentation donc évitez de vous appuyer dessus. Dans de futures versions, React groupera par défaut les mises à jour dans davantage encore de cas de figure.

Pourquoi React ne met-il pas à jour `this.state` de façon synchrone ?

Comme expliqué dans la section précédente, React « attend » volontairement que tous les composants aient fini d'appeler `setState()` dans leurs gestionnaires d'événements avant de commencer à mettre à jour les rendus. Ça améliore les performances en évitant des rafraîchissements inutiles.

Cependant, vous vous demandez peut-être toujours pourquoi React ne met pas juste à jour `this.state` immédiatement sans rafraîchir.

Il y a deux raisons principales :

- Ça briserait la cohérence entre `props` et `state`, entraînant des problèmes très difficiles à déboguer.
- Ça rendrait certaines nouvelles fonctionnalités sur lesquelles nous travaillons impossibles à implémenter.

Ce [commentaire GitHub](#) entre dans le détail d'exemples spécifiques.

Devrais-je utiliser une bibliothèque de gestion d'état, comme par exemple Redux ou Mobx ?

[Peut-être.](#)

Bien connaître React avant d'ajouter des bibliothèques supplémentaires reste une bonne idée. Vous pouvez créer des applications plutôt complexes en utilisant uniquement React.

[Go to TOC](#)

Structure de fichiers

Y-a-t-il une manière à privilégier pour structurer les projets React ?

React n'a pas d'opinion sur la manière dont vous ordonnez vos fichiers à l'intérieur de vos dossiers. Ceci dit, vous souhaiterez peut-être envisager l'une des approches populaires de l'écosystème.

Grouper par fonctionnalité ou par route

Une façon courante de structurer les projets consiste à placer le CSS, le JS et les tests ensemble dans des dossiers groupés par fonctionnalité ou par route.

```
common/
  Avatar.js
  Avatar.css
  APIUtils.js
  APIUtils.test.js
feed/
  index.js
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  FeedAPI.js
profile/
  index.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
  ProfileAPI.js
```

La définition d'une « fonctionnalité » n'est pas universelle, et il vous appartient d'en définir la granularité. Si vous ne parvenez pas à créer une liste des dossiers racines, vous pouvez demander aux utilisateurs de votre produit quelles en sont les principales composantes et utiliser leur modèle mental comme base.

Grouper par type de fichier

Une autre manière répandue de structurer les projets consiste à grouper les fichiers similaires ensemble, par exemple :

```
api/
  APIUtils.js
  APIUtils.test.js
  ProfileAPI.js
  UserAPI.js
components/
  Avatar.js
  Avatar.css
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
```

Profile.js
ProfileHeader.js
ProfileHeader.css

Certaines personnes préfèrent également aller plus loin et séparer les composants dans des dossiers différents en fonction de leur rôle dans l'application. Par exemple, la [conception atomique](#) est une méthode de conception reposant sur ce principe. N'oubliez pas qu'il est souvent plus productif de traiter de telles méthodologies comme des exemples utiles plutôt que comme des règles strictes à suivre.

Évitez trop d'imbrication

L'imbrication excessive de répertoires dans les projets JavaScript est source de nombreuses souffrances. Il devient plus difficile d'écrire des importations relatives entre eux ou de mettre à jour ces importations lorsque les fichiers sont déplacés. À moins d'avoir une excellente raison d'utiliser une structure de dossiers profonde, limitez-vous à un maximum de trois ou quatre imbrications de dossier dans un même projet. Bien entendu, il ne s'agit que d'une recommandation, qui n'est peut-être pas pertinente pour votre projet.

Ne vous prenez pas trop la tête

Si vous démarrez tout juste un projet, [ne passez pas plus de cinq minutes](#) à choisir une structure de fichiers. Choisissez l'une des approches ci-dessus (ou prenez la vôtre) et commencez à écrire votre code ! Vous serez sûrement amené·e à la repenser de toutes façons une fois que vous aurez produit du vrai code.

Si vous vous sentez complètement bloqué·e, commencez par garder tous les fichiers dans un seul dossier. Si votre application se met à grossir, vous aurez envie de séparer certains fichiers du reste. À ce moment-là, vous aurez une bonne idée des fichiers que vous éditez ensemble régulièrement. En général, il est judicieux de conserver en un même endroit des fichiers qui changent souvent ensemble. Ce principe s'appelle « colocation ».

À mesure que les projets prennent de l'ampleur, on utilise souvent un mélange des deux approches ci-dessus. Du coup, choisir la « bonne » au début n'est pas si important.

Styles et CSS

Comment ajouter des classes CSS aux composants ?

En passant une chaîne à la prop `className` :

```
render() {
  return <span className="menu navigation-menu">Menu</span>
}
```

Les classes CSS dépendent fréquemment des props ou de l'état local du composant :

```
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += ' menu-active';
  }
  return <span className={className}>Menu</span>
}
```

Astuce

S'il vous arrive régulièrement d'écrire du code comme celui-ci, le module `classnames` peut vous aider à simplifier votre code.

Puis-je utiliser des styles en ligne ?

Oui, consultez la documentation sur le style [ici](#).

Est-ce que les styles en ligne sont une mauvaise pratique ?

Les classes CSS sont généralement plus performantes que les styles en ligne.

Qu'est-ce que CSS-in-JS ?

"CSS-in-JS" fait référence à un modèle où les styles CSS sont créés en utilisant du JavaScript au lieu d'être définis dans des fichiers externes.

Remarquez bien que cette fonctionnalité ne fait pas partie de React : elle est fournie par des bibliothèques tierces. React n'a pas d'opinion sur la manière dont les styles doivent être définis ; si vous avez un doute, une bonne manière de commencer consiste à définir vos styles dans des fichiers `*.css` séparés comme d'habitude et à y faire référence en utilisant `className`.

Puis-je faire des animations avec React ?

React peut être utilisé pour générer des animations. Voyez par exemple [React Transition Group](#) et [React Spring](#).

react

[Go to TOC](#)

Politique de gestion des versions

React suit les principes de [gestion sémantique de version \(semver\)](#).

Ça signifie qu'avec un numéro de version de type **x.y.z** :

- Pour publier des **corrections de bugs critiques**, nous changeons de **version de correctif** en modifiant le nombre **z** (ex. 15.6.2 à 15.6.3).
- Pour publier des **nouvelles fonctionnalités** ou des **corrections de bugs non-critiques**, nous changeons de **version mineure** en modifiant le nombre **y** (ex. 15.6.2 à 15.7.0).
- Pour publier des **modifications cassant la compatibilité ascendante**, nous changeons de **version majeure** en modifiant le nombre **x** (ex. 15.6.2 à 16.0.0).

Les versions majeures peuvent également contenir de nouvelles fonctionnalités, et toute version peut inclure des corrections de bugs.

Les versions mineures sont de loin les plus courantes.

Ruptures de compatibilité ascendante

Cette politique de versions ne s'applique pas aux pré-versions dans les canaux *Next* et *Experimental*.

[En savoir plus sur les pré-versions](#).

Personne n'aime perdre en compatibilité ascendante, nous essayons donc de minimiser le nombre de versions majeures ; par exemple, React 15 est sorti en avril 2016 et React 16 en septembre 2017 ; React 17 n'est pas prévu avant 2020.

Au lieu de ça, nous publions les nouvelles fonctionnalités dans des versions mineures. Celles-ci sont souvent plus intéressantes et motivantes que les majeures, malgré leur nom modeste.

Nos engagements en termes de stabilité

À mesure que nous améliorons React, nous essayons d'abaisser la barrière d'entrée pour tirer parti des nouvelles fonctionnalités. Chaque fois que possible, nous continuons à prendre en charge une vieille API, même si ça implique de la placer dans un module séparé. Par exemple, [les mixins sont découragés depuis des années](#) mais ils restent pris en charge à ce jour [via create-react-class](#) et de nombreuses bases de code continuent de les utiliser dans du code historique stable.

Plus d'un million de développeurs utilisent React, qui maintiennent collectivement des millions de composants. La base de code de Facebook contient à elle seule plus de 50 000 composants React. Ça signifie que nous devons faciliter au maximum la mise à niveau vers les nouvelles versions de React ; si nous apportons des modifications importantes sans fournir d'aide à la migration, les utilisateurs resteront bloqués sur les anciennes versions. Nous testons ces approches de mise à niveau sur Facebook même : si notre équipe de moins de 10 personnes peut mettre à jour plus de 50 000 composants à elle seule, nous espérons que la

mise à niveau sera gérable pour toute personne utilisant React. Dans de nombreux cas, nous écrivons des [scripts automatisés](#) pour mettre à niveau la syntaxe des composants, que nous incluons ensuite dans la version libre de notre code source pour que chacun·e puisse les utiliser.

Mises à niveau progressives grâce aux avertissements

Les versions de développement de React incluent de nombreux avertissements utiles. Dans la mesure du possible, nous ajoutons des avertissements en prévision de futurs changements radicaux. Ainsi, si votre application ne contient aucun avertissement sur la dernière version, elle sera compatible avec la prochaine version majeure. Ça vous permet de mettre à niveau vos applications composant par composant.

Les avertissements de développement n'affecteront pas le comportement d'exécution de votre application. De cette façon, vous pouvez être sûr·e que votre application se comportera de la même façon entre les versions de développement et de production. La seule différence, c'est que la version de production n'émettra pas les avertissements et qu'elle sera plus efficace. (Si vous remarquez le contraire, veuillez créer une *issue* dans le dépôt GitHub.)

Qu'est ce qui constitue une rupture de compatibilité ?

En général, nous *n'élevons pas* le numéro de version majeure pour les modifications apportées aux aspects suivants :

- **Avertissements de développement.** Dans la mesure où ils n'affectent pas le comportement de production, nous nous réservons le droit d'ajouter de nouveaux avertissements ou de modifier les avertissements existants entre les versions majeures. En fait, c'est ce qui nous permet de vous avertir de manière fiable des changements à venir.
- **API commençant par `unstable_`.** Celles-ci sont fournies en tant que fonctionnalités expérimentales et nous ne sommes pas encore satisfaits de leurs API. En les publiant avec un préfixe `unstable_`, nous pouvons itérer plus rapidement dessus et obtenir une API stable plus tôt.
- **Versions alpha et canary de React.** Nous fournissons des versions alpha de React afin de tester les nouvelles fonctionnalités à un stade précoce, mais nous avons besoin de la souplesse nécessaire pour apporter des modifications en fonction de ce que nous apprenons au cours de la période alpha. Si vous utilisez ces versions, notez que les API peuvent changer avant la version stable.
- **API non documentées et structures de données internes.** Si vous accédez à des noms de propriété internes tels que `__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED` ou `__reactInternalInstance$uk43rzhitjg`, il n'y a aucune garantie. Débrouillez-vous.

Cette politique se veut pragmatique : nous ne voulons évidemment pas vous causer de maux de tête. Si nous élevions la version majeure pour tous ces changements, nous finirions par publier plus de versions majeures, ce qui s'avèrera plus pénible pour la communauté. Ça signifierait également que nous ne pourrions pas améliorer React aussi rapidement que nous le souhaiterions.

Cela dit, si nous nous attendons à ce qu'un changement sur cette liste cause de gros problèmes dans la communauté, nous ferons tout notre possible pour fournir un chemin de migration progressif.

Si une version mineure n'ajoute pas de fonctionnalités, pourquoi n'est-ce pas un correctif ?

Il est possible qu'une version mineure n'inclue pas de nouvelles fonctionnalités. [Semver autorise ça](#), en disant qu'**« [une version mineure] PEUT être incrémentée lors de nouvelles fonctionnalités substantielles ou d'améliorations introduites dans le code privé. Elle PEUT inclure des modifications de type correctif. »**

Toutefois, cela soulève la question de pourquoi ces versions ne sont pas signalées comme de simples correctifs.

Cela tient au fait que toute modification à React (ou à d'autres logiciels) emporte un risque de dysfonctionnements inattendus. Imaginez un scénario dans lequel une version de correctif qui corrige un bug en introduit accidentellement un autre. Ce ne serait pas seulement dérangeant pour les développeur·se·s, mais ça nuirait à la confiance dans les versions de correctif ultérieures. C'est particulièrement regrettable si le correctif d'origine visait un bug survenant rarement en pratique.

Nous avons un historique plutôt honorable de versions de React sans bugs, mais les versions de correctif mettent la barre encore plus haut en termes de fiabilité parce que la plupart des gens supposent qu'ils peuvent les adopter sans risque.

Voilà pourquoi nous réservons les versions de correctifs aux bugs les plus critiques et aux failles de sécurité.

Si une version inclut des modifications qui ne sont pas essentielles—telles que des refactorings internes, des modifications d'implémentation, des améliorations de performances ou des correctifs mineurs—nous incrémenterons la version mineure quand bien même aucune nouvelle fonctionnalité n'est présente.

Formulaires

Les formulaires HTML fonctionnent un peu différemment des autres éléments du DOM en React car ils possèdent naturellement un état interne. Par exemple, ce formulaire en HTML qui accepte juste un nom :

```
<form>
  <label>
    Nom :
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Envoyer" />
</form>
```

Ce formulaire a le comportement classique d'un formulaire HTML et redirige sur une nouvelle page quand l'utilisateur le soumet. Si vous souhaitez ce comportement en React, vous n'avez rien à faire. Cependant, dans la plupart des cas, vous voudrez pouvoir gérer la soumission avec une fonction JavaScript, qui accède aux données saisies par l'utilisateur. La manière classique de faire ça consiste à utiliser les « composants contrôlés ».

Composants contrôlés

En HTML, les éléments de formulaire tels que `<input>`, `<textarea>`, et `<select>` maintiennent généralement leur propre état et se mettent à jour par rapport aux saisies de l'utilisateur. En React, l'état modifiable est généralement stocké dans la propriété `state` des composants et mis à jour uniquement avec `setState()`.

On peut combiner ces deux concepts en utilisant l'état local React comme « source unique de vérité ». Ainsi le composant React qui affiche le formulaire contrôle aussi son comportement par rapport aux saisies de l'utilisateur. Un champ de formulaire dont l'état est contrôlé de cette façon par React est appelé un « composant contrôlé ».

Par exemple, en reprenant le code ci-dessus pour afficher le nom lors de la soumission, on peut écrire le formulaire sous forme de 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});
  }

  handleSubmit(event) {
    alert('Le nom a été soumis : ' + this.state.value);
    event.preventDefault();
  }
}
```

```

render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Nom :
        <input type="text" value={this.state.value} onChange={this.handleChange}>
      />
      </label>
      <input type="submit" value="Envoyer" />
    </form>
  );
}
}

```

Essayer sur CodePen

À présent que l'attribut `value` est défini sur notre élément de formulaire, la valeur affichée sera toujours `this.state.value`, faisant ainsi de l'état local React la source de vérité. Puisque `handleChange` est déclenché à chaque frappe pour mettre à jour l'état local React, la valeur affichée restera mise à jour au fil de la saisie.

Dans un composant contrôlé, la valeur du champ est en permanence pilotée par l'état React. Même si ça signifie que vous devez taper un peu plus de code, vous pouvez désormais passer la valeur à d'autres éléments de l'UI, ou la réinitialiser depuis d'autres gestionnaires d'événements.

La balise `textarea` {#the-textarea-tag}

En HTML, une balise `<textarea>` définit son texte via ses enfants :

```

<textarea>
  Bonjour, voici du texte dans une zone de texte
</textarea>

```

En React, un `<textarea>` utilise à la place l'attribut `value`. Du coup, un formulaire utilisant un `<textarea>` peut être écrit d'une manière très similaire à un formulaire avec un élément `<input>` monoligne.

```

class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Écrivez un essai à propos de votre élément du DOM préféré'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('Un essai a été envoyé : ' + this.state.value);
    event.preventDefault();
  }
}

```

```

    }

    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          <label>
            Essay:
            <textarea value={this.state.value} onChange={this.handleChange} />
          </label>
          <input type="submit" value="Envoyer" />
        </form>
      );
    }
  }
}

```

Remarquez que `this.state.value` est initialisé dans le constructeur, permettant que le textarea démarre avec du texte à l'intérieur.

La balise `select` {#the-select-tag}

En HTML, `<select>` crée une liste déroulante. Par exemple, ce HTML crée une liste déroulante de parfums.

```

<select>
  <option value="grapefruit">Pamplemousse</option>
  <option value="lime">Citron vert</option>
  <option selected value="coconut">Noix de coco</option>
  <option value="mango">Mangue</option>
</select>

```

Notez que l'option Noix de coco est sélectionnée au départ, grâce à l'attribut `selected`. React, au lieu d'utiliser l'attribut `selected`, utilise un attribut `value` sur la balise racine `select`. C'est plus pratique dans un composant contrôlé car vous n'avez qu'un seul endroit à mettre à jour. Par exemple :

```

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('Votre parfum favori est : ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Choisissez votre parfum favori :
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Pamplemousse</option>

```

```

        <option value="lime">Citron vert</option>
        <option value="coconut">Noix de coco</option>
        <option value="mango">Mangue</option>
    </select>
</label>
<input type="submit" value="Envoyer" />
</form>
);
}
}

```

[Essayer sur CodePen](#)

Au final, ça permet aux balises `<input type="text">`, `<textarea>`, et `<select>` de fonctionner de manière très similaire—elles acceptent toutes un attribut `value` que vous pouvez utiliser pour implémenter un composant contrôlé.

Note

Vous pouvez passer un tableau pour l'attribut `value`, permettant de sélectionner plusieurs valeurs dans un élément `select` :

```
<select multiple={true} value={['B', 'C']}>
```

La balise `input type="file"` [\[#the-file-input-tag\]](#)

En HTML, un `<input type="file">` permet à l'utilisateur de sélectionner un ou plusieurs fichiers depuis son appareil et de les téléverser vers un serveur ou de les manipuler en JavaScript grâce à l'[API File](#).

```
<input type="file" />
```

Sa valeur étant en lecture seule, c'est un composant **non-contrôlé** en React. Ce cas de figure et le sujet des composants non-contrôlés en général sera détaillé [plus tard dans la documentation](#).

Gérer plusieurs saisies

Quand vous souhaitez gérer plusieurs champs contrôlés, vous pouvez ajouter un attribut `name` à chaque champ et laisser la fonction gestionnaire choisir quoi faire en fonction de la valeur de `event.target.name`.

Par exemple :

```

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>
        Participe :
        <input
          name="isGoing"
          type="checkbox"
          checked={this.state.isGoing}
          onChange={this.handleInputChange} />
      </label>
      <br />
      <label>
        Nombre d'invités :
        <input
          name="numberOfGuests"
          type="number"
          value={this.state.numberOfGuests}
          onChange={this.handleInputChange} />
      </label>
    </form>
  );
}
}

```

[Essayer sur CodePen](#)

Notez l'utilisation de la syntaxe des [propriétés calculées](#) pour mettre à jour la valeur de l'état correspondant au nom du champ.

```

this.setState({
  [name]: value
});
```

C'est équivalent à ce code ES5 :

```

var partialState = {};
partialState[name] = value;
this.setState(partialState);
```

Qui plus est, comme `setState()` fusionne automatiquement un état partiel dans l'état local courant, il nous a suffi de l'appeler avec les parties modifiées.

Valeur nulle des champs contrôlés

Définir la prop `value` sur un composant contrôlé empêche l'utilisateur de changer la saisie sauf si vous le permettez. Si vous spécifiez une `value` mais que le champ reste modifiable, alors `value` doit s'être accidentellement retrouvée à `undefined` ou `null`.

Le code suivant illustre ce cas de figure. (Le champ est verrouillé au démarrage mais devient modifiable après un court délai.)

```
ReactDOM.render(<input value="Salut" />, mountNode);  
setTimeout(function() {  
  ReactDOM.render(<input value={null} />, mountNode);  
, 1000);
```

Alternatives aux composants contrôlés

Il est parfois fastidieux d'utiliser les composants contrôlés, car il vous faut écrire un gestionnaire d'événement pour chaque possibilité de changement des données, et gérer toute modification des saisies via un composant React. Ça peut devenir particulièrement irritant lors de la conversion d'un projet en React, ou l'intégration d'une application React avec une bibliothèque non-React. Dans ces situations, il est intéressant de connaître les [composants non-contrôlés](#), une technique alternative pour implémenter les formulaires de saisie.

Solutions clé en main

Si vous cherchez une solution complète gérant la validation, l'historique des champs visités, et la gestion de soumission de formulaire, [Formik](#) fait partie des choix populaires. Ceci dit, il repose sur les mêmes principes de composants contrôlés et de gestion d'état local—alors ne faites pas l'impasse dessus.

Transfert de refs

Le transfert de ref est une technique permettant de déléguer automatiquement une `ref` d'un composant à l'un de ses enfants. Ça n'est généralement pas nécessaire pour la plupart des composants dans une application. Cependant, ça peut être utile pour certains types de composants, particulièrement dans les bibliothèques de composants réutilisables. Les scénarios les plus fréquents sont décrits ci-dessous.

Transfert de refs vers des composants du DOM

Prenons un composant `FancyButton` qui affiche l'élément DOM natif `button` : `embed:forwarding-refs/fancy-button-simple.js`

Les composants React masquent leurs détails d'implémentation, y compris leur rendu. Les autres composants utilisant `FancyButton` **n'auront généralement pas besoin** d'obtenir une `ref` sur l'élément DOM interne `button`. C'est une bonne chose, car ça empêche les composants de trop s'appuyer sur la structure DOM les uns et des autres.

Bien qu'une telle encapsulation soit souhaitable pour les composants applicatifs tels que `FeedStory` ou `Comment`, elle peut être gênante pour les composants hautement réutilisables, tels que `FancyButton` ou `MyTextInput`. Ces composants ont tendance à être utilisés un peu partout dans l'application de manière similaire à un `button` ou un `input`, et l'accès à leurs nœuds DOM peut s'avérer nécessaire pour la gestion du focus, de la sélection ou des animations.

Le transfert de ref est une fonctionnalité optionnelle qui permet à certains composants de prendre une `ref` qu'ils reçoivent et de la passer plus bas dans l'arbre (en d'autres termes, la « transférer ») à un composant enfant.

Dans l'exemple ci-dessous, `FancyButton` utilise `React.forwardRef` pour obtenir la `ref` qui lui est passée, puis la transfère au `button` DOM qu'il affiche : `embed:forwarding-refs/fancy-button-simple-ref.js`

De cette façon, les composants utilisant `FancyButton` peuvent obtenir une `ref` sur le nœud DOM `button` sous-jacent et y accéder si nécessaire, comme s'ils utilisaient directement un `button` DOM.

Voici une explication étape par étape de ce qui se passe dans l'exemple ci-dessus :

1. Nous créons une `ref React` en appelant `React.createRef` et l'affectons à une variable `ref`.
2. Nous passons notre `ref` à `<FancyButton ref={ref}>` en la spécifiant comme un attribut JSX.
3. React transmet la `ref` à la fonction `(props, ref) => ...` à l'intérieur de `forwardRef` comme deuxième argument.
4. Nous transférons cet argument `ref` au `<button ref={ref}>` en le spécifiant comme un attribut JSX.
5. Quand la `ref` est liée, `ref.current` pointera vers le nœud DOM `button`.

Remarque

Le second argument `ref` n'existe que quand vous définissez un composant avec l'appel à `React.forwardRef`. Les fonctions composants habituelles et les composants à base de classes ne reçoivent pas l'argument `ref`, et la `ref` n'est pas non plus disponible dans les props du composant.

Le transfert de refs n'est pas limité aux composants DOM. Vous pouvez aussi transférer des refs vers des instances de classe de composant.

Note pour les mainteneurs de bibliothèques de composants

Lorsque vous commencez à utiliser `forwardRef` dans une bibliothèque de composants, vous devez le traiter comme une rupture de compatibilité ascendante et publier une nouvelle version majeure de votre bibliothèque. En effet, votre bibliothèque a probablement un comportement différent (par exemple la cible d'affectation des refs et la nature des types exportés), et ça pourrait casser les applications et autres bibliothèques qui dépendent de l'ancien comportement.

L'application conditionnelle de `React.forwardRef` lorsqu'elle existe est également déconseillée pour les mêmes raisons : ça modifie le comportement de votre bibliothèque et pourrait casser les applications de vos utilisateurs lorsqu'ils mettent à jour React.

Transfert des refs dans les composants d'ordre supérieur

Cette technique peut aussi être particulièrement utile avec les [composants d'ordre supérieur](#) (*Higher-Order Components ou HOC, NdT*). Commençons par un exemple de HOC qui journalise les props du composant dans la console : [embed:forwarding-refs/log-props-before.js](#)

Le HOC `logProps` transmet toutes les `props` au composant qu'il enrobe, ainsi le résultat affiché sera la même. Par exemple, nous pouvons utiliser ce HOC pour lister toutes les props transmises à notre composant *fancy button* : [embed:forwarding-refs/fancy-button.js](#)

Il y a une limitation dans l'exemple ci-dessus : les refs ne seront pas transférées. C'est parce que `ref` n'est pas une prop. Comme `key`, elle est gérée différemment par React. Si vous ajoutez une `ref` à un HOC, la `ref` fera référence au composant conteneur extérieur, et non au composant enrobé.

Ça signifie que les refs destinées à notre composant `FancyButton` seront en réalité attachées au composant `LogProps` : [embed:forwarding-refs/fancy-button-ref.js](#)

Heureusement, nous pouvons explicitement transférer les refs au composant `FancyButton` interne à l'aide de l'API `React.forwardRef`. Celle-ci accepte une fonction de rendu qui reçoit les arguments `props` et `ref` et renvoie un nœud React. Par exemple : [embed:forwarding-refs/log-props-after.js](#)

Affichage d'un nom personnalisé dans les DevTools

`React.forwardRef` accepte une fonction de rendu. Les outils de développement React (*React DevTools, NdT*) utilisent cette fonction pour déterminer quoi afficher pour le composant de transfert de ref.

Par exemple, le composant suivant apparaîtra sous le nom "ForwardRef" dans les DevTools : `embed:forwarding-refs/wrapped-component.js`

Si vous nommez la fonction de rendu, les DevTools incluront également son nom (par exemple, "`ForwardRef(myFunction)`") : `embed:forwarding-refs/wrapped-component-with-function-name.js`

Vous pouvez même définir la propriété `displayName` de la fonction pour y inclure le composant que vous enrobez : `embed:forwarding-refs/customized-display-name.js`

Fragments

En React, il est courant pour un composant de renvoyer plusieurs éléments. Les fragments nous permettent de grouper une liste d'enfants sans ajouter de nœud supplémentaire au DOM.

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

Il existe aussi une nouvelle [syntaxe concise](#) pour les déclarer.

Motivation

Il est courant qu'un composant renvoie une liste d'enfants. Prenez cette exemple de bout de code React :

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

`<Columns />` aurait besoin de renvoyer plusieurs éléments `<td>` pour que le HTML obtenu soit valide. Mais si une div parente était utilisée dans le `render()` de `<Columns />`, alors le HTML résultat serait invalide.

```
class Columns extends React.Component {
  render() {
    return (
      <div>
        <td>Bonjour</td>
        <td>Monde</td>
      </div>
    );
  }
}
```

Ce code donne en sortie le `<Table />` suivant :

```
<table>
  <tr>
    <div>
```

```
<td>Bonjour</td>
<td>Monde</td>
</div>
</tr>
</table>
```

Les fragments résolvent ce problème.

Usage

```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Bonjour</td>
        <td>Monde</td>
      </React.Fragment>
    );
  }
}
```

Ce qui donne en sortie le `<Table />` correct suivant :

```
<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>
```

Syntaxe concise

Il existe une nouvelle syntaxe plus concise, que vous pouvez utiliser pour déclarer des fragments. Ça ressemble à des balises vides :

```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Bonjour</td>
        <td>Monde</td>
      </>
    );
  }
}
```

Vous pouvez utiliser `<>...</>` de la même manière que n'importe quel élément, à ceci près que cette syntaxe n'accepte ni clés ni attributs.

Les fragments à clé

Les fragments déclarés explicitement avec la syntaxe `<React.Fragment>` peuvent avoir des clés (`key`, `NdT`). Un cas d'utilisation consisterait à faire correspondre une collection à un tableau de fragments—par exemple pour créer une liste de descriptions :

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Sans la `key`, React produira un avertissement spécifique
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

`key` est le seul attribut qui peut être passé à `Fragment`. À l'avenir, nous prendrons peut-être en charge des attributs supplémentaires, tels que des gestionnaires d'événements .

Démo interactive

Vous pouvez essayer la nouvelle syntaxe JSX de fragment avec ce [CodePen](#).

Bien démarrer

- "download.html"
- "downloads.html"

Cette page est un aperçu de la documentation React et des ressources associées.

React est une bibliothèque JavaScript pour la construction d'interfaces utilisateur (UI). Pour découvrir à quoi sert React, allez sur [notre page d'accueil](#) ou [dans le tutoriel](#).

- [Essayer React](#)
- [Apprendre React](#)
- [Se tenir au courant](#)
- [Documentation versionnée](#)
- [Quelque chose vous manque ?](#)

Essayer React

React a été conçu dès le départ pour une adoption progressive, et **vous pouvez utiliser React à minima ou autant que nécessaire**. Que vous souhaitiez avoir un aperçu de React, ajouter de l'interactivité à une simple page HTML ou démarrer une application React complète, les liens de cette section vous aideront à bien démarrer.

Terrains de jeu en ligne

Si vous souhaitez tester React, vous pouvez utiliser un terrain de jeu en ligne. Essayez un modèle *Hello World* sur [CodePen](#), [CodeSandbox](#), [Glitch](#) ou [Stackblitz](#).

Si vous préférez utiliser votre propre éditeur de texte, vous pouvez aussi [télécharger ce fichier HTML](#), l'édition et l'ouvrir à partir du système de fichiers local dans votre navigateur. Il transforme le code à la volée lors de l'exécution, ce qui est particulièrement lent. Pour cette raison, nous vous recommandons de ne l'utiliser que pour des démos simples.

Ajouter React à un site Web

Vous pouvez [ajouter React à une page HTML en une minute](#) et ensuite étendre progressivement sa présence ou la limiter à quelques éléments d'interface dynamiques.

Créer une nouvelle application React

Lorsque vous démarrez un projet React, une [simple page HTML avec des balises de script](#) reste peut-être la meilleure option. Ça ne prend qu'une minute à mettre en place !

Au fur et à mesure que votre application grandit, vous voudrez peut-être envisager une configuration plus intégrée. Il y a plusieurs boîtes à outils JavaScript que nous conseillons pour des applications plus importantes. Chacune d'entre elles peut fonctionner avec peu ou pas de configuration et vous permet de tirer pleinement parti du riche écosystème de React. [Découvrez comment faire.](#)

Apprendre React

Les gens qui découvrent React viennent d'horizons et de styles d'apprentissage variés. Que vous préfériez une approche plutôt théorique ou pratique, nous espérons que vous trouverez cette section utile.

- Si vous préférez **apprendre en faisant**, commencez par notre [tutoriel](#).
- Si vous préférez **apprendre les concepts étape par étape**, commencez par notre [guide des fondamentaux](#).

Comme toute technologie que vous ne connaissez pas encore, React a une certaine courbe d'apprentissage. Avec de la pratique et un peu de patience, vous *arriverez* à le maîtriser.

Premiers exemples

La [page d'accueil de React](#) contient quelques petits exemples React que vous pouvez modifier directement en ligne. Même si vous ne connaissez rien encore à React, essayez de changer le code de ces exemples et voyez comment cela affecte le résultat.

React pour les débutant·e·s

Si vous trouvez que la documentation React va un peu trop vite pour vous, consultez [cet aperçu de React par Tania Rascia](#). Il présente les concepts les plus importants de React de façon détaillée et conviviale pour les débutant·e·s. Une fois que vous avez terminé, essayez à nouveau la documentation !

React pour les designers

Si vous avez plutôt une expérience de designer, [ces ressources](#) sont un excellent moyen de commencer.

Ressources JavaScript

La documentation de React suppose une certaine habitude de la programmation en langage JavaScript. Pas besoin d'être un·e expert·e, mais il est plus difficile d'apprendre à la fois React et JavaScript.

Nous vous recommandons de parcourir [cet aperçu JavaScript](#) pour vérifier votre niveau de connaissances. Ça vous prendra entre 30 minutes et une heure, mais vous vous sentirez plus en confiance pour apprendre React.

Astuce

Chaque fois que quelque chose en JavaScript vous semble déroutant, [MDN](#) et [javascript.info](#) sont d'excellents sites pour vérifier. On trouve aussi [des forums de soutien communautaire](#) dans lesquels vous pouvez demander de l'aide.

Tutoriel pratique

Si vous préférez **apprendre par la pratique**, allez voir notre [tutoriel pratique](#). Dans ce tutoriel, nous construisons un jeu de morpion en React. Vous pourriez être tenté·e de l'ignorer sous prétexte que vous n'êtes pas intéressé·e par la construction de jeux—mais donnez-lui sa chance. Les techniques que vous apprenez dans ce tutoriel sont fondamentales pour la construction de n'importe quel type d'appli React, et les maîtriser vous apportera une compréhension profonde de React.

Guide étape par étape

Si vous préférez **apprendre les concepts étape par étape** notre [guide des fondamentaux](#) est le meilleur endroit pour commencer. Chaque chapitre s'appuie sur les connaissances introduites dans les précédents, afin que vous ne manquiez de rien pour avancer.

Penser en React

De nombreux utilisateurs de React estiment que c'est en lisant [Penser en React](#) que React a enfin « cliqué » pour eux. C'est probablement le plus ancien guide pas-à-pas sur React, mais il est toujours aussi pertinent.

Cours recommandés

Parfois les gens préféreront des livres et cours vidéo créés par des tiers à la documentation officielle. Nous maintenons [une liste de ressources fréquemment recommandées](#), dont certaines sont gratuites.

Guides avancés

Une fois que vous serez à l'aise avec les [fondamentaux](#) et que vous aurez joué un peu avec React, vous pourriez être intéressé·e par des sujets plus avancés. Cette section présente les fonctionnalités puissantes, mais moins utilisées, de React, telles que le [contexte](#) et les [refs](#).

Référence de l'API

Cette section de la documentation est utile lorsque vous souhaitez en savoir plus sur une API React spécifique. Par exemple, [la référence de l'API `React.Component`](#) peut vous fournir des détails sur le fonctionnement de `setState()`, et sur les utilités respectives des différentes méthodes de cycle de vie.

Glossaire et FAQ

Le [glossaire](#) contient un aperçu des termes les plus couramment employés dans la documentation de React. Il y a également une section FAQ dédiée aux questions et réponses courtes sur des sujets fréquents, tels que [faire des requêtes AJAX](#), [gérer l'état local des composants](#), et [la structure de fichiers](#).

Se tenir au courant

Le [blog React](#) est la source officielle des mises à jour, par l'équipe de React. Tout ce qui est important, y compris les notes de publication ou les avis de dépréciation, y est publié en priorité.

Vous pouvez également suivre le [compte @reactjs](#) sur Twitter, mais rien d'essentiel ne vous échappera si vous ne lisez que le blog.

Toutes les versions de React ne méritent pas leur propre article de blog, mais vous pouvez trouver un *changelog* détaillé pour chaque version dans le fichier `CHANGELOG.md` du dépôt React, ainsi que sur la page des [Releases](#).

Documentation versionnée

Cette documentation reflète toujours la dernière version stable de React. Depuis React 16, vous pouvez trouver les anciennes versions de la documentation sur une [page séparée](#). Notez que la documentation des versions antérieures est figée au moment de la publication et n'est plus mise à jour par la suite.

Quelque chose vous manque ?

Si quelque chose manque dans la documentation ou si vous en trouvez une partie déroutante, veuillez [créer une issue sur le dépôt de la documentation](#) avec vos suggestions d'amélioration, ou tweetez en mentionnant le compte [@reactjs](#). Nous adorons avoir de vos nouvelles !

Gérer les événements

La gestion des événements pour les éléments React est très similaire à celle des éléments du DOM. Il y a tout de même quelques différences de syntaxe :

- Les événements de React sont nommés en `camelCase` plutôt qu'en minuscules.
- En JSX on passe une fonction comme gestionnaire d'événements plutôt qu'une chaîne de caractères.

Par exemple, le HTML suivant :

```
<button onclick="activateLasers()">
  Activer les lasers
</button>
```

est légèrement différent avec React:

```
<button onClick={activateLasers}>
  Activer les lasers
</button>
```

Autre différence importante : en React, on ne peut pas renvoyer `false` pour empêcher le comportement par défaut. Vous devez appeler explicitement `preventDefault`. Par exemple, en HTML, pour annuler le comportement par défaut des liens qui consiste à ouvrir une nouvelle page, vous pourriez écrire :

```
<a href="#" onclick="console.log('Le lien a été cliqué.'); return false">
  Clique ici
</a>
```

En React, ça pourrait être :

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('Le lien a été cliqué.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Clique ici
    </a>
  );
}
```

Ici, `e` est un événement synthétique. React le définit en suivant les [spécifications W3C](#), afin que vous n'ayez pas à vous préoccuper de la compatibilité entre les navigateurs. Les événements React ne fonctionnent pas tout à fait comme les événements natifs. Pour en apprendre davantage, consultez le guide de référence de `SyntheticEvent`.

Lorsque vous utilisez React, vous n'avez généralement pas besoin d'appeler la méthode `addEventListener` pour ajouter des écouteurs d'événements (*event listeners, NdT*) à un élément du DOM après que celui-ci est créé. À la place, on fournit l'écouteur lors du rendu initial de l'élément.

Lorsque vous définissez un composant en utilisant les [classes ES6](#), il est d'usage que le gestionnaire d'événements soit une méthode de la classe. Par exemple, ce composant `Toggle` affiche un bouton qui permet à l'utilisateur de basculer l'état de "ON" à "OFF".

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // Cette liaison est nécessaire afin de permettre
    // l'utilisation de `this` dans la fonction de rappel.
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

[Essayer dans CodePen](#)

En JSX, vous devez être prudent·e avec l'utilisation de `this` dans les fonctions de rappel. En JavaScript, les méthodes de classes ne sont pas [liées](#) par défaut. Si vous oubliez de lier `this.handleClick` et l'utilisez dans `onClick`, `this` sera `undefined` quand la fonction sera appelée.

Ce n'est pas un comportement spécifique à React, ça fait partie du [fonctionnement normal des fonctions en JavaScript](#). En général, si vous faites référence à une méthode sans l'appeler avec `()`, comme dans `onClick={this.handleClick}`, vous devriez lier cette méthode.

Si vous ne souhaitez pas utiliser `bind`, vous avez deux alternatives possibles. Si vous avez l'habitude d'utiliser la [syntaxe des champs de classes](#), qui est encore expérimentale, vous pourriez l'utiliser pour lier les fonctions de rappel :

```
class LoggingButton extends React.Component {
  // Cette syntaxe nous assure que `this` est bien lié dans la méthode
  handleClick.
  // Attention : cette syntaxe est encore *expérimentale*.
  handleClick = () => {
    console.log('this vaut :', this);
  }

  render() {
```

```

    return (
      <button onClick={this.handleClick}>
        Clique ici
      </button>
    );
}
}

```

Cette syntaxe est activée par défaut dans [Create React App](#).

Si vous n'utilisez pas la syntaxe des champs de classe, vous pouvez utiliser les [fonctions fléchées](#) pour les fonctions de rappel.

```

class LoggingButton extends React.Component {
  handleClick() {
    console.log('this vaut :', this);
  }

  render() {
    // Cette syntaxe nous assure que `this` est bien lié dans la méthode
    handleClick
    return (
      <button onClick={() => this.handleClick()}>
        Clique ici
      </button>
    );
  }
}

```

Cette syntaxe n'est toutefois pas sans défauts, car elle crée une nouvelle fonction de rappel à chaque affichage de `LoggingButton`. Dans la plupart des cas ce n'est pas gênant. Néanmoins, si cette fonction était passée comme prop à des composants plus bas dans l'arbre, ces composants risqueraient de faire des ré-affichages superflus. Nous recommandons donc, en règle générale, de lier ces méthodes dans le constructeur ou d'utiliser un champ de classe afin d'éviter ce genre de problèmes de performances.

Passer des arguments à un gestionnaire d'événements

Au sein d'une boucle, il est courant de vouloir passer un argument supplémentaire à un gestionnaire d'événements. Par exemple, si `id` représente la ligne sélectionnée, on peut faire au choix :

```

<button onClick={(e) => this.deleteRow(id, e)}>Supprimer la ligne</button>
<button onClick={this.deleteRow.bind(this, id)}>Supprimer la ligne</button>

```

Les lignes précédentes sont équivalentes et utilisent respectivement les [fonctions fléchées](#) et `Function.prototype.bind`.

Dans les deux cas, l'argument `e` représente l'événement React qui sera passé en second argument après l'ID. Avec une fonction fléchée, nous devons passer l'argument explicitement, alors qu'avec `bind` tous les arguments sont automatiquement transmis.

[Go to TOC](#)

Bonjour, monde !

Le plus petit exemple de React ressemble à ceci :

```
ReactDOM.render(  
  <h1>Bonjour, monde !</h1>,  
  document.getElementById('root')  
)
```

Il affiche un titre qui dit « Bonjour, monde ! » sur la page.

[Essayer sur CodePen](#)

Cliquez sur le lien ci-dessus pour ouvrir un éditeur en ligne. Vous êtes libres de faire quelques changements et de voir comment ils affectent l'affichage. La plupart des pages de ce guide auront des exemples modifiables comme celui-ci.

Comment lire ce guide

Dans ce guide, nous examinerons les blocs qui constituent une application React : les éléments et les composants. Une fois que vous les aurez maîtrisés, vous pourrez créer des applications complexes à partir de petits blocs réutilisables.

Astuce

Ce guide est destiné aux personnes qui préfèrent **apprendre étape par étape**. Si vous préférez apprendre par la pratique, allez voir notre [tutoriel](#). Vous trouverez peut-être que les deux sont complémentaires.

Ceci est le premier chapitre d'un guide étape par étape à propos des concepts principaux de React. Vous pouvez trouver une liste des chapitres dans la barre latérale de navigation. Si vous lisez ceci depuis un appareil mobile, vous pouvez accéder à la navigation en appuyant sur le bouton situé dans le coin en bas à droite de votre écran.

Chacun des chapitres de ce guide s'appuie sur les connaissances introduites dans les chapitres précédents. **Vous pouvez apprendre l'essentiel de React en lisant les chapitres du guide « Fondamentaux » dans l'ordre où ils apparaissent dans la barre latérale.** Par exemple, le prochain chapitre s'intitule « [Introduction à JSX](#) ».

Niveau de connaissances supposé

React est une bibliothèque JavaScript, donc nous supposerons que vous avez une compréhension suffisante du langage JavaScript. **Si vous ne vous sentez pas à l'aise, nous vous recommandons de passer par un tutoriel JavaScript pour vérifier votre niveau de connaissances** et vous permettre de suivre ce guide sans être perdu·e. Il vous prendra entre 30 minutes et une heure environ, mais au moins vous n'aurez pas le sentiment d'apprendre React et JavaScript en même temps.

Remarque

Ce guide utilise occasionnellement quelques nouvelles syntaxes de JavaScript dans les exemples. Si vous n'avez pas travaillé avec JavaScript ces dernières années, [ces trois points](#) devraient vous aider.

Commençons !

Continuez à défiler pour atteindre le lien vers le [prochain chapitre de ce guide](#) juste avant le pied de page.

[Go to TOC](#)

Composants d'ordre supérieur

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

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.

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

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

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

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

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

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 enrobé 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

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

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.

Construire vos propres 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.

Construire vos propres Hooks vous permet d'extraire la logique d'un composant sous forme de fonctions réutilisables.

Lorsque nous apprenons 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

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

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 [apris 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

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 = [
];

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))}>
        >
          {friendList.map(friend => (
            <option key={friend.id} value={friend.id}>
              {friend.name}
            </option>
          ))}
        </select>
      </>
    );
}
```

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](#).

Utiliser un Hook d'effet

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

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

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

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

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

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

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

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 'Chargement...';
      }
      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

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

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

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

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 :

```
ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // Exécute l'effet 1
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Nettoie l'effet 1
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // Exécute l'effet 2
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

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 suggérera un correctif.

Prochaines étapes

Félicitations ! C'était une bien longue page, mais avec un peu de chance la plupart de vos questions à propos des effets ont pu y trouver une réponse. Vous avez appris à utiliser le Hook d'état et le Hook d'effet, qui vous ouvrent *beaucoup* de possibilités à eux deux. Ils couvrent la majorité des cas d'usage pour les classes —et sinon, peut-être trouverez-vous les [Hooks supplémentaires](#) bien utiles.

Nous commençons également à deviner comment les Hooks peuvent résoudre les problématiques listées dans les [raisons](#). Nous avons vu comment le nettoyage des effets évite la duplication de code dans `componentDidUpdate` et `componentWillUnmount`, regroupe le code par sujet, et aide à éviter les bugs. Enfin, nous avons appris comment séparer les effets selon leur finalité, ce qui était totalement impossible avec les classes.

A ce stade vous vous demandez peut-être comment les Hooks fonctionnent. Comment React sait-il quel appel de `useState` correspond à quelle variable d'état local d'un affichage à l'autre ? Comment React « associe-t-il » l'effet précédent au suivant à chaque mise à jour ? **Rendez-vous dans la prochaine page pour apprendre les règles des Hooks, qui sont indispensables à leur bon fonctionnement.**

FAQ 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 contient les réponses aux questions les plus fréquentes sur les [Hooks](#).

- **Stratégie d'adoption**

- Quelles versions de React incluent les Hooks ?
- Dois-je réécrire tous mes composants à base de classe ?
- Que puis-je faire avec les Hooks qu'il est impossible de faire avec des classes ?
- Quelle proportion de mes connaissances en React reste pertinente ?
- Dois-je utiliser des Hooks, des classes ou un mélange des deux ?
- Est-ce que les Hooks couvrent tous les cas d'utilisation des classes ?
- Est-ce que les Hooks remplacent les props de rendu et les composants d'ordre supérieur ?
- Qu'est-ce que les Hooks changent pour les API populaires telles que Redux `connect()` et React Router ?
- Est-ce que les Hooks sont compatibles avec le typage statique ?
- Comment tester des composants utilisant des Hooks ?
- Qu'est-ce que les règles de linting imposent ?

- **Des classes aux Hooks**

- Quelles sont les correspondances entre les méthodes de cycle de vie et les Hooks ?
- Comment charger des données distantes avec les Hooks ?
- Existe-t-il un équivalent aux variables d'instances ?
- Dois-je utiliser une ou plusieurs variables d'état local ?
- Puis-je exécuter un effet seulement lors des mises à jour ?
- Comment récupérer les props ou l'état local précédents ?
- Pourquoi vois-je des props ou un état local obsolètes dans ma fonction ?
- Comment puis-je implémenter `getDerivedStateFromProps` ?
- Existe-t-il un équivalent à `forceUpdate` ?
- Puis-je créer une ref vers une fonction composant ?
- Comment puis-je mesurer un nœud DOM ?
- Que signifie `const [thing, setThing] = useState()` ?

- **Optimisations des performances**

- Puis-je sauter un effet lors des mises à jour ?
- Est-il acceptable d'omettre les fonctions du tableau de dépendances ?
- Que faire quand mes dépendances d'effet changent trop souvent ?
- Comment puis-je implémenter `shouldComponentUpdate` ?
- Comment mémoiser les calculs ?
- Comment créer paresseusement des objets coûteux ?
- La création de fonctions à la volée pendant le rendu ralentit-elle les Hooks ?
- Comment éviter de transmettre des fonctions de rappel ?
- Comment lire une valeur changeant fréquemment avec `useCallback` ?

- [Sous le capot](#)
 - [Comment React associe-t-il les appels de Hooks avec les composants ?](#)
 - [Quelles sont les sources d'inspiration des Hooks ?](#)

Stratégie d'adoption

Quelles versions de React incluent les Hooks ?

Depuis la version 16.8.0, React embarque une implémentation stable de React Hooks pour :

- React DOM
- React Native
- React DOM Server
- Le moteur de rendu de test de React
- Le moteur de rendu superficiel de React

Remarquez que **pour activer les Hooks, tous les paquets React doivent être en version 16.8.0 ou supérieure.** Les Hooks ne fonctionneront pas si vous oubliez de mettre à jour React DOM, par exemple.

[React Native 0.59](#) et ultérieurs prennent en charge les Hooks.

Dois-je réécrire tous mes composants à base de classe ?

Non. Il n'est [pas prévu](#) de retirer les classes de React : nous avons tous besoin de continuer à livrer nos produits et ne pouvons pas nous permettre de réécrire tout le code. Nous recommandons d'essayer les Hooks dans de nouveaux composants et projets.

Que puis-je faire avec les Hooks qu'il est impossible de faire avec des classes ?

Les Hooks offrent un nouveau moyen puissant et expressif de réutiliser des fonctionnalités entre composants. [« Construire vos propres Hooks »](#) offre un aperçu des possibilités. [Cet article](#) (en anglais) écrit par un membre de l'équipe noyau de React explore plus en détail les nouvelles possibilités apportées par les Hooks.

Quelle proportion de mes connaissances en React reste pertinente ?

Les Hooks sont un moyen plus direct d'utiliser les fonctionnalités de React que vous connaissez déjà, telles que l'état local, le cycle de vie, le contexte et les refs. Ils ne changent pas fondamentalement la façon dont React fonctionne, et vos connaissances des composants, des props, et du flux de données descendant sont toujours valides.

Les Hooks ont tout de même une courbe d'apprentissage. Si quelque chose manque dans leur documentation, [créez un ticket](#) sur le dépôt GitHub et nous essaierons de vous aider.

Dois-je utiliser des Hooks, des classes ou un mélange des deux ?

Quand vous serez prêt·e, nous vous conseillons de commencer à essayer les Hooks dans les nouveaux composants que vous écrirez. Assurez-vous que chaque membre de votre équipe soit partant·e pour les utiliser, et à l'aise avec cette documentation. Nous déconseillons de réécrire vos classes existantes avec les Hooks, sauf si vous aviez déjà prévu de les réécrire de toute façon (ex. pour corriger des bugs).

Vous ne pouvez pas utiliser les Hooks *à l'intérieur* d'un composant à base de classe, mais vous pouvez complètement mélanger classes et fonctions composants utilisant des Hooks dans une même arborescence. Qu'un composant soit une classe ou une fonction utilisant les Hooks ne constitue qu'un détail d'implémentation de ce composant. Sur le long terme, nous nous attendons à ce que l'essentiel des composants React soient écrits à base de Hooks.

Est-ce que les Hooks couvrent tous les cas d'utilisation des classes ?

Notre but est que les Hooks couvrent tous les cas d'utilisation des classes dès que possible. Il n'existe pas pour l'instant d'équivalent en Hook pour les méthodes de cycle de vie moins courantes que sont `getSnapshotBeforeUpdate`, `getDerivedStateFromError` et `componentDidCatch`, mais nous prévoyons de les ajouter rapidement.

Les Hooks en sont encore à leur débuts, et quelques bibliothèques tierces peuvent ne pas être compatibles avec les Hooks à l'heure actuelle.

Est-ce que les Hooks remplacent les props de rendu et les composants d'ordre supérieur ?

Souvent, les props de rendu et les composants d'ordre supérieur n'affichent qu'un seul enfant. Nous pensons que les Hooks simplifient ce cas d'utilisation. Ces deux approches restent pertinentes (par exemple, un composant de défilement virtuel pourrait avoir une prop `renderItem`, ou un composant de conteneur visuel pourrait avoir sa propre structure DOM). Mais dans la plupart des cas, les Hooks seront suffisants et pourront aider à réduire l'imbrication dans votre arborescence de composants.

Qu'est-ce que les Hooks changent pour les API populaires telles que Redux `connect()` et React Router ?

Vous pouvez continuer à utiliser les mêmes API que d'habitude ; elles fonctionneront toujours comme avant.

Depuis sa version v7.1.0, React-Redux prend en charge l'API des Hooks et fournit des Hooks tels que `useDispatch` et `useSelector`.

React Router prend en charge les Hooks depuis sa v5.1.

D'autres bibliothèques pourront prendre en charge les Hooks à l'avenir.

Est-ce que les Hooks sont compatibles avec le typage statique ?

Les Hooks ont été conçus avec le typage statique à l'esprit. Comme ce sont des fonctions, il est plus facile de les typer correctement que d'autres approches telles que les composants d'ordre supérieur. Les dernières définitions Flow et TypeScript pour React prennent en charge les Hooks React.

Surtout, les Hooks personnalisés vous donnent la possibilité de restreindre l'API React si vous souhaitez les typer de façon plus stricte. React vous fournit des primitives, mais vous pouvez les combiner de façons différentes de celles que nous fournissons d'entrée de jeu.

Comment tester des composants utilisant des Hooks ?

Du point de vue de React, un composant utilisant des Hooks est un composant normal. Si votre solution de test ne repose pas sur des fonctionnements internes de React, tester des composants avec des Hooks ne devrait pas être différent de la façon dont vous testez vos composants habituellement.

Note

[Testing Recipes](#) inclut plusieurs exemples que vous pouvez copier-coller.

Par exemple, prenons ce composant de comptage :

```
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>
  );
}
```

Nous le testerons en utilisant React DOM. Pour être certains que le comportement correspond à ce qui se passerait dans le navigateur, nous enroberons le code d'affichage et de mise à jour par des appels à `ReactTestUtils.act()` :

```
import React from 'react';
import ReactDOM from 'react-dom';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});
```

```

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // Test du premier rendu et de son effet
  act(() => {
    ReactDOM.render(<Counter />, container);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('Vous avez cliqué 0 fois');
  expect(document.title).toBe('Vous avez cliqué 0 fois');

  // Test du second rendu et de son effet
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('Vous avez cliqué 1 fois');
  expect(document.title).toBe('Vous avez cliqué 1 fois');
});

```

Les appels à `act()` vont aussi traiter les effets qu'ils contiennent.

Si vous souhaitez tester un Hook personnalisé, c'est possible en créant un composant dans votre test, et en utilisant le Hook depuis celui-ci. Vous pourrez alors tester le composant que vous venez de créer.

Pour réduire le code générique, nous vous conseillons d'utiliser [React Testing Library](#) qui est conçu de manière à encourager l'écriture de tests utilisant les composants comme le feraient les utilisateurs finaux.

Pour plus d'information, consultez [Testing Recipes](#).

Qu'est-ce que les règles de linting imposent ?

Nous mettons à disposition un [plugin ESLint](#) qui impose [les règles des Hooks](#) pour éviter les bugs. Il part du principe que toute fonction commençant par "`use`" suivie d'une lettre majuscule est un Hook. Nous admettons que cette heuristique n'est pas parfaite et peut déclencher des faux positifs, mais sans convention au niveau de l'écosystème, il n'existe aucun moyen de faire fonctionner les Hooks correctement—and des noms plus longs décourageront l'adoption des Hooks ou le respect des conventions par la communauté.

En particulier, la règle impose que :

- Les appels de Hooks soient situés soit à l'intérieur d'une fonction nommée en casse `PascalCase` (supposée être un composant) ou d'une autre fonction `useSomething` (supposée être un Hook personnalisé).
- Les Hooks soient appelés dans le même ordre à chaque rendu.

Il existe quelques autres heuristiques, et elles changeront peut-être avec le temps, au fur et à mesure que nous peaufinons la règle pour améliorer la découverte de bugs tout en évitant les faux positifs.

Des classes aux Hooks

Quelles sont les correspondances entre les méthodes de cycle de vie et les Hooks ?

- `constructor` : les fonctions composants n'ont pas besoin d'un constructeur. Vous pouvez initialiser l'état local lors de l'appel à `useState`. Si le calcul de l'état local initial est trop coûteux, vous pouvez passer une fonction à `useState`.
- `getDerivedStateFromProps` : planifiez plutôt une mise à jour [pendant le rendu](#).
- `shouldComponentUpdate` : voyez `React.memo` [ci-dessous](#).
- `render` : c'est le corps-même de la fonction composant.
- `componentDidMount`, `componentDidUpdate`, `componentWillUnmount` : le Hook `useEffect` peut exprimer toutes les combinaisons de celles-ci (y compris des cas [moins fréquents](#)).
- `getSnapshotBeforeUpdate`, `componentDidCatch` et `getDerivedStateFromError` : il n'existe pas encore de Hook équivalent pour ces méthodes, mais ils seront ajoutés prochainement.

Comment charger des données distantes avec les Hooks ?

Voici une [petite démo](#) pour vous aider à démarrer. Pour en apprendre davantage, jetez un œil à [cet article](#) (en anglais) sur le chargement de données distantes avec les Hooks.

Existe-t-il un équivalent aux variables d'instances ?

Oui ! Le Hook `useRef()` n'est pas seulement pour les refs au DOM. L'objet "ref" est un conteneur générique dont la propriété `current` est modifiable et peut contenir n'importe quelle valeur, de la même façon qu'une propriété d'instance dans une classe.

Vous pouvez lui affecter une valeur depuis `useEffect` :

```
function Timer() {
  const intervalRef = useRef();

  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });
  // ...
}
```

Si nous avions juste voulu définir une horloge, nous n'aurions pas eu besoin de la ref (`id` pouvait rester local à l'effet) mais ça peut être utile si nous voulons arrêter l'horloge depuis un gestionnaire d'événements.

```
// ...
function handleClick() {
  clearInterval(intervalRef.current);
}
// ...
```

Conceptuellement, vous pouvez vous représenter les refs comme des variables d'instance dans une classe. À moins que vous n'ayez recours à de l'[initialisation paresseuse](#), évitez d'écrire dans vos refs pendant le rendu : ça peut donner des comportements hasardeux. Au lieu de ça, vous voudrez généralement modifier les refs au sein de gestionnaires d'événements ou d'effets.

Dois-je utiliser une ou plusieurs variables d'état local ?

Si vous avez l'habitude des classes, vous serez peut-être tenté·e de toujours appeler `useState()` une seule fois, en mettant tout l'état local dans un unique objet. Si vous y tenez, c'est tout à fait possible. Voici un exemple d'un composant qui piste le mouvement de la souris. Nous gardons sa position et sa taille dans l'état local :

```
function Box() {
  const [state, setState] = useState({ left: 0, top: 0, width: 100, height: 100 });
}
// ...
```

Maintenant, disons que nous voulons écrire un bout de code pour modifier `left` et `top` quand l'utilisateur bouge la souris. Voyez comme nous devons fusionner manuellement ces champs dans l'état local précédent :

```
// ...
useEffect(() => {
  function handleWindowMouseMove(e) {
    // La décomposition de "...state" permet de s'assurer qu'on ne « perd » pas
    // width et height
    setState(state => ({ ...state, left: e.pageX, top: e.pageY }));
  }
  // Remarque : cette implémentation est un peu simplifiée
  window.addEventListener('mousemove', handleWindowMouseMove);
  return () => window.removeEventListener('mousemove', handleWindowMouseMove);
}, []);
```

C'est dû au fait que lorsque nous mettons à jour une variable de l'état local, nous *remplaçons* sa valeur, alors qu'avec `this.setState` dans une classe, on *fusionne* les champs mis à jour dans l'objet.

Si la fusion automatique vous manque, vous pouvez écrire un Hook personnalisé `useLegacyState` qui fusionne les mises à jour de l'état local. Cependant, **nous recommandons plutôt de séparer l'état local en de multiple variables d'état en se basant sur celles qui ont tendance à changer de valeur ensemble.**

Par exemple, nous pourrions découper l'état local de notre composant en deux objets `position` et `size`, et toujours remplacer la `position` sans avoir besoin de fusionner :

```
function Box() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  const [size, setSize] = useState({ width: 100, height: 100 });

  useEffect(() => {
    function handleWindowMouseMove(e) {
      setPosition({ left: e.pageX, top: e.pageY });
    }
    // ...
  })
}
```

Séparer les variables d'état indépendantes présente un intérêt supplémentaire : il devient facile d'extraire une partie de la logique dans un Hook personnalisé, par exemple :

```
function Box() {
  const position = useWindowPosition();
  const [size, setSize] = useState({ width: 100, height: 100 });
  // ...
}

function useWindowPosition() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  useEffect(() => {
    // ...
  }, []);
  return position;
}
```

Remarquez comme nous avons pu déplacer l'appel à `useState` pour la variable d'état local `position` et l'effet associé dans un Hook personnalisé sans rien changer au code. Si tout l'état local était un unique objet, extraire cet aspect aurait été plus difficile.

Les deux approches sont possibles : mettre tout l'état local dans un unique appel à `useState` ou avoir un appel à `useState` par champ. La lisibilité des composants sera fonction de l'équilibre que vous trouverez entre ces deux extrêmes, et du regroupement des états locaux associés en quelques variables d'état indépendantes. Si la logique de l'état local devient trop complexe, nous vous conseillons de plutôt la [gérer avec un réducteur](#) ou un Hook personnalisé.

Puis-je exécuter un effet seulement lors des mises à jour ?

C'est un cas d'utilisation assez rare. Si vous en avez besoin, vous pouvez [utiliser une ref modifiable](#) pour stocker manuellement un booléen indiquant si vous êtes sur le premier rendu ou un rendu postérieur, et vérifier ensuite ce drapeau dans votre effet. (Si vous faites ça régulièrement, vous voudrez sans doute créer un Hook personnalisé pour ça.)

Comment récupérer les props ou l'état local précédents ?

Actuellement, vous pouvez le faire manuellement [avec une ref](#) :

```
function Counter() {
  const [count, setCount] = useState(0);

  const prevCountRef = useRef();
  useEffect(() => {
    prevCountRef.current = count;
  });
}
```

```
const prevCount = prevCountRef.current;
return <h1>Maintenant : {count}, avant : {prevCount}</h1>;
}
```

Ça peut sembler un peu biscornu mais vous pouvez l'extraire dans un Hook personnalisé :

```
function Counter() {
  const [count, setCount] = useState(0);
  const prevCount = usePrevious(count);
  return <h1>Maintenant : {count}, avant : {prevCount}</h1>;
}

function usePrevious(value) {
  const ref = useRef();
  useEffect(() => {
    ref.current = value;
  });
  return ref.current;
}
```

Remarquez que ça fonctionne pour les props, l'état local, et toute autre valeur calculée.

```
function Counter() {
  const [count, setCount] = useState(0);

  const calculation = count + 100;
  const prevCalculation = usePrevious(calculation);
  // ...
}
```

Il est possible qu'à l'avenir React fournit un Hook `usePrevious` prêt à l'emploi, puisque c'est un cas d'utilisation assez fréquent.

Voir aussi [l'approche recommandée pour un état local dérivé](#).

Pourquoi vois-je des props ou un état local obsolètes dans ma fonction ?

Toute fonction au sein d'un composant, y compris les gestionnaires d'événements et les effets, « voit » les props et l'état local en vigueur lors du rendu qui les a créées. Par exemple, prenez ce genre de code :

```
function Example() {
  const [count, setCount] = useState(0);

  function handleAlertClick() {
    setTimeout(() => {
      alert('Vous avez cliqué à ' + count);
    }, 3000);
  }

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

```

```
Afficher un message
  </button>
</div>
);
}
```

Si vous cliquez d'abord sur « Afficher un message » puis incrémentez tout de suite le compteur, le message affichera la variable `count` **telle qu'elle était lors du clic sur le bouton « Afficher un message »**. Ça évite les bugs causés par du code qui suppose que les props et l'état local ne changent pas.

Si vous souhaitez explicitement lire le **tout dernier** état depuis une fonction de rappel asynchrone, vous pouvez le conserver dans [une ref](#), la modifier puis la relire.

Pour finir, une autre explication possible pour vos props ou votre état périmés résiderait dans votre utilisation incorrecte de l'optimisation du hook par « tableau de dépendances », auquel il manquerait certaines valeurs. Par exemple, si un effet indique `[]` comme deuxième argument mais lit `someProp` en interne, il continuera à « voir » la valeur initiale de `someProp`. La solution consiste soit à retirer l'argument de tableau de dépendances, soit à le corriger. Voici [comment y gérer des fonctions](#) ainsi que [d'autres stratégies habituelles](#) pour exécuter des effets moins souvent sans ignorer à tort des dépendances.

Remarque

Nous proposons une règle ESLint `exhaustive-deps` dans le cadre du module `eslint-plugin-react-hooks`. Elle vous avertit lorsque les dépendances spécifiées semblent incorrectes et vous propose un correctif.

Comment puis-je implémenter `getDerivedStateFromProps` ?

Même si vous n'en avez probablement [pas besoin](#), dans les rares cas où c'est nécessaire (comme implémenter un composant `<Transition>`) vous pouvez mettre à jour l'état local en plein rendu. React va rafraîchir le composant avec l'état local mis à jour immédiatement après être sorti du premier rendu afin que ça ne soit pas trop coûteux.

Ici, nous stockons la valeur précédente de la prop `row` dans une variable de l'état local afin que nous puissions les comparer :

```
function ScrollView({row}) {
  const [isScrollingDown, setIsScrollingDown] = useState(false);
  const [prevRow, setPrevRow] = useState(null);

  if (row !== prevRow) {
    // Row a changé depuis le dernier rendu. Met à jour isScrollingDown.
    setIsScrollingDown(prevRow !== null && row > prevRow);
    setPrevRow(row);
  }

  return `On défile vers le bas : ${isScrollingDown}`;
}
```

Ça peut sembler étrange à première vue, mais `getDerivedStateFromProps` avait précisément été conçue pour des mises à jour pendant le rendu.

Existe-t-il un équivalent à `forceUpdate` ?

Les deux Hooks `useState` et `useReducer` abandonnent la mise à jour si la valeur suivante est la même que la valeur précédente. Modifier l'état local en place et appeler `setState` ne causera pas de rafraîchissement.

Généralement, vous ne devez pas modifier l'état local directement en React. Cependant, à titre d'échappatoire, vous pouvez maintenir un compteur incrémental pour forcer un rafraîchissement même si l'état local n'a pas changé :

```
const [ignored, forceUpdate] = useReducer(x => x + 1, 0);

function handleClick() {
  forceUpdate();
}
```

Évitez autant que possible d'utiliser cette approche.

Puis-je créer une ref vers une fonction composant ?

Vous ne devriez pas en avoir besoin souvent, mais vous pouvez exposer quelques méthodes impératives à un composant parent avec le Hook `useImperativeHandle`.

Comment puis-je mesurer un nœud DOM ?

Une façon rudimentaire de mesurer la position ou les dimensions d'un nœud DOM consiste à utiliser une [ref avec fonction de rappel](#). React appellera la fonction de rappel chaque fois que la ref est attachée à un nœud différent. Voici une [petite démo](#) :

```
function MeasureExample() {
  const [height, setHeight] = useState(0);

  const measuredRef = useCallback(node => {
    if (node !== null) {
      setHeight(node.getBoundingClientRect().height);
    }
  }, []);

  return (
    <>
      <h1 ref={measuredRef}>Bonjour, monde</h1>
      <h2>L'en-tête ci-dessus fait {Math.round(height)}px de haut</h2>
    </>
  );
}
```

Nous avons évité `useRef` dans cet exemple parce qu'un objet ref ne nous notifie pas des *changements* de la valeur actuelle de la ref. Une ref avec fonction de rappel garantit que [même si un composant enfant affiche ultérieurement le nœud DOM mesuré](#) (ex. en réaction à un clic), nous serons quand même notifiés dans le composant parent et pourrons mettre les mesures à jour.

Remarquez que nous passons `[]` comme tableau de dépendances à `useCallback`. C'est pour nous assurer que notre ref à fonction de rappel ne change pas d'un rendu à l'autre, afin que React ne nous appelle pas pour rien.

Dans cet exemple, la ref avec fonction de rappel ne sera appelée que lors du montage et du démontage du composant, puisque le composant `<h1>` reste présent d'un rendu au suivant. Si vous souhaitez être notifié·e à chaque redimensionnement, vous voudrez peut-être utiliser `ResizeObserver` ou un Hook tiers basé dessus.

Si vous le souhaitez, vous pouvez [extraire cette logique](#) dans un Hook réutilisable :

```
function MeasureExample() {
  const [rect, ref] = useClientRect();
  return (
    <>
      <h1 ref={ref}>Bonjour, monde</h1>
      {rect !== null &&
        <h2>L'en-tête ci-dessus fait {Math.round(rect.height)}px de haut</h2>
      }
    </>
  );
}

function useClientRect() {
  const [rect, setRect] = useState(null);
  const ref = useCallback(node => {
    if (node !== null) {
      setRect(node.getBoundingClientRect());
    }
  }, []);
  return [rect, ref];
}
```

Que signifie `const [thing, setThing] = useState()` ?

Si vous n'avez pas l'habitude de cette syntaxe, allez voir l'[explication](#) dans la documentation du Hook d'état.

Optimisations des performances

Puis-je sauter un effet lors des mises à jour ?

Oui. Reportez-vous au [déclenchement conditionnel d'un effet](#). Remarquez qu'oublier de gérer des mises à jour est souvent [source de bugs](#), ce qui explique pourquoi ce n'est pas le comportement par défaut.

Est-il acceptable d'omettre les fonctions du tableau de dépendances ?

En règle générale, non.

```
function Example({ someProp }) {
  function doSomething() {
    console.log(someProp);
  }
}
```

```
useEffect(() => {
  doSomething();
}, []); // 🔴 Ce n'est pas fiable (ça appelle `doSomething` qui utilise
`someProp`)
}
```

Il n'est pas facile de se souvenir du détail des props et de l'état local utilisés par les fonctions hors de l'effet. C'est pourquoi **vous voudrez généralement déclarer les fonctions dont votre effet a besoin à l'intérieur de celui-ci.** Il devient alors facile de voir de quelles valeurs de la portée du composant dépend cet effet :

```
function Example({ someProp }) {
  useEffect(() => {
    function doSomething() {
      console.log(someProp);
    }

    doSomething();
  }, [someProp]); // ✅ OK (notre effet n'utilise que `someProp`)
}
```

Si après ça vous n'utilisez toujours pas de valeurs issues de la portée du composant, vous pouvez sans problème spécifier `[]` :

```
useEffect(() => {
  function doSomething() {
    console.log('bonjour');
  }

  doSomething();
}, []); // ✅ OK dans ce cas précis car nous n'utilisons *aucune* valeur de la
portée du composant
```

Selon votre cas, vous trouverez quelques options supplémentaires plus bas dans cette page.

Remarque

Nous mettons à disposition la règle ESLint `exhaustive-deps` dans le cadre du module `eslint-plugin-react-hooks`. Elle vous aide à trouver les composants qui ne gèrent pas correctement les mises à jour.

Voyons en quoi c'est important.

Si vous précisez une [liste de dépendances](#) comme dernier argument de `useEffect`, `useLayoutEffect`, `useMemo`, `useCallback`, ou `useImperativeHandle`, cette liste doit inclure toutes les valeurs utilisées dans la fonction passée qui participent au flux de données de React. Ça inclut les props, l'état local, et toute valeur qui en découle.

Le **seul cas** pour lequel vous pouvez sereinement omettre une fonction de la liste des dépendances, c'est lorsque rien à l'intérieur (y compris dans les autres fonctions qu'elle appelle) ne référence les props, l'état local ou des valeurs qui en découlent. L'exemple suivant a ce problème :

```

function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId); // Utilise
    la prop productId
    const json = await response.json();
    setProduct(json);
  }

  useEffect(() => {
    fetchProduct();
  }, []); // ❌ Erroné car `fetchProduct` utilise `productId`
  // ...
}

```

Le correctif recommandé consiste à déplacer la fonction dans votre effet. Ça facilite le repérage des props et variables d'état que votre effet utilise, pour garantir qu'elles sont toutes déclarées :

```

function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    // En déplaçant cette fonction dans l'effet, on voit clairement quelles
    valeurs il utilise.
    async function fetchProduct() {
      const response = await fetch('http://myapi/product/' + productId);
      const json = await response.json();
      setProduct(json);
    }

    fetchProduct();
  }, [productId]); // ✅ Correct car notre effet n'utilise que `productId`
  // ...
}

```

Ça permet aussi de gérer les réponses trop tardives grâce à des variables locales à l'effet :

```

useEffect(() => {
  let ignore = false;
  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId);
    const json = await response.json();
    if (!ignore) setProduct(json);
  }

  fetchProduct();
  return () => { ignore = true };
}, [productId]);

```

Nous avons déplacé la fonction dans l'effet, donc cette variable n'a pas à figurer dans la liste des dépendances.

Astuce

Jetez un coup d'œil à [cette petite démo](#) et [cet article](#) (en anglais) pour en apprendre davantage sur le chargement de données distantes avec les Hooks.

Si pour une raison ou une autre vous ne pouvez pas déplacer la fonction dans l'effet, vous avez d'autres options :

- **Vous pouvez essayer de déplacer la fonction hors du composant.** Dans ce cas, vous êtes sûr·e qu'elle ne pourra pas référencer des props ou variables d'état, et qu'elle n'a donc pas besoin de figurer dans la liste des dépendances.
- Si la fonction que vous appelez est un calcul pur et qu'on peut sereinement l'appeler pendant le rendu, vous pouvez **l'appeler plutôt hors de l'effet** et faire dépendre l'effet de la valeur qu'elle renvoie.
- En dernier recours, vous pouvez **ajouter une fonction aux dépendances de l'effet mais enrober sa définition** dans un Hook `useCallback`. Ça garantit qu'elle ne changera pas à chaque rendu sauf si ses propres dépendances changent aussi :

```
function ProductPage({ productId }) {
  // ✅ Enrobe avec useCallback pour éviter de changer à chaque rendu
  const fetchProduct = useCallback(() => {
    // ... Fait un truc avec productId ...
  }, [productId]); // ✅ Toutes les dépendances de useCallback sont spécifiées

  return <ProductDetails fetchProduct={fetchProduct} />;
}

function ProductDetails({ fetchProduct }) {
  useEffect(() => {
    fetchProduct();
  }, [fetchProduct]); // ✅ Toutes les dépendances de useEffect sont spécifiées
  // ...
}
```

Remarquez que dans cet exemple nous **devons** garder la fonction dans la liste des dépendances. On s'assure ainsi qu'une modification à la prop `productId` de `ProductPage` déclenchera automatiquement un nouveau chargement de données distantes dans le composant `ProductDetails`.

Que faire quand mes dépendances d'effet changent trop souvent ?

Il arrive que votre effet utilise un état qui change trop fréquemment. Vous pourriez alors être tenté·e d'omettre cet état de la liste des dépendances, mais ça engendre souvent des bugs :

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1); // Cet effet dépend de l'état `count`
    }, 1000);
    return () => clearInterval(id);
  }, []); // 🚨 Bug : `count` n'est pas listé comme dépendance

  return <h1>{count}</h1>;
}
```

La liste de dépendances vide, `[]`, signifie que l'effet ne sera exécuté qu'une fois au montage du composant, et non à chaque rafraîchissement. Le problème vient du fait que dans la fonction de rappel passée à `setInterval`, la valeur de `count` ne va pas changer, car on a créé une fermeture lexicale (*closure, NdT*) avec `count` à `0`, tel qu'elle était lorsque la fonction de rappel de l'effet s'est exécutée. À chaque seconde, cette fonction appelle `setCount(0 + 1)`, de sorte que le compteur ne dépasse jamais 1.

On pourrait corriger le bug en spécifiant `[count]` comme liste de dépendances, mais ça réinitialiserait notre horloge à chaque modification. En pratique, chaque `setInterval` aurait une chance de s'exécuter avant d'être réinitialisé (comme pour un `setTimeout`). Ce n'est peut-être pas souhaitable. Pour corriger ça, nous pouvons utiliser la version basée fonction de `useState`. Elle nous permet d'indiquer *comment* l'état change, sans référencer l'état *actuel* :

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1); // ✓ Ça ne dépend pas de la variable `count` issue de
    la portée
      }, 1000);
    return () => clearInterval(id);
  }, []);
  return <h1>{count}</h1>;
}
```

(L'identité de la fonction `setCount` est garantie stable, il est donc naturel de l'omettre.)

À présent, la fonction de rappel de `setInterval` est appelée une fois par seconde, mais à chaque fois l'appel interne à `setCount` peut utiliser une valeur à jour de `count` (appelée `c` dans la fonction de rappel ci-dessus).

Pour des cas plus complexes (comme lorsqu'un état dépend d'un autre état), essayez de déplacer la logique de mise à jour de l'état hors de l'effet avec le Hook `useReducer`. [Cet article](#) (en anglais) vous donne un exemple de cette approche. **L'identité de la fonction `dispatch` fournie par `useReducer` est garantie stable**, même si la fonction de réduction est déclarée dans le composant et lit ses props.

En dernier recours, si vous voulez quelque chose de similaire au `this` d'une classe, vous pouvez utiliser [une ref](#) pour stocker une donnée modifiable. Vous pouvez alors y écrire et la relire. Par exemple :

```
function Example(props) {
  // Garde les dernières props dans une ref.
  const latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });

  useEffect(() => {
    function tick() {
      // Lit les dernières props en vigueur
      console.log(latestProps.current);
    }
  });
}
```

```
const id = setInterval(tick, 1000);
return () => clearInterval(id);
}, []); // Cet effet n'est jamais ré-exécuté
}
```

Ne faites ça que si vous n'avez pas pu trouver de meilleure solution, car se reposer sur des mutations rend les composants plus imprévisibles. Si vous n'arrivez pas à trouver une approche adaptée pour votre besoin, [créez un ticket](#) avec un exemple exécutable de code pour que nous puissions essayer de vous aider.

Comment puis-je implémenter `shouldComponentUpdate` ?

Vous pouvez enrober une fonction composant avec `React.memo` pour comparer superficiellement ses props :

```
const Button = React.memo((props) => {
  // votre composant
});
```

Ce n'est pas un Hook car ce n'est pas composable, alors que les Hooks le sont. `React.memo` est équivalent à `PureComponent`, mais ne compare que les props. (Vous pouvez aussi ajouter un second argument pour spécifier une fonction de comparaison personnalisée qui prendra en arguments les anciennes et nouvelles props. Si elle renvoie `true`, la mise à jour est évitée.)

`React.memo` ne compare pas l'état local car il n'y a pas d'unique objet d'état local à comparer. Mais vous pouvez rendre les descendants purs également, ou même [optimiser les descendants individuellement avec `useMemo`](#).

Comment mémoïser les calculs ?

Le Hook `useMemo` vous permet de mettre en cache les calculs à travers les rendus en « se souvenant » du dernier calcul :

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Ce code appelle `computeExpensiveValue(a, b)`. Mais si les dépendances `[a, b]` n'ont pas changé depuis la dernière fois, `useMemo` saute le second appel et réutilise simplement la dernière valeur renvoyée.

Rappelez-vous que la fonction passée à `useMemo` s'exécute pendant le rendu. N'y faites rien que vous ne feriez normalement pendant le rendu. Par exemple, les effets de bord sont du ressort de `useEffect`, pas de `useMemo`.

Vous pouvez vous appuyer sur `useMemo` pour les optimisations de performances, mais pas comme une garantie sémantique. À l'avenir, React pourrait très bien choisir « d'oublier » certaines valeurs préalablement mémoïsées et de les recalculer lors du rendu suivant, par exemple pour libérer de la mémoire pour les composants non visibles. Écrivez votre code de façon à ce qu'il fonctionne toujours sans `useMemo`, et ajoutez-le ensuite pour optimiser les performances. (Pour les rares cas où une valeur ne doit *jamais* être recalculée, vous pouvez [l'initialiser paresseusement](#) dans une ref.)

Vous pouvez aussi utiliser `useMemo` pour éviter le rafraîchissement coûteux d'un enfant :

```
function Parent({ a, b }) {
  // N'est rafraîchi que si `a` change :
  const child1 = useMemo(() => <Child1 a={a} />, [a]);
  // N'est rafraîchi que si `b` change :
  const child2 = useMemo(() => <Child2 b={b} />, [b]);
  return (
    <>
      {child1}
      {child2}
    </>
  )
}
```

Remarquez que cette approche ne fonctionne pas dans une boucle car les appels aux Hooks **ne doivent pas** être placés dans des boucles. Mais vous pouvez extraire la liste dans un composant à part, et appeler `useMemo` sur celui-ci.

Comment créer paresseusement des objets coûteux ?

`useMemo` vous permet de **mémoiser un calcul coûteux** si les dépendances sont les mêmes. Cependant, il n'est là que pour aider, et ne *garantit* pas que le calcul ne sera pas refait. Mais parfois vous devez vous assurer qu'un objet n'est créé qu'une seule fois.

Le principal cas d'utilisation concerne la création d'un état initial coûteux :

```
function Table(props) {
  // ⚠️ createRows() est appelée à chaque rendu
  const [rows, setRows] = useState(createRows(props.count));
  // ...
}
```

Pour éviter de recréer l'état initial ignoré, nous pouvons passer une **fonction** à `useState` :

```
function Table(props) {
  // ✅ createRows() n'est appelée qu'une seule fois
  const [rows, setRows] = useState(() => createRows(props.count));
  // ...
}
```

React n'appellera cette fonction que lors du premier rendu. Vous trouverez de plus amples détails dans l'[API de référence de `useState`](#).

Vous pouvez aussi vouloir occasionnellement éviter de recréer la valeur initiale de `useRef()`. Par exemple, vous voulez peut-être vous assurer qu'une instance de classe impérative n'est créée qu'une seule fois :

```
function Image(props) {
  // ⚠️ IntersectionObserver est créé à chaque rendu
  const ref = useRef(new IntersectionObserver(onIntersect));
  // ...
}
```

`useRef` **n'accepte pas** un argument de fonction spécial comme pour `useState`. Au lieu de ça, vous pouvez écrire votre propre fonction qui la crée et la définit paresseusement :

```

function Image(props) {
  const ref = useRef(null);

  // ✅ IntersectionObserver est créé paresseusement une seule fois
  function getObserver() {
    if (ref.current === null) {
      ref.current = new IntersectionObserver(onIntersect);
    }
    return ref.current;
  }

  // Quand vous en avez besoin, appelez getObserver()
  // ...
}

```

Ça permet d'éviter de recréer un objet coûteux tant qu'il n'est pas nécessaire. Si vous utilisez Flow ou TypeScript, vous pouvez aussi donner à `getObserver()` un type non-null pour un typage plus fin.

La création de fonctions à la volée pendant le rendu ralentit-elle les Hooks ?

Non. Dans les navigateurs modernes, les performances brutes des fermetures lexicales comparées à celles des classes diffèrent peu, sauf dans des scénarios extrêmes.

Par ailleurs, gardez à l'esprit que la conception de Hooks est plus efficace pour deux raisons :

- Les Hooks permettent d'éviter une grande partie de la « graisse » amenée par les classes, comme le coût de la création d'instances et la liaison des gestionnaires d'événements dans le constructeur.
- **Le code idiomatique utilisant des Hooks n'a pas besoin d'une imbrication profonde de composants** qui prévaudrait dans des projets utilisant des composants d'ordre supérieur, des props de rendu et des contextes. Avec des arborescences plus petites, React a moins de travail à faire.

Traditionnellement, les problématiques de performance associées aux fonctions définies à la volée en React sont liées au fait que passer de nouvelles fonctions de rappel à chaque rendu empêche les optimisations basées sur `shouldComponentUpdate` dans les composants enfants. Les Hooks abordent ce problème sous trois angles.

- Le Hook `useCallback` vous permet de référencer la même fonction de rappel d'un rendu à l'autre afin que `shouldComponentUpdate` puisse continuer à fonctionner :

```

// Ne changera pas sauf si `a` ou `b` change
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);

```

- Le Hook `useMemo` permet de contrôler plus facilement les mises à jour des enfants, réduisant le besoin de composants purs.
- Enfin, le Hook `useReducer` réduit le besoin de passer en profondeur des fonctions de rappel, comme expliqué ci-dessous.

Comment éviter de transmettre des fonctions de rappel ?

Nous nous sommes aperçus que la majorité des gens n'aiment pas passer des fonctions de rappel à travers chaque niveau de l'arborescence. Même si c'est plus explicite, ça fait franchement « plomberie ».

Dans de grandes arborescences de composants, nous conseillons plutôt de transmettre une fonction `dispatch` issue de `useReducer` via le contexte :

```
const TodosDispatch = React.createContext(null);

function TodosApp() {
  // Remarque : `dispatch` ne va pas changer d'un rendu à l'autre
  const [todos, dispatch] = useReducer(todosReducer);

  return (
    <TodosDispatch.Provider value={dispatch}>
      <DeepTree todos={todos} />
    </TodosDispatch.Provider>
  );
}
```

N'importe quel enfant de l'arborescence de `TodosApp` peut utiliser la fonction `dispatch` pour remonter des actions à `TodosApp` :

```
function DeepChild(props) {
  // Si nous voulons exécuter une action, nous pouvons récupérer dispatch depuis
  // le contexte.
  const dispatch = useContext(TodosDispatch);

  function handleClick() {
    dispatch({ type: 'add', text: 'salut' });
  }

  return (
    <button onClick={handleClick}>Ajouter une tâche</button>
  );
}
```

C'est à la fois plus pratique d'un point de vue maintenance (pas besoin de continuer à passer des fonctions de rappel), et ça règle au passage le problème (de mémoïsation) des fonctions de rappel. Pour les mises à jour déclenchées en profondeur, nous conseillons de transmettre `dispatch` de cette façon.

Remarquez que vous pouvez toujours choisir de transmettre l'*état* applicatif comme props (plus explicite) ou comme contexte (plus pratique pour les mises à jour très profondes). Si vous utilisez le contexte pour transmettre également l'*état* local, utilisez deux types de contexte différents : la donnée de contexte `dispatch` ne changera jamais, donc les composants qui l'utilisent n'ont pas besoin de se rafraîchir à moins qu'ils n'aient aussi besoin de l'*état* applicatif.

Comment lire une valeur changeant fréquemment avec `useCallback` ?

Remarque

Nous recommandons de `transmettre dispatch dans le contexte` plutôt que des fonctions de rappel individuelles dans les props. L'approche ci-dessous n'est mentionnée que par souci d'exhaustivité et à titre d'échappatoire.

Notez aussi que cette approche peut causer des problèmes avec le `mode concurrent`. Nous prévoyons de proposer des alternatives plus ergonomiques à l'avenir, mais la solution la plus sûre pour l'instant consiste à toujours invalider la fonction de rappel si des valeurs dont elle dépend changent.

Dans de rares cas vous pourriez avoir besoin de mémoiser une fonction de rappel avec `useCallback` mais la mémoisation ne fonctionne pas très bien parce que la fonction interne a tout de même trop souvent besoin d'être recréée. Si la fonction que vous mémoisez est un gestionnaire d'événements et n'est pas utilisée pendant le rendu, vous pouvez utiliser une `ref comme variable d'instance`, et y stocker manuellement la dernière valeur renvoyée :

```
function Form() {
  const [text, updateText] = useState('');
  const textRef = useRef();

  useLayoutEffect(() => {
    textRef.current = text; // Écrit dans la ref
  });

  const handleSubmit = useCallback(() => {
    const currentText = textRef.current; // Lit depuis la ref
    alert(currentText);
  }, [textRef]); // Ne recrée pas handleSubmit comme `[text]` le ferait

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}
```

C'est une approche un peu biscornue mais ça montre que vous pouvez faire cette optimisation en dernier ressort, si vous en avez vraiment besoin. Vous pouvez en masquer les détails dérangeants en l'extrayant dans un Hook personnalisé :

```
function Form() {
  const [text, updateText] = useState('');
  // Sera mémoisé même si `text` change :
  const handleSubmit = useEventCallback(() => {
    alert(text);
  }, [text]);

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}
```

```

function useEventCallback(fn, dependencies) {
  const ref = useRef() => {
    throw new Error('Cannot call an event handler while rendering.');
  };

  useEffect(() => {
    ref.current = fn;
  }, [fn, ...dependencies]);

  return useCallback(() => {
    const fn = ref.current;
    return fn();
  }, [ref]);
}

```

Dans les deux cas, nous **déconseillons cette approche** et ne l'illustrons ici que dans un souci d'exhaustivité. Au lieu de ça, il est préférable d'[éviter de transmettre des fonctions de rappel en profondeur](#).

Sous le capot

Comment React associe-t-il les appels de Hooks avec les composants ?

React garde trace du composant en cours de rendu. Grâce aux [règles des Hooks](#), nous savons que les Hooks sont uniquement appelés depuis des composants React (ou des Hooks personnalisés, qui sont aussi uniquement appelés depuis des composants React).

Il existe une liste interne de « cellules mémoire » associées à chaque composant. Ce sont juste des objets JavaScript où nous stockons quelques données. Quand vous appelez un Hook tel que `useState()`, il lit la cellule courante (ou l'initialise pendant le premier rendu), et déplace alors le pointeur sur la prochaine. C'est ainsi que de multiples appels à `useState()` peuvent avoir chacun un état local distinct.

Quelles sont les sources d'inspiration des Hooks ?

Les Hooks font la synthèse d'idées issues de plusieurs sources :

- Nos précédentes expériences autour d'API fonctionnelles dans le dépôt [react-future](#).
- Les expériences de la communauté React avec les API de props de rendu, notamment le [composant Reactions](#) de Ryan Florence.
- La proposition de mot-clé `adopt` de Dominic Gannaway comme sucre syntaxique pour les props de rendu.
- Les [variables d'état et les cellules à état en DisplayScript](#).
- Les [composants réducteurs](#) de ReasonReact.
- Les [abonnements](#) (*Subscriptions, NdT*) en Rx.
- Les [effets algébriques](#) de OCaml Multicore.

[Sebastian Markbåge](#) a proposé la conception initiale des Hooks, peaufinée ensuite par [Andrew Clark](#), [Sophie Alpert](#), [Dominic Gannaway](#) et d'autres membres de l'équipe React.

Introduction aux 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.

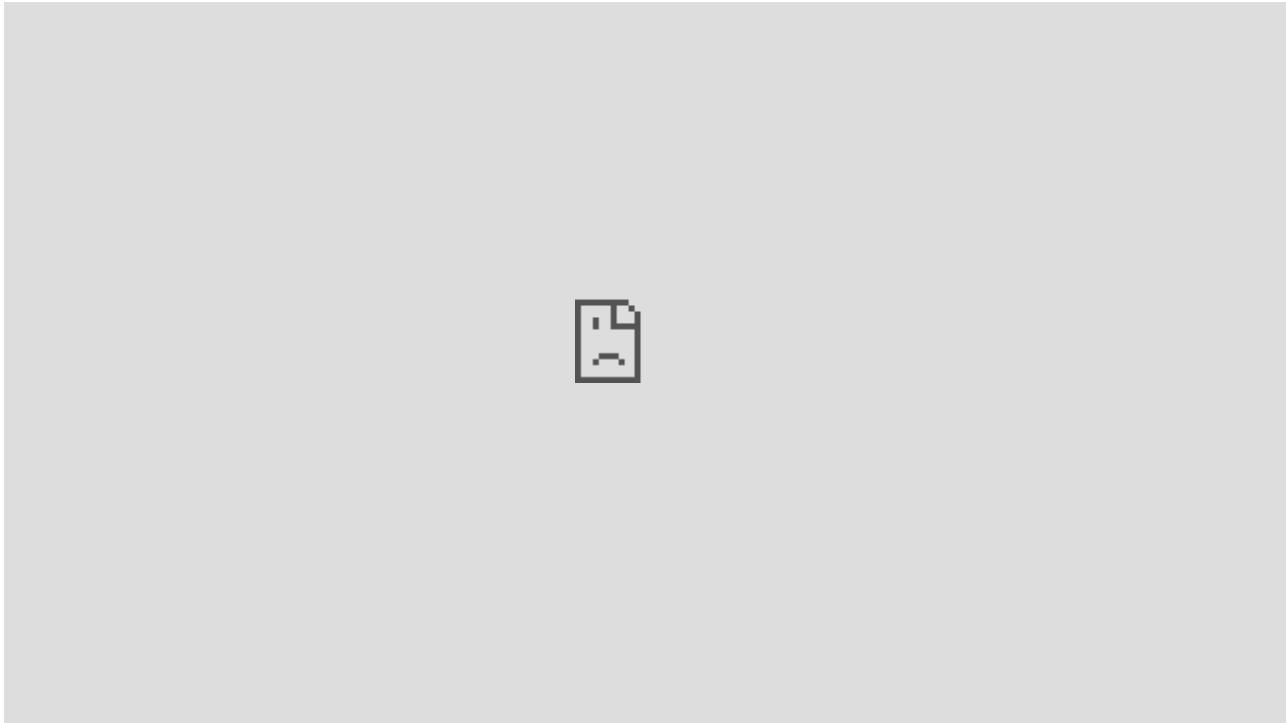
Remarque

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](#).

Vidéo de présentation

Lors de la React Conf 2018, Sophie Alpert et Dan Abramov ont présenté les Hooks, suivis de Ryan Florence qui a montré comment refactoriser une application pour les utiliser. Regardez la vidéo ici :



Pas de rupture de compatibilité

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

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

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

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.

Nous verrons tout ça plus en détail dans [Utiliser le Hook d'effet](#).

Les classes sont déroutantes pour les gens comme pour les 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 rechargement 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

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.

Nous 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, côté à côté avec les classes.

Questions fréquemment posées

Nous avons préparé une [FAQ des Hooks](#) qui répond aux questions les plus couramment posées sur les Hooks.

Prochaines étapes

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.**

[Go to TOC](#)

Aperçu 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.

Les Hooks sont [rétro-compatibles](#). Cette page fournit un survol des Hooks pour les utilisateurs expérimentés de React. C'est un tour d'horizon rapide. Si vous trouvez le contenu déroutant, cherchez un panneau jaune comme celui-ci :

Explication détaillée

Lisez les [raisons](#) qui nous ont fait ajouter les Hooks à React.

↑ ↑ ↑ **Chaque section se termine par un panneau jaune comme celui ci.** Ils pointent vers une documentation détaillée.

Hook d'état

Cet exemple affiche un compteur. Quand vous cliquez sur le bouton, la valeur augmente :

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

function Example() {
  // Déclaration d'une nouvelle variable d'état, que l'on appellera "count"
  const [count, setCount] = useState(0);

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

Dans le code ci-dessus, `useState` est un *Hook* (nous verrons ce que ça veut dire dans un instant). Nous l'appelons au sein d'une fonction composant pour y ajouter un état local. React va préserver cet état d'un affichage à l'autre. `useState` retourne une paire : la valeur de l'état *actuel* et une fonction qui vous permet de la mettre à jour. Vous pouvez appeler cette fonction depuis un gestionnaire d'événements, par exemple. Elle est similaire à `this.setState` dans une classe, à ceci près qu'elle ne fusionne pas l'ancien état et le nouveau. (Nous verrons un exemple de comparaison entre `useState` et `this.state` dans [Utiliser le Hook d'état](#).)

Le seul argument de `useState` est l'état initial. Dans l'exemple précédent, c'est `0` puisque notre compteur démarre à zéro. Remarquez que contrairement à `this.state`, ici l'état n'est pas nécessairement un objet, même si ça reste possible. L'argument d'état initial n'est utilisé que pour le premier affichage.

Déclarer plusieurs variables d'état

Vous pouvez utiliser le Hook d'état plus d'une fois dans un seul composant :

```
function ExampleWithManyStates() {
  // Déclaration de multiples variables d'état !
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banane');
  const [todos, setTodos] = useState([ { text: 'Apprendre les Hooks' } ]);
  // ...
}
```

La syntaxe de la [déstructuration positionnelle](#) nous permet de donner des noms distincts aux variables d'état que nous déclarons en appelant `useState`. Ces noms ne font pas partie de l'API `useState`. Au lieu de ça, React suppose que si vous appelez `useState` plusieurs fois, vous le faites dans le même ordre à chaque affichage. Nous reviendrons plus tard sur ce qui fait que ça fonctionne et les situations où c'est utile.

Mais qu'est ce qu'un Hook ?

Les Hooks sont des fonctions qui permettent de « se brancher » sur la gestion d'état local et de cycle de vie de React depuis des fonctions composants. Les Hooks ne fonctionnent pas dans des classes : ils vous permettent d'utiliser React sans classes. (Nous [ne recommandons pas](#) de ré-écrire vos composants existants du jour au lendemain, mais vous pouvez si vous le souhaitez commencer à utiliser les Hooks dans vos nouveaux composants.)

React fournit quelques Hooks prédéfinis comme `useState`. Vous pouvez aussi créer vos propres Hooks pour réutiliser un comportement à état dans différents composants. Dans un premier temps, nous allons aborder les Hooks prédéfinis.

Explication détaillée

Vous pouvez en apprendre davantage sur le Hook d'état sur la page dédiée : [Utiliser le Hook d'état](#).

⚡ Hook d'effet

Vous avez sûrement déjà réalisé un chargement de données distantes, des abonnements ou des modifications manuelles sur le DOM depuis un composant React. Nous appelons ces opérations « effets de bord » (ou effets pour faire court) parce qu'elles peuvent affecter d'autres composants et ne peuvent être réalisées pendant l'affichage.

Le Hook d'effet, `useEffect`, permet aux fonctions composants de gérer des effets de bord. Il joue le même rôle que `componentDidMount`, `componentDidUpdate`, et `componentWillUnmount` dans les classes React, mais au travers d'une API unique. (Nous verrons des exemples comparant `useEffect` à ces méthodes dans [Utiliser le Hook d'effet](#).)

Par exemple, ce composant change le titre du document après que React a mis à jour le DOM :

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

function Example() {
  const [count, setCount] = useState(0);

  // Équivalent à componentDidMount plus componentDidUpdate :
  useEffect(() => {
    // Mettre à jour le titre du document en utilisant 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>
  );
}
```

Lorsque vous appelez `useEffect`, vous dites à React de lancer votre fonction d'« effet » après qu'il a mis à jour le DOM. Les effets étant déclarés au sein du composant, ils ont accès à ses props et son état. Par défaut, React exécute les effets après chaque affichage, *y compris* le premier. (Nous aborderons plus en détails la comparaison avec le cycle de vie des classes dans [Utiliser le Hook d'effet](#).)

Les effets peuvent aussi préciser comment les « nettoyer » en renvoyant une fonction. Par exemple, ce composant utilise un effet pour s'abonner au statut de connexion d'un ami, et se nettoie en résiliant l'abonnement :

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Chargement...';
  }
  return isOnline ? 'En ligne' : 'Hors-ligne';
}
```

Dans cet exemple, React nous désabonnerait de notre `ChatAPI` quand le composant est démonté, mais aussi juste avant de relancer l'effet suite à un nouvel affichage. (Le cas échéant, vous pouvez [dire à React de sauter le ré-abonnement](#) si la `props.friend.id` passée à `ChatAPI` n'a pas changé.)

Tout comme avec `useState`, vous pouvez utiliser plus d'un seul effet dans un composant :

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `Vous avez cliqué ${count} fois`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  // ...
}
```

Les Hooks vous permettent d'organiser les effets de bord dans un composant en rassemblant leurs parties (telles que l'abonnement et le désabonnement), plutôt que de vous forcer à les répartir dans les méthodes de cycle de vie.

Explication détaillée

Vous pouvez en apprendre davantage sur `useEffect` sur la page dédiée : [Utiliser le Hook d'effet](#).

👉 Règles des Hooks

Les Hooks sont des fonctions JavaScript, mais ils imposent deux règles supplémentaires :

- Appelez les Hooks uniquement **au niveau racine**. Nappelez pas de Hooks à l'intérieur de boucles, de code conditionnel ou de fonctions imbriquées.
- Appelez les Hooks uniquement **dès depuis des fonctions composants React**. Nappelez pas les Hooks depuis des fonctions JavaScript classiques. (Il n'y a qu'un seul autre endroit où vous pouvez appeler des Hooks : vos propres Hook personnalisés. Nous en reparlerons dans un moment.)

Nous fournissons un [plugin ESLint](#) qui assure le respect de ces deux règles. Nous comprenons que ces règles peuvent sembler limitantes ou déroutantes au premier abord, mais elles sont essentielles pour que les Hooks fonctionnent correctement.

Explication détaillée

Vous pouvez en apprendre davantage sur ces règles sur la page dédiée : [Règles des Hooks](#).

Construire vos propres Hooks

Parfois, on veut réutiliser la même logique à état dans plusieurs composants. Traditionnellement, on avait deux solutions répandues à ce besoin : les [composants d'ordre supérieur](#) et les [props de rendu](#). Les Hooks personnalisés vous permettent de faire la même chose, mais sans ajouter de composants à votre arbre.

Un peu plus tôt sur cette page, nous avons présenté un composant `FriendStatus` qui appelle les Hooks `useState` et `useEffect` pour s'abonner à l'état de connexion d'un ami. Disons que l'on veut réutiliser cette logique d'abonnement dans un autre composant.

Tout d'abord, nous allons extraire cette logique dans un Hook personnalisé appelé `useFriendStatus` :

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

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });
  return isOnline;
}
```

Il prend `friendID` comme argument, et renvoie l'état de connexion de notre ami.

Nous pouvons désormais l'utiliser dans les deux composants :

```
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>
  );
}
```

Les états locaux de chaque composant sont complètement indépendants. Les Hooks permettent de réutiliser la *logique à état*, et non l'état lui-même. En fait, chaque *appel* à un Hook a un état complètement isolé ; vous pouvez même utiliser deux fois le même Hook personnalisé dans un même composant.

Les Hooks personnalisés sont plus une convention qu'une fonctionnalité. Si le nom d'une fonction commence par `use` et qu'elle appelle un autre Hook, nous dirons que c'est un Hook personnalisé. La convention de nommage `useSomething` permet à notre plugin ESLint de détecter les bugs dans un code utilisant les Hooks.

Vous pouvez écrire des Hooks personnalisés pour gérer un large éventail de cas d'usage tels que la gestion des formulaires, les animations, les abonnements déclaratifs, les horloges et probablement de nombreux autres auxquels nous n'avons pas pensé. Nous avons hâte de voir quels Hooks personnalisés la communauté React va inventer.

Explication détaillée

Vous pouvez en apprendre davantage sur les Hooks personnalisés sur une page dédiée : [Construire vos propres Hooks](#).

Autres Hooks

Il y a quelques Hooks prédéfinis plus rarement utilisés qui pourraient vous intéresser. Par exemple, `useContext` vous permet d'utiliser les Contextes de React sans imbrication superflue :

```
function Example() {
  const locale = useContext(LocaleContext);
  const theme = useContext(ThemeContext);
  // ...
}
```

Et `useReducer` vous permet de gérer l'état local de composants complexes avec un réducteur :

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer);
  // ...
}
```

Explication détaillée

Vous pouvez en apprendre davantage sur les Hooks prédéfinis sur une page dédiée : [Référence de l'API des Hooks](#).

Prochaines étapes

Eh bien, c'était rapide ! Si quelque chose n'était pas clair ou que vous souhaitez en savoir plus, vous pouvez consulter les prochaines pages, en débutant avec la documentation du [Hook d'état](#).

Vous pouvez aussi consulter la [référence de l'API des Hooks](#) et la [FAQ des Hooks](#).

Pour finir, pensez à lire [l'introduction aux Hooks](#), qui explique *pourquoi* nous avons ajouté les Hooks et comment nous allons commencer à les utiliser en parallèle des classes—sans ré-écrire nos applis.

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

`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 toute sécurité.

Mises à jour fonctionnelles

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 `setState` 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

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

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

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

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

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

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 toute sécurité.

Préciser l'état local initial

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

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

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 feriez 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 du 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 notifie 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

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());
```

Les règles des 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.

Les Hooks sont des fonctions JavaScript, mais vous devez suivre deux règles lorsque vous les utilisez. Nous mettons à votre disposition un [plugin de linter](#) pour vérifier ces règles automatiquement :

Appelez les Hooks uniquement au niveau racine

N'appelez pas de Hooks à l'intérieur de boucles, de code conditionnel ou de fonctions imbriquées. Au lieu de ça, utilisez seulement les Hooks au niveau racine de votre fonction React. En suivant cette règle, vous gardez que les Hooks sont appelés dans le même ordre à chaque affichage du composant. C'est ce qui permet à React de garantir le bon état des Hooks entre plusieurs appels à `useState` et `useEffect`. (Si vous êtes curieux-se, nous expliquerons ça en détail [plus bas](#).)

Appelez les Hooks uniquement depuis des fonctions React

N'appelez pas les Hooks depuis des fonctions JavaScript classiques. Vous pouvez en revanche :

- Appeler les Hooks depuis des fonctions composants React.
- Appeler les Hooks depuis des Hooks personnalisés (nous aborderons le sujet [dans la prochaine page](#)).

En suivant cette règle, vous gardez que toute la logique d'état d'un composant est clairement identifiable dans son code source.

Plugin ESLint

Nous avons publié un plugin ESLint appelé `eslint-plugin-react-hooks` qui assure le respect de ces deux règles. Vous pouvez ajouter ce plugin à votre projet si vous souhaitez l'utiliser :

Ce plugin est inclus par défaut dans Create React App [Create React App](#).

```
npm install eslint-plugin-react-hooks --save-dev
```

```
// Votre configuration ESLint
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Vérifie les règles des Hooks
    "react-hooks/exhaustive-deps": "warn"   // Vérifie les tableaux de dépendances
  }
}
```

Vous pouvez maintenant passer directement à la prochaine page, qui explique comment écrire vos propres Hooks. Dans cette page, nous allons maintenant expliquer le raisonnement qui se cache derrière ces règles.

Explications

Comme nous l'avons appris plus tôt, nous pouvons utiliser plusieurs Hooks *State* ou *Effect* au sein d'un même composant :

```
function Form() {
  // 1. Utilise la variable d'état name
  const [name, setName] = useState('Mary');

  // 2. Utilise un effet pour persister le formulaire
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. Utilise la variable d'état surname
  const [surname, setSurname] = useState('Poppins');

  // 4. Utilise un effet pour mettre à jour le titre
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}
```

Alors, comment React sait-il quel état correspond à quel appel à `useState` ? Réponse : **React se repose sur l'ordre dans lequel les Hooks sont appelés.** Notre exemple fonctionne car l'ordre d'appel des Hooks est le même à chaque affichage :

```
// -----
// Premier affichage
// -----
useState('Mary')           // 1. Initialise la variable d'état name avec 'Mary'
useEffect(persistForm)     // 2. Ajoute un effet pour persister le formulaire
useState('Poppins')        // 3. Initialise la variable d'état surname avec
'Poppins'                  // 4. Ajoute un effet pour mettre à jour le titre

// -----
// Deuxième affichage
// -----
useState('Mary')           // 1. Lit la variable d'état name (l'argument est
ignoré)                    // 2. Remplace l'effet pour persister le formulaire
useEffect(persistForm)     // 3. Lit la variable d'état surname (l'argument est
useState('Poppins')        // ignoré)
ignoré)                    // 4. Remplace l'effet pour mettre à jour le titre
useEffect(updateTitle)

// ...
```

Tant que l'ordre d'appel aux Hooks est le même d'un affichage à l'autre, React peut associer un état local à chacun d'entre eux. Mais que se passerait-il si nous appelions un Hook (par exemple, l'effet `persistForm`) de façon conditionnelle ?

```
// ⚡ Nous enfreignons la première règle en utilisant un Hook de façon
conditionnelle
if (name !== '') {
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });
}
```

La condition `name !== ''` est vraie au premier affichage, donc nous exécutons ce Hook. Cependant, lors du prochain affichage l'utilisateur risque de vider le formulaire, ce qui invalidera la condition. À présent que nous sautons ce Hook lors de l'affichage, l'ordre d'appel aux Hooks devient différent :

```
useState('Mary')           // 1. Lit la variable d'état name (l'argument est
ignoré)
// useEffect(persistForm) // ⚡ Ce Hook n'a pas été appelé !
useState('Poppins')       // ⚡ 2. (mais était 3). Échoue à lire la variable
d'état surname
useEffect(updateTitle)    // ⚡ 3. (mais était 4). Échoue à remplacer de l'effet
```

React ne saurait pas quoi renvoyer lors du second appel au Hook `useState`. React s'attendait à ce que le deuxième appel à un Hook dans ce composant corresponde à l'effet `persistForm`, comme lors de l'affichage précédent, mais ce n'est plus le cas. A partir de là, chaque appel à un Hook ultérieur à celui que nous avons sauté sera aussi décalé de un, provoquant des bugs.

C'est pourquoi les Hooks doivent être appelés au niveau racine de vos composants. Si vous voulez exécuter un effet de manière conditionnelle, vous pouvez mettre cette condition à *l'intérieur* de votre Hook :

```
useEffect(function persistForm() {
  // 👍 Nous n'enfreignons plus la première règle
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

Remarquez que vous n'avez pas à vous inquiéter de ça si vous utilisez le plugin de linter fourni. Mais maintenant, vous savez *pourquoi* les Hooks fonctionnent de cette manière et quels problèmes ces règles évitent.

Prochaines étapes

Enfin, nous sommes prêts pour apprendre à [écrire nos propres Hooks](#) ! Les Hooks personnalisés vous permettent de combiner les Hooks fournis par React pour créer vos propres abstractions et réutiliser une logique d'état commune dans différents composants.

Utiliser le Hook d'état

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

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

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 ?

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 découvrons dans les [Règles des Hooks](#).

Déclarer une variable d'état

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

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

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

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 ?

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

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](#).

Prochaines étapes

Dans cette section, vous avez appris à utiliser un des Hooks fournis par React, appelé `useState`. Nous y ferons parfois référence sous le terme « Hook d'état ». Il nous permet d'ajouter un état local à des fonctions composants React—ce qui est une première !

Nous en avons également appris un peu plus sur la nature des Hooks. Les Hooks sont des fonctions qui nous permettent de « nous brancher » sur des fonctionnalités React depuis des fonctions composants. Leur noms commencent toujours par `use`, et il y a encore beaucoup de Hooks que nous n'avons pas encore vus !

Continuons maintenant en découvrant le prochain Hook : `useEffect`. Il vous permet de déclencher des effets de bord dans les composants, ce qui est similaire aux méthodes de cycle de vie dans les classes.

Comment contribuer

React est l'un des premiers projets open source de Facebook qui est à la fois en développement intensif et utilisé en production sur les pages publiques de facebook.com. Nous travaillons encore à faire qu'il soit aussi facile et transparent que possible de contribuer à ce projet, et ce chantier n'est pas terminé. Mais avec un peu de chance, ce document éclaircira le processus de contribution et répondra à certaines des questions que vous pourriez avoir.

Code de conduite {#code-of-conduct}

Facebook a adopté le code de conduite [Contributor Covenant](#) et nous nous attendons à ce que les participant·e·s au projet y adhèrent. Veuillez lire [le texte complet](#) afin de comprendre quelles actions seront ou ne seront pas tolérées.

Développement ouvert

Tout travail sur React se passe directement sur [GitHub](#). Les membres de l'équipe noyau (*core team, NdT*) tout comme les contributeurs externes y envoient leur *pull requests*, lesquelles passent à travers le même processus de revue.

Gestion sémantique des versions

React utilise une [gestion sémantique de version](#). Nous publions des versions de correctifs pour les corrections de bugs critiques, des versions mineures pour les nouvelles fonctionnalités et les modifications non-essentielles, et des versions majeures s'il y a rupture de la compatibilité ascendante. Quand nous introduisons de telles ruptures, nous ajoutons aussi des avertissements de dépréciation dans une version mineure afin que nos utilisateur·rice·s puissent se familiariser avec les changements à venir et migrer leur code en amont. Vous pouvez en apprendre davantage sur notre engagement en termes de stabilité et de migration incrémentielle dans notre [politique de gestion des versions](#).

Toute modification substantielle est documentée dans le [journal des modifications](#).

Organisation des branches

Déposez toutes vos modifications directement sur la branche `master`. Nous n'utilisons pas de branches séparées pour le développement et pour les versions à venir. Nous faisons de notre mieux pour garder la branche `master` en bon état, avec des tests toujours au vert.

Le code qui atterrit dans `master` doit être compatible avec la dernière version stable. Il peut contenir de nouvelles fonctionnalités, mais pas de rupture de compatibilité ascendante. Nous devrions pouvoir sortir une version mineure à partir de la pointe de `master` à tout moment.

Fonctionnalités conditionnelles

Afin de conserver la branche `master` dans un état sortable, les ruptures de compatibilité ascendante et les fonctionnalités expérimentales doivent être protégées par un drapeau de fonctionnalité conditionnelle.

Ces drapeaux sont définis dans `packages/shared/ReactFeatureFlags.js`. D'une version de React à l'autre, des jeux de fonctionnalités conditionnelles distincts sont susceptibles d'être utilisés ; par exemple, React Native peut être configuré différemment de React DOM. Ces variations sont signalées dans `packages/shared/forks`. Les drapeaux de fonctionnalités conditionnelles sont généralement typés par Flow, de sorte que vous pouvez exécuter `yarn flow` pour confirmer que vous avez mis à jour tous les fichiers nécessaires.

Le système de construction de React retirera toutes les branches fonctionnelles désactivées avant publication. Une routine d'intégration continue est exécutée à chaque commit pour auditer les évolutions de la taille du *bundle*. Vous pouvez utiliser ces changements de taille comme confirmation que votre fonctionnalité a bien été périmétrée.

Bugs

Où trouver les problèmes connus

Nous utilisons les [GitHub Issues](#) pour les bugs publics. Nous les surveillons attentivement et essayons d'être transparents sur le développement en cours d'un correctif interne. Avant de créer un nouveau ticket, essayez de vérifier que votre problème n'a pas déjà été signalé.

Signaler de nouveaux problèmes

Le meilleur moyen d'obtenir un correctif pour votre problème consiste à en fournir une reproduction minimale. Cet [exemple JSFiddle](#) est un excellent point de départ.

Problèmes de sécurité

Facebook a un [programme de récompenses](#) pour la communication sécurisée de problèmes de sécurité. En conséquence, merci de ne pas créer de ticket public pour ça : suivez le processus expliqué sur la page du programme.

Comment nous contacter

- IRC : [#reactjs sur freenode](#)
- [Forums de discussion](#)

Au cas où vous auriez besoin d'aide avec React, il existe aussi [une communauté active d'utilisateurs de React sur la plate-forme de discussion Discord](#).

Proposer un changement

Si vous comptez proposer un changement de l'API publique ou faire un changement non trivial à l'implémentation, nous recommandons de [créer un ticket](#). Ça nous permettra de nous mettre d'accord sur votre proposition avant que vous n'y investissiez un effort trop important.

Si vous corrigez seulement un bug, il est tout à fait acceptable d'envoyer directement une *pull request*, mais nous conseillons tout de même de créer d'abord un ticket détaillant ce que vous corrigez. C'est utile pour le cas où nous n'accepterions pas ce correctif spécifique mais souhaiterions garder une trace du problème.

Votre première *pull request* {#your-first-pull-request}

Vous travaillez sur votre première *pull request* ? Vous pouvez apprendre comment faire ça au mieux grâce à cette série de vidéos gratuites (en anglais) :

[Comment contribuer à un projet open source sur GitHub](#)

Pour vous aider à vous jeter à l'eau et vous familiariser avec le processus de contribution, nous avons une liste de [bons premiers tickets](#) qui contient des bugs d'étendue relativement limitée. C'est un excellent point de départ.

Si vous décidez de corriger un bug, assurez-vous de vérifier le fil de commentaires au cas où quelqu'un serait déjà en train de travailler dessus. Si personne n'est dessus, veuillez laisser un commentaire indiquant que vous comptez vous y attaquer pour éviter que d'autres personnes ne dupliquent votre travail par accident.

Si quelqu'un dit travailler sur un correctif mais ne donne pas de nouvelle pendant deux semaines, vous pouvez prendre la relève, mais vous devriez tout de même laisser un commentaire dans ce sens.

Envoyer une *pull request* {#sending-a-pull-request}

L'équipe noyau surveille les *pull requests*. Nous ferons une revue de la vôtre et soit nous la fusionnerons, soit nous vous demanderons des ajustements, soit nous la fermerons en expliquant pourquoi. Pour les changements d'API, nous aurons peut-être besoin d'ajuster notre utilisation interne à facebook.com, ce qui pourrait retarder la fusion. Nous ferons cependant de notre mieux pour vous tenir informé·e tout au long du processus.

Avant d'envoyer une *pull request*, suivez attentivement ces instructions :

1. [Forkez le dépôt](#) et créez votre branche depuis `master`.
2. Lancez `yarn` à la racine du dépôt.
3. Si vous avez corrigé un bug ou ajouté du code qui devrait être testé, ajoutez les tests !
4. Assurez-vous que tous les tests passent (`yarn test`). Astuce : `yarn test --watch NomDuTest` est très utile en phase de développement.
5. Lancez `yarn test-prod` pour tester dans l'environnement de production. Cette commande accepte les mêmes options que `yarn test`.
6. Si vous avez besoin d'un débogueur, lancez `yarn debug-test --watch NomDuTest`, ouvrez `chrome://inspect`, et appuyez sur « Inspecter ».
7. Formattez votre code avec `prettier` (`yarn prettier`).
8. Assurez-vous que votre code passe la vérification du *linter* (`yarn lint`). Astuce : `yarn linc` ne vérifiera que les fichiers qui ont changé.
9. Lancez les vérifications de types `Flow` (`yarn flow`).
10. Si vous ne l'avez pas encore fait, remplissez le CLA (voir ci-dessous).

Accord de licence de contribution (CLA)

Afin que nous puissions accepter votre *pull request*, nous avons besoin que vous remplissiez un CLA (*Contributor License Agreement, NdT*). Vous n'avez besoin de faire ça qu'une seule fois, donc si vous l'avez déjà fait pour un autre projet open source de Facebook, tout va bien. Si vous envoyez une *pull request* pour la première fois, dites-nous simplement que vous avez déjà rempli le CLA et nous pourrons le vérifier sur base de votre identifiant GitHub.

[Remplissez votre CLA ici.](#)

Pré-requis pour contribuer

- Vous avez [Node](#) installé en v8.0.0+ et [Yarn](#) en v1.2.0+.
- Vous avez le [JDK](#) installé.
- Vous avez [gcc](#) installé ou êtes à l'aise avec le fait d'installer un compilateur si besoin. Certaines de nos dépendances peuvent nécessiter une étape de compilation. Sur OS X, les outils de ligne de commande XCode s'en occupent. Sur Ubuntu, [apt-get install build-essential](#) installera les paquets nécessaires. Des commandes similaires devraient fonctionner pour d'autres distributions Linux. Windows nécessite quelques étapes supplémentaires, consultez les [instructions d'installation de node-gyp](#) pour plus de détails.
- Vous êtes à l'aise avec Git

Workflow de développement

Après avoir cloné votre fork de React, lancez [yarn](#) afin d'aller chercher les dépendances du projet. Ensuite, vous pouvez lancer différentes commandes :

- [yarn lint](#) pour vérifier le style du code.
- [yarn linc](#) fonctionne comme [yarn lint](#) mais va plus vite car elle ne vérifie que les fichiers qui ont changé sur votre branche.
- [yarn test](#) lance la suite de tests complète.
- [yarn test --watch](#) lance un superviseur interactif de tests.
- [yarn test <motif>](#) lance les tests des fichiers dont le nom correspond au motif.
- [yarn test-prod](#) lance les tests dans l'environnement de production. Elle accepte toutes les mêmes options que [yarn test](#).
- [yarn debug-test](#) fonctionne exactement comme [yarn test](#) mais avec un débogueur. Ouvrez [chrome://inspect](#) et appuyez sur « Inspecter ».
- [yarn flow](#) lance les vérifications de types [Flow](#).
- [yarn build](#) crée un dossier [build](#) avec tous les modules.
- [yarn build react/index,react-dom/index --type=UMD](#) crée des *builds* UMD seulement des modules indiqués, ici React et ReactDOM.

Nous recommandons d'utiliser [yarn test](#) (ou ses variations mentionnées ci-dessus) pour vous assurer de ne pas introduire de régressions en travaillant sur votre contribution. Cependant, il peut être pratique d'essayer votre build de React dans un vrai projet.

Tout d'abord, lancez `yarn build`. Ça produira des *bundles* pré-compilés dans le dossier `build`, et préparera les modules npm dans `build/packages`.

La manière la plus simple d'essayer vos modifications consiste à lancer `yarn build react/index,react-dom/index --type=UMD` et ensuite ouvrir `fixtures/packaging/babel-standalone/dev.html`. Ce fichier utilise déjà `react.development.js` depuis le dossier `build`, donc il utilisera vos évolutions.

Si vous voulez essayer vos évolutions dans votre projet React existant, vous pouvez copier `build/dist/react.development.js`, `build/dist/react-dom.development.js`, ou tout autre produit de la compilation dans votre appli et les utiliser au lieu de la version stable.

Si votre projet utilise React via npm, vous pouvez supprimer `react` et `react-dom` dans ses dépendances et utiliser `yarn link` pour les faire pointer vers votre dossier local `build`. Remarquez qu'**au lieu de `--type=UMD` vous voudrez plutôt passer `--type=NODE` à la construction**. Vous aurez aussi besoin du module `scheduler` :

```
cd ~/chemin_vers_votre_clone_de_react/
yarn build react/index,react-dom/index,scheduler --type=NODE

cd build/node_modules/react
yarn link
cd build/node_modules/react-dom
yarn link

cd ~/chemin/vers/votre/projet
yarn link react react-dom
```

Chaque fois que vous lancez `yarn build` dans le dossier de React, les versions mises à jour apparaîtront dans le dossier `node_modules` de votre projet. Vous pouvez alors recompiler votre projet pour essayer vos modifications.

Si un module reste manquant (par ex. peut-être utilisez-vous `react-dom/server` dans votre projet), vous pouvez toujours faire une construction intégrale avec `yarn build`. Gardez à l'esprit que l'exécution de `yarn build` sans options prend beaucoup de temps.

Nous exigeons tout de même que votre *pull request* contienne des tests unitaires pour chaque nouvelle fonctionnalité. Ainsi nous pouvons nous assurer de ne pas casser votre code par la suite.

Guide de style

Nous utilisons un outil de formatage automatique appelé [Prettier](#). Lancez `yarn prettier` après avoir changé le code.

Ensuite, notre *linter* repérera la plupart des problèmes qui pourraient exister dans votre code. Vous pouvez vérifier l'état du style de votre code simplement en lançant `yarn linc`.

Cependant, il y a toujours certains styles que le *linter* ne peut pas remarquer. Si vous n'êtes pas sûr·e de votre coup, laissez-vous guider par le [Guide de style de Airbnb](#).

Appels à commentaires (RFC)

Beaucoup de modifications, y compris les correctifs de bugs et les améliorations de la documentation, peuvent être implémentés et revus via le workflow normal de *pull requests* sur GitHub.

Certaines évolutions sont cependant plus « substantielles », et nous demandons à ce que celles-ci passent par une petite phase de conception afin d'obtenir un consensus au sein de l'équipe noyau de React.

Le processus de « RFC » (*Request For Comments, NdT*) a pour but de fournir un chemin contrôlé et cohérent pour l'introduction de nouvelles fonctionnalités dans le projet. Vous pouvez apporter votre contribution en consultant le [dépôt des RFC](#).

Licence

En contribuant à React, vous acceptez que toute contribution que vous apportez soit licenciée sous la licence MIT.

Et maintenant ?

Lisez la [page suivante](#) pour apprendre comment la base de code est organisée.

[Go to TOC](#)

Notes d'implémentation

Cette section fournit un ensemble de notes relatives à l'implémentation du [réconciliateur "stack"](#).

C'est très technique et suppose une solide compréhension de l'API publique de React ainsi que de sa structure divisée en noyau, moteurs de rendu et réconciliateur. Si vous ne connaissez pas bien la base de code React, lisez d'abord l'[aperçu du code source](#).

Ça suppose également de comprendre les [différences entre les composants React, leurs instances et leurs éléments](#).

Le réconciliateur "stack" était utilisé jusqu'à React 15 inclus. Il se trouve dans [src/renderers/shared/stack/reconciler](#).

Vidéo : construire React à partir de zéro

[Paul O'Shannessy](#) a donné une conférence (en anglais) sur la [construction de React à partir de zéro](#) qui a largement inspiré ce document.

Ce document et sa conférence sont des simplifications de la véritable base de code : en vous familiarisant avec ces deux sources, vous devriez mieux comprendre.

Aperçu

Le réconciliateur lui-même n'a pas d'API publique. Les [moteurs de rendu](#) comme React DOM et React Native l'utilisent pour mettre à jour efficacement l'interface utilisateur en fonction des composants React écrits par l'utilisateur.

Le montage, un processus récursif

Intéressons-nous à la première fois que vous montez un composant :

```
ReactDOM.render(<App />, rootEl);
```

React DOM passera `<App />` au réconciliateur. Rappelez-vous que `<App />` est un élément React, c'est-à-dire une description de *quoi* afficher. Vous pouvez le considérer comme un simple objet :

```
console.log(<App />);
// { type: App, props: {} }
```

Le réconciliateur vérifiera si `App` est une classe ou une fonction.

Si `App` est une fonction, le réconciliateur appellera `App(props)` pour obtenir le rendu de l'élément associé.

Si `App` est une classe, le réconciliateur instanciera une `App` avec `new App(props)`, appellera la méthode de cycle de vie `componentWillMount()`, puis appellera la méthode `render()` pour obtenir le rendu de l'élément.

Dans les deux cas, le réconciliateur saura quel élément « a été produit par le rendu » de `App`.

Ce processus est récursif. Le rendu de `App` peut produire un `<Greeting />`, celui de `Greeting` peut produire un `<Button />`, et ainsi de suite. Le réconciliateur « creusera » récursivement dans les composants définis par l'utilisateur, et saura ainsi ce que produit le rendu de chacun.

Vous pouvez imaginer ce processus comme un pseudo-code :

```

function isClass(type) {
  // Les sous-classes de React.Component ont ce drapeau
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// Cette fonction prend en paramètre un élément React (par exemple <App />)
// et renvoie un nœud DOM ou natif représentant l'arbre monté.
function mount(element) {
  var type = element.type;
  var props = element.props;

  // Nous déterminerons l'élément rendu
  // soit en exécutant le type comme une fonction
  // soit en créant une instance puis en appelant render().
  var renderedElement;
  if (isClass(type)) {
    // Composant basé classe
    var publicInstance = new type(props);
    // Initialise les props
    publicInstance.props = props;
    // Appelle le cycle de vie si besoin
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
    // Obtient l'élément rendu en appelant render()
    renderedElement = publicInstance.render();
  } else {
    // Fonction composant
    renderedElement = type(props);
  }

  // Ce processus est récursif parce qu'un composant peut
  // renvoyer un élément avec un autre type de composant.
  return mount(renderedElement);

  // Remarque : cette implémentation est incomplète et la récursivité est
  // infinie !
  // Ça gère uniquement les éléments comme <App /> ou <Button />.
  // Ça ne gère pas pour l'instant les éléments comme <div /> ou <p />.
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

Remarque

C'est vraiment du pseudo-code. L'implémentation actuelle diffère pas mal. Ce code saturerait la pile car nous n'avons pas précisé de condition d'arrêt pour la récursivité.

Récapitulons quelques idées clés dans l'exemple ci-dessus :

- Les éléments React sont des objets simples représentant le type du composant (par exemple `App`) et les props.
- Les composants définis par l'utilisateur (par exemple `App`) peuvent être des classes ou des fonctions mais toutes « font un rendu » avec des éléments comme résultat.
- Le « montage » est un processus récursif qui crée un arbre DOM ou natif à partir de l'élément racine React (par exemple `<App />`).

Montage d'éléments hôtes

Ce processus serait inutile si nous n'affichions pas quelque chose à l'écran au final.

En plus des composants définis par l'utilisateur (« composites »), les éléments React peuvent également représenter des composants pour des plates-formes spécifiques (« hôtes »). Par exemple, la méthode de rendu de `Button` pourrait renvoyer une `<div />`.

Si la propriété `type` de l'élément est une chaîne de caractères, il s'agit d'un élément hôte :

```
console.log(<div />);
// { type: 'div', props: {} }
```

Il n'y a aucun code défini par l'utilisateur associé aux éléments hôtes.

Lorsque le réconciliateur rencontre un élément hôte, il laisse le moteur de rendu s'occuper du montage. Par exemple, React DOM créerait un nœud DOM.

Si l'élément hôte a des enfants, le réconciliateur les monte de manière récursive en suivant le même algorithme que celui décrit plus haut. Peu importe que les enfants soient hôtes (comme `<div><hr /></div>`), composites (comme `<div><Button /></div>`) ou un mélange des deux.

Les nœuds DOM produits par les composants enfants seront ajoutés au nœud DOM parent, et donc de manière récursive, l'ensemble de la structure DOM sera constituée.

Remarque

Le réconciliateur lui-même n'est pas lié au DOM. Le résultat exact du montage (parfois appelé "mount image" dans le code source) dépend du moteur de rendu, et donc peut être un nœud DOM (React DOM), une chaîne de caractères (React DOM Server) ou un nombre représentant une vue native (React Native).

Si nous devions étendre le code pour gérer des éléments hôtes, il ressemblerait à ceci :

```

function isClass(type) {
  // Les sous-classes de React.Component ont ce drapeau
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// Cette fonction gère uniquement les éléments avec un type composite.
// Par exemple, elle gère <App /> et <Button />, mais pas une <div />.
function mountComposite(element) {
  var type = element.type;
  var props = element.props;

  var renderedElement;
  if (isClass(type)) {
    // Composant basé classe
    var publicInstance = new type(props);
    // Initialise les props
    publicInstance.props = props;
    // Appelle le cycle de vie si besoin
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
    renderedElement = publicInstance.render();
  } else if (typeof type === 'function') {
    // Fonction composant
    renderedElement = type(props);
  }

  // C'est récursif mais nous atteindrons finalement le bas de la récursion
  lorsque
  // l'élément sera hôte (par exemple <div />) au lieu de composite (par exemple
  <App />):
  return mount(renderedElement);
}

// Cette fonction gère uniquement les éléments avec un type hôte.
// Par exemple, il gère <div /> et <p />, mais pas un <App />.
function mountHost(element) {
  var type = element.type;
  var props = element.props;
  var children = props.children || [];
  if (!Array.isArray(children)) {
    children = [children];
  }
  children = children.filter(Boolean);

  // Ce bloc de code ne devrait pas être dans le réconciliateur.
  // Les différents moteurs de rendu peuvent initialiser les nœuds différemment.
  // Par exemple, React Native crée des vues iOS ou Android.
  var node = document.createElement(type);
  Object.keys(props).forEach(propName => {
    if (propName !== 'children') {
      node.setAttribute(propName, props[propName]);
    }
  });

  // Monte les enfants
  children.forEach(childElement => {
    // L'enfant peut être hôte (par exemple <div />) ou composite (par exemple
    <Button />).
  });
}

```

```
// Nous les monterons également de manière récursive :
var childNode = mount(childElement);

// Cette ligne de code est également spécifique au moteur de rendu.
// Ce serait différent en fonction du moteur de rendu :
node.appendChild(childNode);
};

// Renvoie le nœud DOM comme résultat de montage.
// C'est ici que se termine la récursion.
return node;
}

function mount(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // Composants définis par l'utilisateur
    return mountComposite(element);
  } else if (typeof type === 'string') {
    // Composants spécifiques aux plates-formes
    return mountHost(element);
  }
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);
```

Ça fonctionne mais ça reste loin de la manière dont le réconciliateur est véritablement implémenté. L'ingrédient clé manquant est la prise en charge des mises à jour.

Voici venir les instances internes

La principale caractéristique de React est que vous pouvez refaire tout le rendu sans recréer le DOM ni réinitialiser l'état :

```
ReactDOM.render(<App />, rootEl);
// Devrait réutiliser le DOM existant :
ReactDOM.render(<App />, rootEl);
```

Cependant, notre implémentation ci-dessus sait uniquement monter l'arbre initial. Elle ne peut pas effectuer de mises à jour dans l'arborescence car elle ne stocke pas toutes les informations nécessaires, telles que toutes les références `publicInstance` ou les nœuds DOM qui correspondent aux composants.

La base de code du réconciliateur "stack" résout ce problème en faisant de la fonction `mount()` une méthode au sein d'une classe. Cette approche présente des inconvénients et nous allons dans la direction opposée pour la [réécriture en cours du réconciliateur](#). Cependant, voici comment ça fonctionne dans le réconciliateur historique.

Plutôt que deux fonctions distinctes `mountHost` et `mountComposite`, nous créerons deux classes : `DOMComponent` et `CompositeComponent`.

Les deux classes ont un constructeur acceptant `element`, ainsi qu'une méthode `mount()` qui renvoie le nœud monté. Nous remplacerons la fonction racine `mount()` par une fabrique (*factory*, *NdT*) qui instanciera la bonne classe :

```

function instantiateComponent(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // Composants définis par l'utilisateur
    return new CompositeComponent(element);
  } else if (typeof type === 'string') {
    // Composants spécifiques aux plates-formes
    return new DOMComponent(element);
  }
}

```

Tout d'abord, examinons l'implémentation de `CompositeComponent` :

```

class CompositeComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedComponent = null;
    this.publicInstance = null;
  }

  getPublicInstance() {
    // Pour les composants composites, exposons l'instance de la classe.
    return this.publicInstance;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;

    var publicInstance;
    var renderedElement;
    if (isClass(type)) {
      // Composant basé classe
      publicInstance = new type(props);
      // Initialise les props
      publicInstance.props = props;
      // Appelle le cycle de vie si besoin
      if (publicInstance.componentWillMount) {
        publicInstance.componentWillMount();
      }
      renderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Fonction composant
      publicInstance = null;
      renderedElement = type(props);
    }

    // Sauvegarde l'instance publique
    this.publicInstance = publicInstance;

    // Construit l'instance interne de l'enfant en fonction de l'élément.
    // Ce sera un DOMComponent pour <div /> ou <p />,
    // et un CompositeComponent pour <App /> ou <Button /> :
    var renderedComponent = instantiateComponent(renderedElement);
    this.renderedComponent = renderedComponent;

    // Monte le résultat du rendu
    return renderedComponent.mount();
  }
}

```

Ce n'est pas très différent de notre implémentation précédente de `mountComposite()`, mais maintenant, nous pouvons sauver certaines informations, telles que `this.currentElement`, `this.renderedComponent` et `this.publicInstance`, pour les utiliser lors des mises à jour d'éléments.

Remarquez qu'une instance de `CompositeComponent` n'est pas la même chose qu'une instance du `element.type` fourni par l'utilisateur. `CompositeComponent` est un détail d'implémentation de notre réconciliateur et n'est jamais exposé à l'utilisateur. La classe définie par l'utilisateur est celle que nous avons lue dans `element.type`, et `CompositeComponent` crée une instance de celle-ci.

Pour éviter toute confusion, nous appellerons les instances de `CompositeComponent` et `DOMComponent` des « instances internes ». Elles existent pour nous permettre de leur associer des données durables. Seuls le moteur de rendu et le réconciliateur s'en servent.

En revanche, nous appelons une instance de classe définie par l'utilisateur « instance publique ». L'instance publique, c'est ce que vous voyez comme `this` dans `render()` et les autres méthodes de vos composants personnalisés.

La fonction `mountHost()` a été refactorisée pour devenir la méthode `mount()` dans la classe `DOMComponent`, et son code a des airs familiers :

```
class DOMComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedChildren = [];
    this.node = null;
  }

  getPublicInstance() {
    // Pour les composants DOM, expose uniquement le nœud du DOM.
    return this.node;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;
    var children = props.children || [];
    if (!Array.isArray(children)) {
      children = [children];
    }

    // Crée et sauvegarde le nœud
    var node = document.createElement(type);
    this.node = node;

    // Initialise les attributs
    Object.keys(props).forEach(propName => {
      if (propName !== 'children') {
        node.setAttribute(propName, props[propName]);
      }
    });

    // Crée et sauvegarde les enfants contenus.
    // Chacun d'eux peut être un DOMComponent ou un CompositeComponent,
    // selon que le type de l'élément est une chaîne de caractères ou une
    // fonction.
  }
}
```

```

var renderedChildren = children.map(instantiateComponent);
this.renderedChildren = renderedChildren;

// Collecte les nœuds DOM qui sont renvoyés lors du montage
var childNodes = renderedChildren.map(child => child.mount());
childNodes.forEach(childNode => node.appendChild(childNode));

// Renvoie le nœud DOM comme résultat du montage
return node;
}
}

```

Après la refactorisation de `mountHost()`, la différence principale est que nous conservons `this.node` et `this.renderedChildren`, qui sont maintenant associés à l'instance interne du composant DOM. Nous les utiliserons également à l'avenir pour appliquer des mises à jour non destructives.

Par conséquent, chaque instance interne, composite ou hôte, pointe maintenant vers ses instances internes enfants. Pour vous aider à le visualiser, si une fonction composant `<App>` donne un composant basé classe `<Button>`, et qu'un `Button` donne une `<div>`, l'arbre des instances internes ressemblera à ceci :

```

[object CompositeComponent] {
  currentElement: <App />,
  publicInstance: null,
  renderedComponent: [object CompositeComponent] {
    currentElement: <Button />,
    publicInstance: [object Button],
    renderedComponent: [object DOMComponent] {
      currentElement: <div />,
      node: [object HTMLDivElement],
      renderedChildren: []
    }
  }
}

```

Dans le DOM, vous verriez uniquement la `<div>`. Pourtant, l'arbre d'instances internes contient tant les instances internes composites que celles hôtes.

Les instances internes composites ont besoin de stocker :

- L'élément actuel.
- L'instance publique si le type de l'élément est une classe.
- L'instance interne unique produite par le rendu. Il peut s'agir d'un `DOMComponent` ou d'un `CompositeComponent`.

Les instances internes hôtes ont besoin de stocker :

- L'élément actuel.
- Le nœud DOM.
- Toutes les instances internes enfants. Chacune d'elles peut être soit un `DOMComponent` soit un `CompositeComponent`.

Si vous avez du mal à imaginer la structure d'un arbre d'instances internes dans des applications plus complexes, [React DevTools](#) peut vous donner une bonne approximation, dans la mesure où il signale les instances hôtes en gris, et les instances composites en violet :

```

<TodoApp>
  ▼<div>
    <h3>TODO</h3>
    ▼<TodoList items=[{...}]>
      ▼<ul>
        <li key="1475260226151">Buy milk</li>
      </ul>
    </TodoList>
    ▼<form onSubmit=bound handleSubmit()>
      <input onChange=bound onChange() value="" />
      <button>Add #2</button>
    </form>
  </div>
</TodoApp>

```

Pour terminer cette refactorisation, nous allons introduire une fonction qui monte une arbre complet dans un nœud conteneur, tout comme `ReactDOM.render()`. Elle renvoie une instance publique, là aussi comme `ReactDOM.render()` :

```

function mountTree(element, containerNode) {
  // Crée une instance interne racine
  var rootComponent = instantiateComponent(element);

  // Monte le composant racine dans le conteneur
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Renvoie l'instance publique fournie
  var publicInstance = rootComponent.getPublicInstance();
  return publicInstance;
}

var rootEl = document.getElementById('root');
mountTree(<App />, rootEl);

```

Démontage

Maintenant que nous avons des instances internes qui conservent leurs enfants et les nœuds DOM, nous pouvons implémenter le démontage. Pour un composant composite, le démontage appelle une méthode de cycle de vie et opère récursivement.

```

class CompositeComponent {
  // ...

  unmount() {
    // Appelle la méthode de cycle de vie si besoin
    var publicInstance = this.publicInstance;
    if (publicInstance) {
      if (publicInstance.componentWillUnmount) {

```

```

        publicInstance.componentWillUnmount();
    }

    // Démonte l'unique composant affiché
    var renderedComponent = this.renderedComponent;
    renderedComponent.unmount();
}

}

```

Pour `DOMComponent`, le démontage appelle celui de chaque enfant :

```

class DOMComponent {

    // ...

    unmount() {
        // Démonte tous les éléments enfants
        var renderedChildren = this.renderedChildren;
        renderedChildren.forEach(child => child.unmount());
    }
}

```

En pratique, le démontage des composants DOM enlève également les écouteurs d'événements et efface certains caches, mais nous ignorerons ces détails.

Nous pouvons maintenant ajouter une nouvelle fonction racine appelée `unmountTree(containerNode)`, semblable à `ReactDOM.unmountComponentAtNode()` :

```

function unmountTree(containerNode) {
    // Lit l'instance interne depuis un nœud DOM :
    // (Ça ne fonctionne pas encore, il faudra changer mountTree() pour le stocker.)
    var node = containerNode.firstChild;
    var rootComponent = node._internalInstance;

    // Démonte l'arbre et efface le conteneur
    rootComponent.unmount();
    containerNode.innerHTML = '';
}

```

Pour que ça fonctionne, nous devons lire une instance interne racine à partir d'un nœud DOM. Nous modifierons `mountTree()` pour ajouter la propriété `_internalInstance` au nœud DOM racine. Nous apprendrons aussi à `mountTree()` à détruire n'importe quel arbre existant, donc nous pourrons l'appeler plusieurs fois :

```

function mountTree(element, containerNode) {
    // Détruit un éventuel arbre existant
    if (containerNode.firstChild) {
        unmountTree(containerNode);
    }

    // Crée l'instance interne racine
    var rootComponent = instantiateComponent(element);

    // Monte le composant racine dans un conteneur
    var node = rootComponent.mount();
    containerNode.appendChild(node);

    // Sauvegarde une référence à l'instance interne
}

```

```

node._internalInstance = rootComponent;

// Renvoie l'instance publique fournie
var publicInstance = rootComponent.getPublicInstance();
return publicInstance;
}

```

À présent, l'exécution répétée de `unmountTree()` ou de `mountTree()` supprime l'ancien arbre et exécute la méthode de cycle de vie `componentWillUnmount()` sur les composants.

Mise à jour

Dans la section précédente, nous avons implémenté le démontage. Cependant, React ne serait pas très utile si à chaque modification de prop, l'arbre entier était démonté puis remonté. Le but du réconciliateur est de réutiliser dans la mesure du possible les instances existantes afin de préserver le DOM et l'état local :

```

var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Devrait réutiliser le DOM existant :
mountTree(<App />, rootEl);

```

Nous allons ajouter une méthode supplémentaire au contrat de nos instances internes. En plus de `mount()` et `unmount()`, `DOMComponent` et `CompositeComponent` implémenteront une nouvelle méthode appelée `receive(nextElement)` :

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

class DOMComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

```

Son travail consiste à faire tout le nécessaire pour mettre à jour le composant (et ses enfants) avec la description fournie par `nextElement`.

C'est la partie qui est souvent décrite comme la « comparaison de DOM virtuel » (*virtual DOM diffing, NdT*), même si ça consiste en réalité à parcourir récursivement l'arbre interne pour permettre à chaque instance interne de recevoir une mise à jour.

Mise à jour des composants composites

Lorsqu'un composant composite reçoit un nouvel élément, nous exécutons la méthode de cycle de vie `componentWillUpdate()`.

Ensuite nous rafraîchissons le composant avec les nouvelles props et récupérons le prochain élément issu du rendu :

```
class CompositeComponent {
  // ...

  receive(nextElement) {
    var prevProps = this.currentElement.props;
    var publicInstance = this.publicInstance;
    var prevRenderedComponent = this.renderedComponent;
    var prevRenderedElement = prevRenderedComponent.currentElement;

    // Met à jour l'élément *associé*
    this.currentElement = nextElement;
    var type = nextElement.type;
    var nextProps = nextElement.props;

    // Détermine quel est le prochain résultat de render()
    var nextRenderedElement;
    if (isClass(type)) {
      // Composant basé classe
      // Appelle le cycle de vie si besoin
      if (publicInstance.componentWillUpdate) {
        publicInstance.componentWillUpdate(nextProps);
      }
      // Met à jour les props
      publicInstance.props = nextProps;
      // Rafraîchit
      nextRenderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Fonction composant
      nextRenderedElement = type(nextProps);
    }
  }

  // ...
}
```

Ensute, nous pouvons regarder le `type` de l'élément produit. Si le `type` n'a pas changé depuis le dernier rendu, le composant sous-jacent peut être mis à jour directement.

Par exemple, s'il renvoie `<Button color="red" />` la première fois et `<Button color="blue" />` la seconde fois, nous pouvons simplement dire à l'instance interne correspondante de recevoir (`receive()`) l'élément mis à jour :

```
// ...

// Si le type de l'élément affiché n'a pas changé,
// réutilise l'instance du composant existant et sort.
if (prevRenderedElement.type === nextRenderedElement.type) {
  prevRenderedComponent.receive(nextRenderedElement);
  return;
}

// ...
```

Cependant, si l'élément mis à jour a un `type` différent de celui affiché jusqu'ici, nous ne pouvons pas mettre à jour l'instance interne. Un `<button>` ne peut pas simplement « devenir » un `<input>`.

Dans ce cas, nous devons démonter l'instance interne existante et monter la nouvelle correspondant au type du nouvel élément. Par exemple, voici ce qui se produit lorsqu'un composant ayant précédemment affiché un `<button />` affiche désormais un `<input />` :

```
// ...

// Si nous arrivons jusqu'ici, nous devons démonter le composant
// précédemment monté, monter le nouveau et échanger leurs nœuds.

// Récupère l'ancien nœud car il devra être remplacé
var prevNode = prevRenderedComponent.getHostNode();

// Démonte l'ancien élément enfant et en monte un nouveau
prevRenderedComponent.unmount();
var nextRenderedComponent = instantiateComponent(nextRenderedElement);
var nextNode = nextRenderedComponent.mount();

// Remplace la référence à l'enfant
this.renderedComponent = nextRenderedComponent;

// Remplace l'ancien nœud par le nouveau.
// Remarque : il s'agit d'un code spécifique au moteur de rendu et
// idéalement il devrait vivre en dehors de CompositeComponent :
prevNode.parentNode.replaceChild(nextNode, prevNode);
}

}
```

En résumé, lorsqu'un composant composite reçoit un nouvel élément, il peut soit déléguer la mise à jour à son instance interne issue du rendu précédent, soit la démonter et en monter une nouvelle en remplacement.

Il existe un autre cas dans lequel un composant va être remonté plutôt que de recevoir un élément : lorsque la `key` de cet élément a été modifiée. Nous ne voyons pas la gestion des `key` dans ce document, car elle ajoute une complexité supplémentaire à un tutoriel déjà venu.

Remarquez que nous devons encore ajouter une méthode appelée `getHostNode()` à l'instance interne afin qu'il soit possible de localiser le nœud spécifique à la plate-forme et de le remplacer lors de la mise à jour. Son implémentation est simple pour les deux classes :

```
class CompositeComponent {
// ...

getHostNode() {
    // Demande au composant issu du rendu de le fournir.
    // Ça analysera récursivement tous les composites.
    return this.renderedComponent.getHostNode();
}
}

class DOMComponent {
// ...

getHostNode() {
    return this.node;
}
}
```

Mise à jour des composants hôtes

Les implémentations de composant hôte, comme `DOMComponent`, font leurs mises à jour différemment. À la réception d'un élément, ils ont besoin de mettre à jour la vue spécifique à la plate-forme sous-jacente. Dans le cas de React DOM, ça implique de mettre à jour les attributs DOM :

```
class DOMComponent {
  // ...

  receive(nextElement) {
    var node = this.node;
    var prevElement = this.currentElement;
    var prevProps = prevElement.props;
    var nextProps = nextElement.props;
    this.currentElement = nextElement;

    // Supprime les anciens attributs.
    Object.keys(prevProps).forEach(propName => {
      if (propName !== 'children' && !nextProps.hasOwnProperty(propName)) {
        node.removeAttribute(propName);
      }
    });
    // Initialise les nouveaux attributs.
    Object.keys(nextProps).forEach(propName => {
      if (propName !== 'children') {
        node.setAttribute(propName, nextProps[propName]);
      }
    });
  }

  // ...
}
```

Ensuite, les composants hôtes doivent mettre à jour leurs enfants. Contrairement aux composants composites, ils peuvent posséder plusieurs enfants.

Dans cet exemple simplifié, nous utilisons un tableau d'instances internes et itérons dessus. Si le `type` reçu des instances internes correspond au `type` précédent, nous les mettons à jour, sinon nous les remplaçons. Le vrai réconciliateur prend également en compte la `key` de l'élément et détecte les déplacements en plus des insertions et des suppressions, mais nous omettons cette logique.

Nous collectons les opérations DOM sur les enfants dans une liste, afin de pouvoir les exécuter par lot :

```
// ...

// Ce sont des tableaux d'éléments React :
var prevChildren = nextProps.children || [];
if (!Array.isArray(prevChildren)) {
  prevChildren = [prevChildren];
}
var nextChildren = nextProps.children || [];
if (!Array.isArray(nextChildren)) {
  nextChildren = [nextChildren];
}
// Ce sont des tableaux d'instances internes :
var prevRenderedChildren = this.renderedChildren;
var nextRenderedChildren = [];

// Au fil de l'itération sur les enfants, nous ajouterons des opérations dans
```

```

le tableau.

var operationQueue = [];

// Remarque : la section ci-dessous est extrêmement simplifiée !
// Elle ne gère pas le réordonnancement, les enfants avec des trous ou les
// clés (`key`).
// Elle existe uniquement pour illustrer le flux général, pas les détails.

for (var i = 0; i < nextChildren.length; i++) {
  // Essaie de récupérer une instance interne existante pour cet enfant
  var prevChild = prevRenderedChildren[i];

  // S'il n'y a aucune instance interne pour cet index,
  // c'est qu'un enfant a été ajouté à la fin. Nous créons une nouvelle
  // instance interne, nous la montons, et nous utilisons son nœud.
  if (!prevChild) {
    var nextChild = instantiateComponent(nextChildren[i]);
    var node = nextChild.mount();

    // Enregistre le besoin d'ajouter un nœud
    operationQueue.push({type: 'ADD', node});
    nextRenderedChildren.push(nextChild);
    continue;
  }

  // Nous ne pouvons mettre à jour l'instance que si le type de son élément
  // est identique.
  // Par exemple, <Button size="small" /> peut être mis à jour par
  // <Button size="large" /> mais pas par un <App />.
  var canUpdate = prevChildren[i].type === nextChildren[i].type;

  // Si nous ne pouvons pas mettre à jour une instance existante, nous devons
  // la démonter
  // et monter une nouvelle à la place.
  if (!canUpdate) {
    var prevNode = prevChild.getHostNode();
    prevChild.unmount();

    var nextChild = instantiateComponent(nextChildren[i]);
    var nextNode = nextChild.mount();

    // Enregistre le besoin de remplacer un nœud
    operationQueue.push({type: 'REPLACE', prevNode, nextNode});
    nextRenderedChildren.push(nextChild);
    continue;
  }

  // Si nous pouvons mettre à jour une instance existante,
  // laissons-la simplement recevoir l'élément cible et gérer sa propre mise à
  // jour.
  prevChild.receive(nextChildren[i]);
  nextRenderedChildren.push(prevChild);
}

// Enfin, démonte tous les enfants qui n'existent pas :
for (var j = nextChildren.length; j < prevChildren.length; j++) {
  var prevChild = prevRenderedChildren[j];
  var node = prevChild.getHostNode();
  prevChild.unmount();

  // Enregistre le besoin de supprimer un nœud
  operationQueue.push({type: 'REMOVE', node});
}

```

```
}
```

```
// Actualise la liste des enfants issus du rendu.
```

```
this.renderedChildren = nextRenderedChildren;
```

```
// ...
```

Pour finir, nous exécutons les opérations sur le DOM. Encore une fois, le vrai code du réconciliateur est plus complexe car il gère également les déplacements :

```
// ...
// Traite les opérations dans la file d'attente
while (operationQueue.length > 0) {
    var operation = operationQueue.shift();
    switch (operation.type) {
        case 'ADD':
            this.node.appendChild(operation.node);
            break;
        case 'REPLACE':
            this.node.replaceChild(operation.nextNode, operation.prevNode);
            break;
        case 'REMOVE':
            this.node.removeChild(operation.node);
            break;
    }
}
```

Et c'est tout pour la mise à jour des composants hôtes.

Mise à jour de la racine

Maintenant que `CompositeComponent` et `DOMComponent` implémentent la méthode `receive(nextElement)`, nous pouvons modifier la fonction racine `mountTree()` pour l'utiliser lorsque le type de l'élément est identique à celui de la dernière fois :

```
function mountTree(element, containerNode) {
  // Vérifie l'existence d'un arbre
  if (containerNode.firstChild) {
    var prevNode = containerNode.firstChild;
    var prevRootComponent = prevNode._internalInstance;
    var prevElement = prevRootComponent.currentElement;

    // Si possible, réutilise le composant racine existant
    if (prevElement.type === element.type) {
      prevRootComponent.receive(element);
      return;
    }

    // Autrement, démonte l'arborescence existante
    unmountTree(containerNode);
  }
}

// ...
```

Désormais, appeler deux fois `mountTree()` avec le même type n'est plus destructif :

```
var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Réutilise le DOM existant :
mountTree(<App />, rootEl);
```

Et voilà les bases du fonctionnement interne de React.

Ce que nous avons laissé de côté

Ce document est simplifié par rapport au vrai code. Il y a quelques aspects importants que nous n'avons pas abordés :

- Les composants peuvent faire un rendu à `null` et le réconciliateur peut gérer des « emplacements vides » dans les tableaux et le résultat du rendu.
- Le réconciliateur lit également la `key` des éléments, et l'utilise pour déterminer quelle instance interne correspond à quel élément dans un tableau. Une grande partie de la complexité de la véritable implémentation de React est liée à ça.
- En plus des classes d'instance interne composite et hôte, il existe également des classes pour les composants « texte » et « vide ». Ils représentent des nœuds de texte et des « emplacements vides » que vous obtenez en faisant un rendu `null`.
- Les moteurs de rendu utilisent l'[injection](#) pour passer la classe interne hôte au réconciliateur. Par exemple, React DOM indique au réconciliateur d'utiliser `ReactDOMComponent` pour l'implémentation de l'instance interne hôte.
- La logique de mise à jour de la liste des enfants est extraite dans un *mixin* appelé `ReactMultiChild`, utilisé par les implémentations de la classe d'instance interne hôte dans React DOM comme dans React Native.
- Le réconciliateur implémente également la prise en charge de `setState()` dans les composants composites. Les mises à jour multiples dans les gestionnaires d'événements sont regroupées en lot pour une mise à jour unique.
- Le réconciliateur prend également en charge l'attachement et le détachement des refs aux composants composites et aux nœuds hôtes.
- Les méthodes de cycle de vie appelées une fois que le DOM est prêt, telles que `componentDidMount()` et `componentDidUpdate()`, sont rassemblées dans des « files d'attente de fonctions de rappel » et exécutées en un seul lot.
- React stocke les informations sur la mise à jour en cours dans un objet interne appelé « transaction ». Les transactions sont utiles pour garder trace de la liste des méthodes de cycle de vie en attente, de l'imbrication actuelle du DOM pour les avertissements et de tout ce qui est « global » à une mise à jour spécifique. Les transactions garantissent également que React « nettoie tout » après les mises à jour. Par exemple, la classe de transaction fournie par React DOM restaure après toute mise à jour l'éventuelle sélection d'une saisie.

Sauter dans le code

- `ReactMount` est l'endroit où réside le code équivalent aux `mountTree()` et `unmountTree()` de ce tutoriel. Il s'occupe du montage et du démontage des composants racines. `ReactNativeMount` est son homologue pour React Native.
- `ReactDOMComponent` est l'équivalent de `DOMComponent` dans ce tutoriel. Il implémente la classe de composant hôte pour le moteur de rendu de React DOM. `ReactNativeBaseComponent` est son équivalent React Native.
- `ReactCompositeComponent` est l'équivalent de `CompositeComponent` dans ce tutoriel. Il gère l'appel des composants définis par l'utilisateur et la maintenance de leur état local.
- `instantiateReactComponent` contient le code qui sélectionne la classe d'instance interne appropriée pour construire un élément. C'est l'équivalent de `instantiateComponent()` dans ce tutoriel.
- `ReactReconciler` est un enrobage avec les méthodes `mountComponent()`, `receiveComponent()` et `unmountComponent()`. Il appelle les implémentations sous-jacentes des instances internes, mais contient en prime du code de gestion partagé par toutes ces implémentations.
- `ReactChildReconciler` implémente la logique de montage, de mise à jour et de démontage des enfants en fonction de la `key` de leurs éléments.
- `ReactMultiChild` implémente le traitement de la file d'attente des opérations pour les insertions, les suppressions et les déplacements d'enfants indépendamment du moteur de rendu.
- `mount()`, `receive()` et `unmount()` sont nommées en réalité `mountComponent()`, `receiveComponent()` et `unmountComponent()` dans la base de code React pour des raisons historiques, mais elles reçoivent des éléments.
- Les propriétés des instances internes commencent par un tiret bas (*underscore*, *NdT*), par exemple `_currentElement`. Elles sont considérées dans la base de code comme des champs publics en lecture seule.

Orientations futures

Le réconciliateur "stack" a des limitations intrinsèques, comme le fait qu'il soit synchrone et incapable d'interrompre le travail ou de le découper en plusieurs morceaux. Le [nouveau réconciliateur "fiber"](#) a une [architecture complètement différente](#). À partir de React 16, il remplace le réconciliateur "stack".

Prochaines étapes

Lisez la [page suivante](#) pour en apprendre davantage sur les principes de conception que nous utilisons pour le développement de React.

[Go to TOC](#)

Intégration avec d'autres bibliothèques

React peut être utilisé dans n'importe quelle application web. Il peut être intégré à d'autres applications et, avec un peu de soin, d'autres applications peuvent être intégrées à React. Ce guide examine certains des cas d'utilisation les plus courants, en se concentrant sur l'intégration avec [jQuery](#) et [Backbone](#), mais les mêmes idées peuvent être appliquées à l'intégration de composants à n'importe quel code existant.

Intégration aux plugins de manipulation DOM

React ignore les modifications apportées au DOM en dehors de React. Il détermine les mises à jour en fonction de sa propre représentation interne et, si les mêmes noeuds du DOM sont manipulés par une autre bibliothèque, React s'emmêle les pinceaux et ne dispose d aucun moyen de se rétablir.

Ça ne signifie pas qu'il est impossible ni même nécessairement difficile de combiner React avec d'autres moyens d'affecter le DOM, vous devez simplement être attentifs à ce que chacun fait.

Le moyen le plus simple d'éviter les conflits consiste à empêcher le composant React de mettre à jour l'élément « partagé ». Vous pouvez faire ça en utilisant des éléments que React n'aura aucune raison de mettre à jour, comme une `<div />` vide.

Comment aborder le problème

Pour illustrer ça, esquissons un enrobage pour un plugin générique jQuery.

Nous attacherons une `ref` à l'élément racine DOM. Dans `componentDidMount`, nous obtiendrons une référence qui pourra être passée au plugin jQuery.

Pour empêcher React de toucher au DOM après le montage, nous renverrons une `<div />` vide depuis la méthode `render()`. L'élément `<div />` n'a pas de propriétés ni d'enfants, donc React n'a aucune raison de le mettre à jour, laissant le plugin jQuery libre de gérer cette partie du DOM :

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.somePlugin();
  }

  componentWillUnmount() {
    this.$el.somePlugin('destroy');
  }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

Notez que nous avons défini les deux méthodes de cycle de vie `componentDidMount` et `componentWillUnmount`. De nombreux plugins jQuery attachent des gestionnaires d'événements au DOM, il est donc important de les détacher dans `componentWillUnmount`. Si le plugin ne fournit pas de méthode de nettoyage, vous devrez probablement fournir la vôtre, en vous souvenant de supprimer tous les gestionnaires d'événements enregistrés par le plugin pour éviter les fuites de mémoire.

Intégration avec le plugin jQuery Chosen

Pour un exemple plus concret de ces concepts, écrivons un enrobage minimal pour le plugin `Chosen`, qui enrichit les champs `<select>`.

Remarque

Ça n'est pas parce que c'est possible qu'il s'agit de la meilleure approche pour les applications React. Nous vous encourageons à utiliser les composants React lorsque vous le pouvez. Les composants React sont plus faciles à réutiliser dans les applications React et permettent souvent de mieux contrôler leur comportement et leur apparence.

Tout d'abord, regardons ce que `Chosen` fait au DOM.

Si vous lappelez sur un nœud DOM `<select>`, il lit les attributs du nœud DOM d'origine, le masque avec du style en ligne, puis ajoute un nœud DOM distinct avec sa propre représentation visuelle juste après le `<select>`. Ensuite, il déclenche des événements jQuery pour nous informer des modifications.

Disons que nous visons cette API pour notre composant React d'enrobage `<Chosen>` :

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>vanille</option>
      <option>chocolat</option>
      <option>fraise</option>
    </Chosen>
  );
}
```

Nous allons l'implémenter en tant que `composant non-contrôlé` pour plus de simplicité.

Pour commencer, nous allons créer un composant vide avec une méthode `render()` où nous renvoyons `<select>` enrobé dans une `<div>` :

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

```

    );
}
}
```

Remarquez comment nous avons enrobé `<select>` dans une `<div>` supplémentaire. C'est nécessaire car Chosen ajoutera un autre élément DOM juste après le nœud `<select>` que nous lui aurons transmis. Cependant, en ce qui concerne React, `<div>` n'a toujours qu'un seul enfant. C'est de cette manière que nous nous assurons que les mises à jour React n'entreront pas en conflit avec le nœud DOM supplémentaire ajouté par Chosen. Si vous modifiez le DOM en-dehors du flux React, il est important que vous vous assurez que React n'ait aucune raison de toucher à ces nœuds du DOM.

Ensuite, nous allons implémenter les méthodes de cycle de vie. Nous devons initialiser Chosen avec la référence du nœud `<select>` dans `componentDidMount`, et nettoyer le tout dans `componentWillUnmount` :

```

componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();
}

componentWillUnmount() {
  this.$el.chosen('destroy');
}
```

Essayez dans CodePen

Notez que React n'attribue aucune signification particulière au champ `this.el`. Ça ne fonctionne que parce que nous avons déjà affecté une `ref` à ce champ dans la méthode `render()` :

```
<select className="Chosen-select" ref={el => this.el = el}>
```

Ça suffit pour afficher notre composant, mais nous souhaitons également être informés des modifications de valeur. Pour ce faire, nous nous abonnerons à l'événement jQuery `change` sur le `<select>` géré par Chosen.

Nous ne transmettrons pas `this.props.onChange` directement à Chosen, car les props du composant peuvent changer avec le temps, ce qui inclut les gestionnaires d'événements. Au lieu de ça, nous allons déclarer une méthode `handleChange()` qui appelle `this.props.onChange`, et l'utiliserons pour nous abonner à l'événement jQuery `change` :

```

componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}
```

```
handleChange(e) {
  this.props.onChange(e.target.value);
}
```

Essayez dans CodePen

Enfin, il reste une dernière chose à faire. Dans React, les props peuvent changer avec le temps. Par exemple, le composant `<Chosen>` peut avoir différents enfants si l'état du composant parent change. Ça signifie qu'aux points d'intégration, il est important de mettre à jour manuellement le DOM en réponse aux mises à jour des props, car nous ne laissons plus React gérer le DOM pour nous.

La documentation de Chosen suggère que nous pouvons utiliser l'API `trigger()` de jQuery pour l'informer des modifications apportées à l'élément DOM d'origine. Nous laisserons React se charger de la mise à jour de `this.props.children` dans `<select>`, mais nous ajouterons également une méthode de cycle de vie `componentDidUpdate()` notifiant Chosen de tout changement dans la liste des enfants:

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}
```

De cette façon, Chosen saura mettre à jour son élément DOM lorsque les enfants de `<select>` gérés par React changent.

L'implémentation complète du composant `Chosen` ressemble à ceci :

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
  }

  componentDidUpdate(prevProps) {
    if (prevProps.children !== this.props.children) {
      this.$el.trigger("chosen:updated");
    }
  }

  componentWillUnmount() {
    this.$el.off('change', this.handleChange);
    this.$el.chosen('destroy');
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
    );
  }
}
```

```

        </div>
    );
}
}

```

[Essayez dans CodePen](#)

Intégration à d'autres bibliothèques de vues

React peut être intégré à d'autres applications grâce à la flexibilité de `ReactDOM.render()`.

Bien que React soit couramment utilisé au démarrage pour charger un seul composant racine React dans le DOM, `ReactDOM.render()` peut également être appelé plusieurs fois pour afficher des parties indépendantes de l'interface utilisateur (UI), qui peuvent être aussi petites qu'un bouton ou aussi grandes qu'une application.

En fait, c'est exactement comme ça que React est utilisé chez Facebook. Ça nous permet d'écrire des applications en React morceau par morceau, et de les combiner avec nos gabarits existants générés par le serveur et d'autres codes côté client.

Remplacement d'un rendu basé chaînes par React

Une pratique courante dans les anciennes applications web consiste à décrire des fragments du DOM sous forme de chaîne et à les insérer dans le DOM comme suit : `$el.html(htmlString)`. Ces points d'insertion dans une base de code sont parfaits pour introduire React. Il suffit de réécrire le rendu basé sur une chaîne sous la forme d'un composant React.

Donc, l'implémentation jQuery suivante...

```

$('#container').html('<button id="btn">Dis bonjour</button>');
$('#btn').click(function() {
  alert('Bonjour !');
});

```

... pourrait être réécrite en utilisant un composant React :

```

function Button() {
  return <button id="btn">Dis bonjour</button>;
}

ReactDOM.render(
  <Button />,
  document.getElementById('container'),
  function() {
    $('#btn').click(function() {
      alert('Bonjour !');
    });
  }
);

```

À partir de là, vous pouvez intégrer plus de logique dans le composant et commencer à adopter des pratiques React plus courantes. Par exemple, dans les composants, il est préférable de ne pas compter sur les ID, car un même composant peut être affiché plusieurs fois. Nous utiliserons plutôt le [système d'événements React](#) et enregistrerons le gestionnaire de clics directement sur l'élément React `<button>` :

```
function Button(props) {
  return <button onClick={props.onClick}>Dis bonjour</button>;
}

function HelloButton() {
  function handleClick() {
    alert('Bonjour !');
  }
  return <Button onClick={handleClick} />;
}

ReactDOM.render(
  <HelloButton />,
  document.getElementById('container')
);
```

[Essayez dans CodePen](#)

Vous pouvez avoir autant de composants isolés que vous le souhaitez et utiliser `ReactDOM.render()` pour les restituer dans différents conteneurs DOM. Au fur et à mesure que vous convertissez les parties de votre application en React, vous pourrez les combiner en composants plus volumineux et déplacer une partie des appels à `ReactDOM.render()` plus haut dans la hiérarchie.

Incorporer React dans une vue Backbone

Les vues [Backbone](#) utilisent généralement des chaînes HTML ou des fonctions de gabarit produisant des chaînes pour créer le contenu de leurs éléments DOM. Ce processus peut également être remplacé par le rendu d'un composant React.

Ci-dessous, nous allons créer une vue Backbone appelée `ParagraphView`. La fonction `render()` de Backbone sera remplacée pour afficher un composant React `<Paragraph>` dans l'élément DOM fourni par Backbone (`this.el`). Ici aussi, nous utilisons `ReactDOM.render()` :

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  render() {
    const text = this.model.get('text');
    ReactDOM.render(<Paragraph text={text} />, this.el);
    return this;
  },
  remove() {
    ReactDOM.unmountComponentAtNode(this.el);
    Backbone.View.prototype.remove.call(this);
  }
});
```

[Essayez dans CodePen](#)

Il est important que nous appelions également `ReactDOM.unmountComponentAtNode()` dans la méthode `remove` afin que React désinscrive les gestionnaires d'événements et les autres ressources associées à l'arborescence des composants lorsqu'ils sont détachés.

Lorsqu'un composant est supprimé *depuis* une arborescence React, le nettoyage est effectué automatiquement, mais comme nous supprimons l'arborescence complète à la main, nous devons appeler cette méthode.

Intégration avec des couches modèles

Bien qu'il soit généralement recommandé d'utiliser un flux de données unidirectionnel tel que [l'état de React](#), [Flux](#) ou [Redux](#), les composants React peuvent utiliser une couche de modèle provenant d'autres frameworks et bibliothèques.

Utiliser les modèles Backbone dans les composants React

Le moyen le plus simple de consommer des modèles et des collections [Backbone](#) à partir d'un composant React consiste à écouter les divers événements de modification et à forcer manuellement une mise à jour.

Les composants responsables de l'affichage des modèles écouteraient les événements `'change'`, tandis que les composants responsables de l'affichage des collections écouteraient les événements `'add'` et `'remove'`. Dans les deux cas,appelez `this.forceUpdate()` pour rafraîchir le composant avec les nouvelles données.

Dans l'exemple ci-dessous, le composant `List` affiche une collection Backbone, en utilisant le composant `Item` pour restituer des éléments individuels.

```
class Item extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

class List extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }
```

```

}

handleChange() {
  this.forceUpdate();
}

componentDidMount() {
  this.props.collection.on('add', 'remove', this.handleChange);
}

componentWillUnmount() {
  this.props.collection.off('add', 'remove', this.handleChange);
}

render() {
  return (
    <ul>
      {this.props.collection.map(model => (
        <Item key={model.cid} model={model} />
      ))}
    </ul>
  );
}
}

```

[Essayez dans CodePen](#)

Extraction de données à partir des modèles Backbone

L'approche ci-dessus nécessite que vos composants React soient conscients des modèles et des collections Backbone. Si vous envisagez par la suite de migrer vers une autre solution de gestion de données, vous voudrez peut-être concentrer les connaissances sur Backbone dans le moins possible de parties du code.

Une solution à ce problème consiste à extraire les attributs du modèle sous forme de données simples à chaque modification, et à conserver cette logique en un seul endroit. Ce qui suit est [un composant d'ordre supérieur](#) qui extrait tous les attributs d'un modèle Backbone et les stocke dans son état, pour finalement passer les données au composant enveloppé.

De cette façon, seul le composant d'ordre supérieur doit connaître les détails du modèle Backbone, et la plupart des composants de l'application peuvent rester indépendants de Backbone.

Dans l'exemple ci-dessous, nous allons copier les attributs du modèle pour former l'état initial. Nous nous abonnons à l'événement `change` (et nous nous désabonnons lors du démontage), et lorsque l'événement survient, nous mettons à jour l'état avec les attributs actuels du modèle. Enfin, nous nous assurons que si la propriété `model` elle-même change, nous n'oublions pas de nous désabonner de l'ancien modèle et de nous abonner au nouveau.

Notez que cet exemple n'est pas exhaustif en ce qui concerne l'utilisation de Backbone, mais il devrait vous donner une idée de la manière générique de l'aborder :

```

function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = Object.assign({}, props.model.attributes);
    }
  }
}

```

```
        this.handleChange = this.handleChange.bind(this);
    }

    componentDidMount() {
        this.props.model.on('change', this.handleChange);
    }

    componentWillReceiveProps(nextProps) {
        this.setState(Object.assign({}, nextProps.model.attributes));
        if (nextProps.model !== this.props.model) {
            this.props.model.off('change', this.handleChange);
            nextProps.model.on('change', this.handleChange);
        }
    }

    componentWillUnmount() {
        this.props.model.off('change', this.handleChange);
    }

    handleChange(model) {
        this.setState(model.changedAttributes());
    }

    render() {
        const propsExceptModel = Object.assign({}, this.props);
        delete propsExceptModel.model;
        return <WrappedComponent {...propsExceptModel} {...this.state} />;
    }
}
```

Pour montrer comment l'utiliser, connectons un composant `NameInput` React à un modèle Backbone et mettons à jour son attribut `firstName` chaque fois que la saisie change :

```
function NameInput(props) {
  return (
    <p>
      <input value={props.firstName} onChange={props.handleChange} />
      <br />
      Je m'appelle {props.firstName}.
    </p>
  );
}

const BackboneNameInput = connectToBackboneModel(NameInput);

function Example(props) {
  function handleChange(e) {
    props.model.set('firstName', e.target.value);
  }

  return (
    <BackboneNameInput
      model={props.model}
      handleChange={handleChange}
    />
  );
}

const model = new Backbone.Model({ firstName: 'Frodo' });
ReactDOM.render(
```

```
<Example model={model} />,
document.getElementById('root')
);
```

Essayez dans CodePen

Cette technique ne se limite pas à Backbone. Vous pouvez utiliser React avec n'importe quelle bibliothèque de modèles en vous abonnant à ses modifications dans les méthodes de cycle de vie et, éventuellement, en copiant les données dans l'état local React.

[Go to TOC](#)

Introduction à JSX

Observez cette déclaration de variable :

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

Cette drôle de syntaxe n'est ni une chaîne de caractères ni du HTML.

Ça s'appelle du JSX, et c'est une extension syntaxique de JavaScript. Nous recommandons de l'utiliser avec React afin de décrire à quoi devrait ressembler l'interface utilisateur (UI). JSX vous fait sûrement penser à un langage de balisage, mais il recèle toute la puissance de JavaScript.

JSX produit des « éléments » React. Nous verrons comment les retranscrire dans le DOM dans la [prochaine section](#). Dans la suite de ce document, nous verrons les bases de JSX dont vous aurez besoin pour bien démarrer.

Pourquoi JSX ?

Le fonctionnement d'une UI conditionnera toujours les logiques de rendu, de la gestion des événements à la préparation des données pour l'affichage, en passant par l'évolution de l'état au fil du temps. React a choisi d'assumer pleinement cet état de fait.

Au lieu de séparer artificiellement les *technologies* en mettant le balisage et la logique dans des fichiers séparés, React [sépare les préoccupations](#) via des unités faiblement couplées appelées « composants », qui contiennent les deux. Nous reviendrons sur les composants dans une [prochaine section](#), mais si l'idée d'injecter des balises dans du JS vous met mal à l'aise, [cette présentation](#) vous fera peut-être changer d'avis.

React [ne vous oblige pas](#) à utiliser JSX, mais la plupart des gens y trouvent une aide visuelle quand ils manipulent l'interface utilisateur dans le code JavaScript. Ça permet aussi à React de produire des messages d'erreurs et d'avertissements plus utiles.

Ceci étant posé, commençons !

Utiliser des expressions dans JSX

Dans l'exemple suivant, nous déclarons une variable appelée `name` et nous l'utilisons ensuite dans JSX en l'encadrant avec des accolades :

```
const name = 'Clarisse Agbegnenou';
const element = <h1>Bonjour, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Vous pouvez utiliser n'importe quelle [expression JavaScript](#) valide dans des accolades en JSX. Par exemple, `2 + 2`, `user.firstName`, ou `formatName(user)` sont toutes des expressions JavaScript valides.

Dans l'exemple suivant, on intègre le résultat de l'appel d'une fonction JavaScript, `formatName(user)`, dans un élément `<h1>`.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Kylian',
  lastName: 'Mbappé'
};

const element = (
  <h1>
    Bonjour, {formatName(user)} !
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

[Essayer sur CodePen](#)

On découpe le JSX en plusieurs lignes pour une meilleure lisibilité. Par la même occasion, nous recommandons également de le mettre entre parenthèses afin d'éviter les pièges [d'insertion de point-virgule automatique](#), même si cette pratique n'est pas obligatoire.

JSX n'est rien d'autre qu'une expression

Après la compilation, les expressions JSX deviennent de simples appels de fonctions JavaScript, dont l'évaluation renvoie des objets JavaScript.

Ça signifie que vous pouvez utiliser JSX à l'intérieur d'instructions `if` ou de boucles `for`, l'affecter à des variables, l'accepter en tant qu'argument, et le renvoyer depuis des fonctions :

```
function getGreeting(user) {
  if (user) {
    return <h1>Bonjour, {formatName(user)} !</h1>;
  }
  return <h1>Bonjour, Belle Inconnue.</h1>;
}
```

Spécifier des attributs en JSX

Vous pouvez utiliser des guillemets pour spécifier des littéraux chaînes de caractères dans les attributs :

```
const element = <div tabIndex="0"></div>;
```

Vous pouvez aussi utiliser des accolades pour utiliser une expression JavaScript dans un attribut :

```
const element = <img src={user.avatarUrl}></img>;
```

Ne mettez pas de guillemets autour des accolades quand vous utilisez une expression JavaScript dans un attribut. Vous pouvez utiliser soit des guillemets (pour des valeurs textuelles) soit des accolades (pour des expressions), mais pas les deux à la fois pour un même attribut.

Attention :

Dans la mesure où JSX est plus proche de JavaScript que de HTML, React DOM utilise la casse `camelCase` comme convention de nommage des propriétés, au lieu des noms d'attributs HTML.

Par exemple, `class` devient `className` en JSX, et `tabindex` devient `tabIndex`.

Spécifier des éléments enfants en JSX

Si une balise est vide, vous pouvez la fermer immédiatement avec `/>`, comme en XML :

```
const element = <img src={user.avatarUrl} />;
```

Les balises JSX peuvent contenir des enfants :

```
const element = (
  <div>
    <h1>Bonjour !</h1>
    <h2>Content de te voir ici.</h2>
  </div>
);
```

JSX empêche les attaques d'injection

Vous ne risquez rien en utilisant une saisie utilisateur dans JSX :

```
const title = response.potentiallyMaliciousInput;
// Ceci est sans risque :
const element = <h1>{title}</h1>;
```

Par défaut, React DOM `échappe` toutes les valeurs intégrées avec JSX avant d'en faire le rendu. Il garantit ainsi que vous ne risquez jamais d'injecter quoi que ce soit d'autre que ce que vous avez explicitement écrit dans votre application. Tout est converti en chaîne de caractères avant de produire le rendu. Ça aide à éviter les attaques [XSS \(cross-site-scripting\)](#).

JSX représente des objets

Babel compile JSX vers des appels à `React.createElement()`.

Ces deux exemples sont identiques :

```
const element = (
  <h1 className="greeting">
    Bonjour, monde !
  </h1>
);
```

```
const element = React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Bonjour, monde !'  
)
```

`React.createElement()` effectue quelques vérifications pour vous aider à écrire un code sans bug, mais pour l'essentiel il crée un objet qui ressemble à ceci :

```
// Remarque : cette structure est simplifiée  
const element = {  
  type: 'h1',  
  props: {  
    className: 'greeting',  
    children: 'Bonjour, monde !'  
  }  
};
```

Ces objets sont appelés des « éléments React ». Vous pouvez les considérer comme des descriptions de ce que vous voulez voir sur l'écran. React lit ces objets et les utilise pour construire le DOM et le tenir à jour.

Nous explorerons la retranscription des éléments React dans le DOM dans la [prochaine section](#).

Astuce :

Nous recommandons d'utiliser la [définition de langage « Babel »](#) dans votre éditeur préféré, afin que les codes ES6 et JSX soient correctement colorisés.

JSX dans le détail

Fondamentalement, JSX fournit juste du sucre syntaxique pour la fonction `React.createElement(component, props, ...children)`.

Le code JSX :

```
<MyButton color="blue" shadowSize={2}>
  Cliquez ici
</MyButton>
```

est compilé en :

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

Il est aussi possible d'utiliser la balise auto-fermante si il n'y a pas d'enfants. Donc :

```
<div className="sidebar" />
```

est compilé en :

```
React.createElement(
  'div',
  {className: 'sidebar'}
)
```

Si vous souhaitez voir comment certains éléments JSX spécifiques sont compilés en JavaScript, vous pouvez utiliser [le compilateur Babel en ligne](#).

Spécifier le type d'un élément React

La première partie d'une balise JSX détermine le type de l'élément React en question.

Les types commençant par une lettre majuscule indiquent que la balise JSX fait référence à un composant React. Ces balises sont compilées en références directes à la variable nommée, donc si vous utilisez l'expression JSX `<Foo />`, l'identifiant `Foo` doit être présent dans la portée.

React doit être présent dans la portée

Étant donné que JSX se compile en appels à `React.createElement`, la bibliothèque `React` doit aussi être présente dans la portée de votre code JSX.

Par exemple, les deux imports sont nécessaires dans le code ci-dessous même si `React` et `CustomButton` ne sont pas directement référencés depuis JavaScript :

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

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

Si vous n'utilisez pas un *bundler* JavaScript mais que vous chargez React à partir d'une balise `<script>`, il est déjà dans la portée en tant que variable globale `React`.

Utiliser la notation à points pour un type JSX

Vous pouvez également référencer un composant React en utilisant la notation à points dans JSX. C'est pratique si vous avez un seul module qui exporte de nombreux composants React. Par exemple si `MyComponents.DatePicker` est un composant, vous pouvez directement l'utiliser dans JSX comme ceci :

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imaginez un sélecteur de dates {props.color} ici.</div>;
  }
}

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

Les composants utilisateurs doivent commencer par une majuscule

Quand un élément commence par une lettre minuscule il fait référence à un composant natif tel que `<div>` ou ``, ce qui donne une chaîne de caractères `'div'` ou `'span'` passée à `React.createElement`. Les types qui commencent par une lettre majuscule comme `<Foo />` sont compilés en `React.createElement(Foo)` et correspondent à un composant défini ou importé dans votre fichier JavaScript.

Nous recommandons de nommer vos composants avec une initiale majuscule. Si vous avez un composant qui démarre avec une lettre minuscule, affectez-le à une variable avec une initiale majuscule avant de l'utiliser dans votre JSX.

Par exemple, ce code ne s'exécutera pas comme prévu :

```
import React from 'react';

// Faux ! C'est un composant, il devrait commencer par une lettre majuscule :
function hello(props) {
  // Correct ! Cette utilisation de <div> fonctionne car div est une balise HTML
  valide :
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Faux ! React pense que <hello /> est une balise HTML car il ne commence pas
```

```
par une majuscule :
  return <hello toWhat="World" />;
}
```

Pour corriger ça, nous allons renommer `hello` en `Hello` et utiliser `<Hello />` lorsqu'on y fait référence :

```
import React from 'react';

// Correct ! C'est un composant, il doit avoir une initiale majuscule :
function Hello(props) {
  // Correct ! Cette utilisation de <div> fonctionne car div est une balise HTML valide :
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Correct ! React sait que <Hello /> est un composant car il commence par une majuscule.
  return <Hello toWhat="World" />;
}
```

Choix du type au moment de l'exécution

Vous ne pouvez pas utiliser une expression générale pour le type d'un élément React. Si vous voulez utiliser une expression pour définir le type d'un élément, affectez-la d'abord à une variable dont l'initiale est majuscule. Ça arrive en général lorsque vous voulez afficher un composant différent en fonction d'une prop :

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

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

function Story(props) {
  // Faux ! Un type JSX ne peut pas être une expression.
  return <components[props.storyType] story={props.story} />;
}
```

Pour corriger ça, nous allons d'abord affecter le type à une variable dont l'identifiant commence par une majuscule :

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

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

function Story(props) {
  // Correct ! Un type JSX peut être une variable commençant par une majuscule.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

Les props en JSX

Il y a différents moyens de définir les props en JSX.

Les expressions JavaScript comme props

Vous pouvez passer n'importe quelle expression JavaScript comme prop, en l'entourant avec des accolades `{}`. Par exemple, dans ce code JSX :

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

Pour `MyComponent`, la valeur de `props.foo` sera `10` parce que l'expression `1 + 2 + 3 + 4` est calculée.

Les instructions `if` et les boucles `for` ne sont pas des expressions en JavaScript, donc elle ne peuvent pas être directement utilisées en JSX. Au lieu de ça, vous pouvez les mettre dans le code environnant. Par exemple :

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>pair</strong>;
  } else {
    description = <i>impair</i>;
  }
  return <div>{props.number} est un nombre {description}</div>;
}
```

Vous pouvez en apprendre davantage sur les [conditions](#) et les [boucles](#) au sein des sections correspondantes de la documentation.

Les littéraux chaînes

Vous pouvez passer un littéral chaîne comme prop. Les deux expressions JSX ci-dessous sont équivalentes :

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

Quand vous passez un littéral chaîne, sa valeur subit un échappement HTML inverse. Ces deux expressions JSX sont donc équivalentes :

```
<MyComponent message=&lt;3&gt; />
<MyComponent message={'<3'}/>
```

Ce comportement n'est en général pas pertinent (au sens où vous n'avez pas à vous en soucier particulièrement), ce n'est mentionné ici que par souci d'exhaustivité.

Les props valent `true` par défaut

Si vous n'affectez aucune valeur à une prop, sa valeur par défaut sera `true`. Ces deux expressions JSX sont équivalentes :

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

En général, nous déconseillons cette syntaxe qui *ne passe pas* de valeur à une *prop*, car ça peut être confondu avec [la notation ES6 de propriétés concises `{foo}`](#) qui est l'abréviation de `{foo: foo}` et non de `{foo: true}`. Ce comportement existe uniquement par souci de cohérence avec HTML.

Décomposition des props

Si vous avez déjà un objet `props` et souhaitez l'utiliser en JSX, vous pouvez utiliser l'opérateur de décomposition (*spread operator, NdT*) `...` pour passer l'ensemble de l'objet `props`. Ces deux composants sont équivalents :

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

Vous pouvez également choisir certaines `props` que votre composant utilisera en passant toutes les autres `props` avec l'opérateur de *rest*.

```
const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("cliqué !")}>
        Bonjour monde !
      </Button>
    </div>
  );
};
```

Dans l'exemple ci-dessus, la prop `kind` est extraite pour le composant principal et *n'est pas* passée à l'élément `<button>` du DOM. Toutes les autres `props` sont passées via l'objet `...other`, ce qui rend ce composant très flexible. Vous pouvez voir qu'il passe les `props` `onClick` et `children`.

La décomposition des `props` peut être utile, mais elle permet aussi de passer trop facilement des `props` inutiles aux composants, ou de passer des attributs HTML invalides au DOM. Nous vous conseillons d'utiliser cette syntaxe avec parcimonie.

Les éléments enfants en JSX

Dans les expressions JSX qui comportent une balise ouvrante et une balise fermante, le contenu entre ces deux balises est passé comme une prop spéciale : `props.children`. Il existe plusieurs moyens pour passer ces enfants :

Littéraux chaînes

Vous pouvez mettre une chaîne de caractères entre une balise ouvrante et une fermante et `props.children` sera juste cette chaîne de caractères. C'est utile pour la plupart des éléments HTML natifs. Par exemple :

```
<MyComponent>Bonjour monde !</MyComponent>
```

C'est du JSX valide, et `props.children` dans `MyComponent` sera simplement la chaîne de caractères "Bonjour monde !" . Le HTML subit un échappement inverse, donc vous pouvez généralement écrire du JSX de la même façon que vous écrivez du HTML, c'est-à-dire :

```
<div>Ce contenu est valide en HTML && en JSX.</div>
```

JSX supprime les espaces en début et en fin de ligne. Il supprime également les lignes vides. Les sauts de lignes adjacents aux balises sont retirés ; les sauts de lignes apparaissant au sein de littéraux chaînes sont ramenés à une seule espace. Du coup, tous les codes ci-dessous donnent le même résultat :

```
<div>Bonjour monde</div>

<div>
  Bonjour monde
</div>

<div>
  Bonjour
  monde
</div>

<div>
  Bonjour monde
</div>
```

Éléments JSX enfants

Vous pouvez fournir des éléments JSX supplémentaires en tant qu'enfants. C'est utile pour afficher des composants imbriqués :

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

Vous pouvez mélanger différents types d'enfants, comme par exemple des littéraux chaînes et des éléments JSX. Là encore, JSX est similaire à HTML, de sorte que le code suivant est valide tant en HTML qu'en JSX :

```
<div>
  Voici une liste :
  <ul>
    <li>Élément 1</li>
    <li>Élément 2</li>
  </ul>
</div>
```

Un composant React peut aussi renvoyer un tableau d'éléments :

```
render() {
  // Pas besoin d'enrober les éléments de la liste dans un élément
  // supplémentaire !
  return [
    // N'oubliez pas les "keys" :)
    <li key="A">Premier élément</li>,
    <li key="B">Deuxième élément</li>,
    <li key="C">Troisième élément</li>,
  ];
}
```

Les expressions JavaScript comme enfants

Vous pouvez passer n'importe quelle expression JavaScript en tant qu'enfant, en l'enrobant avec des accolades `{ }`. Ainsi, ces expressions sont équivalentes :

```
<MyComponent>foo</MyComponent>
<MyComponent>{'foo'}</MyComponent>
```

C'est souvent utile pour afficher une liste d'expressions JSX de longueur quelconque. Par exemple, ce code affiche une liste HTML :

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finir la doc', 'envoyer la pr', 'tanner Chris pour une revue'];
  return (
    <ul>
      {todos.map((message) => <Item key={message} message={message} />)}
    </ul>
  );
}
```

Les expressions JavaScript peuvent être mélangées avec d'autres types d'enfants. C'est souvent utile en remplacement de gabarits textuels :

```
function Hello(props) {
  return <div>Bonjour {props.addressee} !</div>;
}
```

Les fonctions comme enfants

En temps normal, les expressions Javascript insérées dans JSX produiront une chaîne, un élément React ou une liste de ces types. Cependant, `props.children` fonctionne exactement comme n'importe quelle prop dans le sens où elle peut passer n'importe quel genre de données, pas seulement celles que React sait afficher.

Par exemple, si vous avez un composant personnalisé, vous pouvez lui faire accepter une fonction de rappel dans `props.children` :

```
// Appelle la fonction de rappel children à raison de numTimes fois
// afin de produire une répétition du composant
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

function ListOfTenThings() {
  return (
    <Repeat numTimes={10}>
      {(index) => <div key={index}>Ceci est l'élément {index} de la liste</div>}
    </Repeat>
  );
}
```

Les enfants passés à un composant personnalisé peuvent être n'importe quoi, du moment que ce composant les transforme en quelque chose que React peut comprendre avant le rendu. Cette utilisation n'est pas courante, mais elle fonctionne si vous voulez étendre ce dont JSX est capable.

Les booléens ainsi que `null` et `undefined` sont ignorés

`false`, `null`, `undefined`, et `true` sont des enfants valides. Ils ne sont simplement pas exploités. Ces expressions JSX produiront toutes la même chose :

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

Ça peut être utile pour afficher des éléments React de façon conditionnelle. Ce JSX produit un composant `<Header />` uniquement si `showHeader` est à `true` :

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

Une mise en garde s'impose : certaines valeurs *falsey*, comme le nombre `0`, sont tout de même affichées par React. Par exemple, ce code ne se comportera pas comme vous l'espérez car il affichera `0` lorsque `props.messages` est un tableau vide :

```
<div>
  {props.messages.length &&
   <MessageList messages={props.messages} />
  }
</div>
```

Pour corriger ça, assurez-vous que l'expression avant `&&` est toujours un booléen :

```
<div>
  {props.messages.length > 0 &&
   <MessageList messages={props.messages} />
  }
</div>
```

Réciproquement, si vous voulez qu'une valeur comme `false`, `true`, `null`, ou `undefined` soit bien affichée, vous devez d'abord la [convertir en chaîne](#) :

```
<div>
  Ma variable Javascript est {String(myVariable)}.
</div>
```

Legacy Context

Note:

The legacy context API will be removed in a future major version. Use the [new context API](#) introduced with version 16.3. The legacy API will continue working for all 16.x releases.

How To Use Context

This section documents a legacy API. See the [new API](#).

Suppose you have a structure like:

```
class Button extends React.Component {
  render() {
    return (
      <button style={{background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
}

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  render() {
    const color = "purple";
    const children = this.props.messages.map((message) =>
      <Message text={message.text} color={color} />
    );
    return <div>{children}</div>;
  }
}
```

In this example, we manually thread through a `color` prop in order to style the `Button` and `Message` components appropriately. Using context, we can pass this through the tree automatically:

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

class Button extends React.Component {
```

```

render() {
  return (
    <button style={{background: this.context.color}}>
      {this.props.children}
    </button>
  );
}

Button.contextTypes = {
  color: PropTypes.string
};

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  getChildContext() {
    return {color: "purple"};
  }

  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}

MessageList.childContextTypes = {
  color: PropTypes.string
};

```

By adding `childContextTypes` and `getChildContext` to `MessageList` (the context provider), React passes the information down automatically and any component in the subtree (in this case, `Button`) can access it by defining `contextTypes`.

If `contextTypes` is not defined, then `context` will be an empty object.

Note:

`React.PropTypes` has moved into a different package since React v15.5. Please use [the prop-type library instead](#) to define `contextTypes`.

We provide a [codemod script](#) to automate the conversion.

Parent-Child Coupling

This section documents a legacy API. See the [new API](#).

Context can also let you build an API where parents and children communicate. For example, one library that works this way is [React Router V4](#):

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr />

      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/topics" component={Topics} />
    </div>
  </Router>
);
```

By passing down some information from the `Router` component, each `Link` and `Route` can communicate back to the containing `Router`.

Before you build components with an API similar to this, consider if there are cleaner alternatives. For example, you can pass entire React components as props if you'd like to.

Referencing Context in Lifecycle Methods

This section documents a legacy API. See the [new API](#).

If `contextTypes` is defined within a component, the following [lifecycle methods](#) will receive an additional parameter, the `context` object:

- `constructor(props, context)`
- `componentWillReceiveProps(nextProps, nextContext)`
- `shouldComponentUpdate(nextProps, nextState, nextContext)`
- `componentWillUpdate(nextProps, nextState, nextContext)`

Note:

As of React 16, `componentDidUpdate` no longer receives `prevContext`.

Referencing Context in Function Components

This section documents a legacy API. See the [new API](#).

Function components are also able to reference `context` if `contextTypes` is defined as a property of the function. The following code shows a `Button` component written as a function component.

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

const Button = ({children}, context) =>
  <button style={{background: context.color}}>
    {children}
  </button>

Button.contextTypes = {color: PropTypes.string};
```

Updating Context

This section documents a legacy API. See the [new API](#).

Don't do it.

React has an API to update context, but it is fundamentally broken and you should not use it.

The `getChildContext` function will be called when the state or props changes. In order to update data in the context, trigger a local state update with `this.setState`. This will trigger a new context and changes will be received by the children.

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

class MediaQuery extends React.Component {
  constructor(props) {
    super(props);
    this.state = {type: 'desktop'};
  }

  getChildContext() {
    return {type: this.state.type};
  }

  componentDidMount() {
    const checkMediaQuery = () => {
      const type = window.matchMedia("(min-width: 1025px)").matches ? 'desktop' : 'mobile';
      if (type !== this.state.type) {
        this.setState({type});
      }
    };
    checkMediaQuery();
    window.addEventListener('resize', checkMediaQuery);
  }
}
```

```
};

window.addEventListener('resize', checkMediaQuery);
checkMediaQuery();
}

render() {
  return this.props.children;
}

MediaQuery.childContextTypes = {
  type: PropTypes.string
};
```

The problem is, if a context value provided by component changes, descendants that use that value won't update if an intermediate parent returns `false` from `shouldComponentUpdate`. This is totally out of control of the components using context, so there's basically no way to reliably update the context. [This blog post](#) has a good explanation of why this is a problem and how you might get around it.

[Go to TOC](#)

Faire remonter l'état

Plusieurs composants ont souvent besoin de refléter les mêmes données dynamiques. Nous conseillons de faire remonter l'état partagé dans leur ancêtre commun le plus proche. Voyons comment ça marche.

Dans cette section, nous allons créer un calculateur de température qui détermine si l'eau bout à une température donnée.

Commençons par un composant appelé `BoilingVerdict`. Il accepte une prop `celsius` pour la température, et il affiche si elle est suffisante pour faire bouillir l'eau :

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>L'eau bout.</p>;
  }
  return <p>L'eau ne bout pas.</p>;
}
```

Ensuite, nous allons créer un composant appelé `Calculator`. Il affiche un `<input>` qui permet de saisir une température et de conserver sa valeur dans `this.state.temperature`.

Par ailleurs, il affiche le `BoilingVerdict` pour la température saisie.

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

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

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Saisissez la température en Celsius :</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

[Essayer sur CodePen](#)

Ajouter un deuxième champ

Il nous faut à présent proposer, en plus d'une saisie en Celsius, une saisie en Fahrenheit, les deux devant rester synchronisées.

On peut commencer par extraire un composant `TemperatureInput` du code de `Calculator`. Ajoutons-y une prop `scale` qui pourra être soit `"c"`, soit `"f"` :

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

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

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

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Saisissez la température en {scaleNames[scale]} :</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Nous pouvons désormais modifier le composant `Calculator` pour afficher deux saisies de température :

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

[Essayer sur CodePen](#)

Nous avons maintenant deux champs de saisie, mais lorsque vous saisissez la température dans un des deux, l'autre ne se met pas à jour. Nous avons besoin de les garder synchronisés.

Qui plus est, nous ne pouvons pas afficher le `BoilingVerdict` depuis `Calculator`. Le composant `Calculator` n'a pas accès à la température saisie, car elle est cachée dans le `TemperatureInput`.

Écrire des fonctions de conversion

D'abord, écrivons deux fonctions pour convertir de Celsius à Fahrenheit et réciproquement :

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

Ces deux fonctions convertissent des nombres. Écrivons une autre fonction qui prend en arguments une chaîne de caractères `temperature` et une fonction de conversion, et qui renvoie une chaîne. Nous utiliserons cette nouvelle fonction pour calculer la valeur d'un champ en fonction de l'autre.

Elle renvoie une chaîne vide pour une `temperature` incorrecte, et arrondit la valeur de retour à trois décimales :

```
function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
```

Par exemple, `tryConvert('abc', toCelsius)` renvoie une chaîne vide, et `tryConvert('10.22', toFahrenheit)` renvoie `'50.396'`.

Faire remonter l'état

Pour l'instant, les deux éléments `TemperatureInput` conservent leur propre état local indépendamment l'un de l'autre :

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

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

  render() {
    const temperature = this.state.temperature;
    // ...
  }
}
```

Cependant, nous voulons que les deux champs soient synchronisés. Lorsqu'on modifie le champ en Celsius, celui en Fahrenheit doit refléter la température convertie, et réciproquement.

Avec React, partager l'état est possible en le déplaçant dans le plus proche ancêtre commun. On appelle ça « faire remonter l'état ». Nous allons supprimer l'état local de `TemperatureInput` et le déplacer dans le composant `Calculator`.

Si le composant `Calculator` est responsable de l'état partagé, il devient la « source de vérité » pour la température des deux champs. Il peut leur fournir des valeurs qui soient cohérentes l'une avec l'autre. Comme les props des deux composants `TemperatureInput` viennent du même composant parent `Calculator`, les deux champs seront toujours synchronisés.

Voyons comment ça marche étape par étape.

D'abord, remplaçons `this.state.temperature` par `this.props.temperature` dans le composant `TemperatureInput`. Pour le moment, faisons comme si `this.props.temperature` existait déjà, même si nous allons devoir la passer depuis `Calculator` plus tard :

```
render() {
  // Avant : const temperature = this.state.temperature;
  const temperature = this.props.temperature;
  // ...
```

On sait que [les props sont en lecture seule](#). Quand la `temperature` était dans l'état local, le composant `TemperatureInput` pouvait simplement appeler `this.setState()` pour la changer. Cependant, maintenant que `temperature` vient du parent par une prop, le composant `TemperatureInput` n'a pas le contrôle dessus.

Avec React, on gère généralement ça en rendant le composant « contrôlé ». Tout comme un élément DOM `<input>` accepte des props `value` et `onChange`, notre `TemperatureInput` peut accepter des props `temperature` et `onTemperatureChange` fournies par son parent `Calculator`.

Maintenant, quand le composant `TemperatureInput` veut mettre à jour la température, il appelle `this.props.onTemperatureChange` :

```
handleChange(e) {
  // Avant : this.setState({temperature: e.target.value});
  this.props.onTemperatureChange(e.target.value);
  // ...
```

Remarque

Les noms de props `temperature` et `onTemperatureChange` n'ont pas de sens particulier. On aurait pu les appeler n'importe comment, par exemple `value` et `onChange`, qui constituent une convention de nommage répandue.

La prop `onTemperatureChange` sera fournie par le composant parent `Calculator`, tout comme la prop `temperature`. Elle s'occupera du changement en modifiant son propre état local, entraînant un nouvel affichage des deux champs avec leurs nouvelles valeurs. Nous allons nous pencher très bientôt sur l'implémentation du nouveau composant `Calculator`.

Avant de modifier le composant `Calculator`, récapitulons les modifications apportées au composant `TemperatureInput`. Nous en avons retiré l'état local, et nous lisons désormais `this.props.temperature` au lieu de `this.state.temperature`. Plutôt que d'appeler `this.setState()` quand on veut faire un changement, on appelle désormais `this.props.onTemperatureChange()`, qui est fournie par le `Calculator` :

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Saisissez la température en {scaleNames[scale]} :</legend>
        <input value={temperature}>
          onChange={this.handleChange}
        </input>
      </fieldset>
    );
  }
}
```

Intéressons-nous maintenant au composant `Calculator`.

Nous allons stocker la valeur courante de `temperature` et de `scale` dans son état local. C'est l'état que nous avons « remonté » depuis les champs, et il servira de « source de vérité » pour eux deux. C'est la représentation minimale des données dont nous avons besoin afin d'afficher les deux champs.

Par exemple, si on saisit 37 dans le champ en Celsius, l'état local du composant `Calculator` sera :

```
{
  temperature: '37',
  scale: 'c'
}
```

Si plus tard on change le champ Fahrenheit à 212, l'état local du composant `Calculator` sera :

```
{
  temperature: '212',
  scale: 'f'
}
```

On pourrait avoir stocké les valeurs des deux champs, mais en fait ce n'est pas nécessaire. Stocker uniquement la valeur la plus récente et son unité s'avère suffisant. On peut déduire la valeur de l'autre champ rien qu'à partir des valeurs de `temperature` et de `scale` stockées.

Les champs restent synchronisés car leurs valeurs sont calculées depuis le même état :

```

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
      temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
      temperature;

    return (
      <div>
        <TemperatureInput
          scale="c"
          temperature={celsius}
          onTemperatureChange={this.handleCelsiusChange} />
        <TemperatureInput
          scale="f"
          temperature={fahrenheit}
          onTemperatureChange={this.handleFahrenheitChange} />
        <BoilingVerdict
          celsius={parseFloat(celsius)} />
      </div>
    );
  }
}

```

[Essayer sur CodePen](#)

Désormais, quel que soit le champ que vous modifiez, `this.state.temperature` et `this.state.scale` seront mis à jour au sein du composant `Calculator`. L'un des deux champ recevra la valeur telle quelle, et l'autre valeur de champ sera toujours recalculée à partir de la valeur modifiée.

Récapitulons ce qui se passe quand on change la valeur d'un champ :

- React appelle la fonction spécifiée dans l'attribut `onChange` de l'élément DOM `<input>`. Dans notre cas, c'est la méthode `handleChange` du composant `TemperatureInput`.
- La méthode `handleChange` du composant `TemperatureInput` appelle `this.props.onTemperatureChange()` avec la nouvelle valeur. Ses props, notamment `onTemperatureChange`, ont été fournies par son composant parent, `Calculator`.

- Au dernier affichage en date, le composant `Calculator` avait passé la méthode `handleCelsiusChange` de `Calculator` comme prop `onTemperatureChange` du `TemperatureInput` en Celsius, et la méthode `handleFahrenheitChange` de `Calculator` comme prop `onTemperatureChange` du `TemperatureInput` en Fahrenheit. L'une ou l'autre de ces méthodes de `Calculator` sera ainsi appelée en fonction du champ modifié.
- Dans ces méthodes, le composant `Calculator` demande à React de le rafraîchir en appelant `this.setState()` avec la nouvelle valeur du champ et l'unité du champ modifié.
- React appelle la méthode `render` du composant `Calculator` afin de savoir à quoi devrait ressembler son UI. Les valeurs des deux champs sont recalculées en fonction de la température actuelle et de l'unité active. La conversion de température est faite ici.
- React appelle les méthodes `render` des deux composants `TemperatureInput` avec leurs nouvelles props, spécifiées par le `Calculator`. React sait alors à quoi devraient ressembler leurs UI.
- React appelle la méthode `render` du composant `BoilingVerdict`, en lui passant la température en Celsius dans les props.
- React DOM met à jour le DOM avec le verdict d'ébullition, et retranscrit les valeurs de champs souhaitées. Le champ que nous venons de modifier reçoit sa valeur actuelle, et l'autre champ est mis à jour avec la température convertie.

Chaque mise à jour suit ces mêmes étapes, ainsi les champs restent synchronisés.

Ce qu'il faut retenir

Il ne doit y avoir qu'une seule « source de vérité » pour toute donnée qui change dans une application React. En général, l'état est d'abord ajouté au composant qui en a besoin pour s'afficher. Ensuite, si d'autres composants en ont également besoin, vous pouvez faire remonter l'état dans l'ancêtre commun le plus proche. Au lieu d'essayer de synchroniser les états de différents composants, vous devriez vous baser sur des données qui se propagent [du haut vers le bas](#).

Faire remonter l'état implique d'écrire plus de code générique (*boilerplate code, NdT*) qu'avec une liaison de données bidirectionnelle, mais le jeu en vaut la chandelle, car ça demande moins de travail pour trouver et isoler les bugs. Puisque tout état « vit » dans un composant et que seul ce composant peut le changer, la surface d'impact des bugs est grandement réduite. Qui plus est, vous pouvez implémenter n'importe quelle logique personnalisée pour rejeter ou transformer les saisies des utilisateurs.

Si quelque chose peut être dérivé des props ou de l'état, cette chose ne devrait probablement pas figurer dans l'état. Par exemple, plutôt que de stocker à la fois `celsiusValue` et `fahrenheitValue`, on stocke uniquement la dernière `temperature` modifiée et son unité `scale`. La valeur de l'autre champ peut toujours être calculée dans la méthode `render()` à partir de la valeur de ces données. Ça nous permet de voir ou d'arrondir la valeur de l'autre champ sans perdre la valeur saisie par l'utilisateur.

Quand vous voyez quelque chose qui ne va pas dans l'UI, vous pouvez utiliser les [outils de développement React](#) pour examiner les props et remonter dans l'arborescence des composants jusqu'à trouver le composant responsable de la mise à jour de l'état. Ça vous permet de remonter à la source des bugs :

Enter temperature in Celsius:

Enter temperature in Fahrenheit:

The water would not boil.

The screenshot shows the React DevTools component inspector. At the top, there are several tabs: Elements (selected), React, Console, Sources, Network, Timeline, Profiles, and a more options menu. Below the tabs are three checkboxes: Trace React Updates, Highlight Search, and Use Regular Expressions. The main area displays the component tree under the <Calculator> component. The <Calculator> component has two child components: <TemperatureInput scale="c" temperature="" onTemperatureChange=> and <TemperatureInput scale="f" temperature="" onTemperatureChange=>. These two components have a parent component <BoilingVerdict celsius=null>...</BoilingVerdict>. The entire <Calculator> component is wrapped in a <div> tag. On the right side of the inspector, there are sections for <Calculator> props and state. The props section shows an empty object. The state section shows the scale as "c" and the temperature as an empty string. A blue bar at the bottom of the sidebar is labeled "Calculator".

[Go to TOC](#)

Listes et clés

Tout d'abord, voyons comment transformer des listes en JavaScript.

Dans le code suivant, on utilise la méthode `map()` pour prendre un tableau de nombres et doubler leurs valeurs. On affecte le nouveau tableau retourné par `map()` à une variable `doubled` et on l'affiche dans la console :

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

Ce code affiche `[2, 4, 6, 8, 10]` dans la console.

Avec React, transformer un tableau en une liste d'[éléments](#) est presque identique.

Afficher plusieurs composants

On peut construire des collections d'éléments et [les inclure dans du JSX](#) en utilisant les accolades `{}`.

Ci-dessous, on itère sur le tableau de nombres en utilisant la méthode JavaScript `map()`. On retourne un élément `` pour chaque entrée du tableau. Enfin, on affecte le tableau d'éléments résultant à `listItems` :

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

On inclut tout le tableau `listItems` dans un élément ``, et [on l'affiche dans le DOM](#) :

```
ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

[Essayer sur CodePen](#)

Ce code affiche une liste à puces de nombres entre 1 et 5.

Composant basique de liste

Généralement, on souhaite afficher une liste au sein d'un [composant](#).

On peut transformer l'exemple précédent pour en faire un composant qui accepte un tableau de nombres et produit une liste d'éléments.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
```

```

    );
    return (
      <ul>{listItems}</ul>
    );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);

```

En exécutant ce code, vous obtiendrez un avertissement disant qu'une clé devrait être fournie pour les éléments d'une liste. Une « clé » (*key, NdT*), est un attribut spécial que vous devez inclure quand vous créez une liste d'éléments. Nous verrons pourquoi c'est important dans la prochaine section.

Assignons une `key` aux éléments de notre liste dans `numbers.map()` afin de corriger le problème de clés manquantes.

```

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);

```

[Essayer sur CodePen](#)

Les clés

Les clés aident React à identifier quels éléments d'une liste ont changé, ont été ajoutés ou supprimés. Vous devez donner une clé à chaque élément dans un tableau afin d'apporter aux éléments une identité stable :

```

const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);

```

Le meilleur moyen de choisir une clé est d'utiliser quelque chose qui identifie de façon unique un élément d'une liste parmi ses voisins. Le plus souvent on utilise l'ID de notre donnée comme clé :

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

Quand vous n'avez pas d'ID stable pour les éléments affichés, vous pouvez utiliser l'index de l'élément en dernier recours :

```
const todoItems = todos.map((todo, index) =>
  // Ne faites ceci que si les éléments n'ont pas d'ID stable
  <li key={index}>
    {todo.text}
  </li>
);
```

Nous vous recommandons de ne pas utiliser l'index comme clé si l'ordre des éléments est susceptible de changer. Ça peut avoir un effet négatif sur les performances, et causer des problèmes avec l'état du composant. Vous pouvez lire l'article de Robin Pokorny pour une [explication en profondeur de l'impact négatif de l'utilisation de l'index comme clé](#) (en anglais). Si vous choisissez de ne pas donner explicitement de clé aux éléments d'une liste, React utilisera l'index par défaut.

Si vous voulez en apprendre davantage, consultez cette [explication en profondeur de la raison pour laquelle les clés sont nécessaires](#).

Extraire des composants avec des clés

Les clés n'ont une signification que dans le contexte du tableau qui les entoure.

Par exemple, si on [extrait](#) un composant `ListItem`, on doit garder la clé sur l'élément `<ListItem />` dans le tableau, et non sur l'élément `` dans le composant `ListItem` lui-même.

Exemple : utilisation erronée des clés

```
function ListItem(props) {
  const value = props.value;
  return (
    // Erroné ! Pas la peine de spécifier la clé ici :
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Erroné : la clé doit être spécifiée ici :
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

```
const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Exemple : utilisation correcte des clés

```
function ListItem(props) {
  // Correct ! Pas la peine de spécifier la clé ici :
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correct ! La clé doit être spécifiée dans le tableau.
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

[Essayer sur CodePen](#)

Gardez en tête cette règle simple : chaque élément à l'intérieur d'un appel à `map()` a besoin d'une clé.

Les clés n'ont besoin d'être uniques qu'au sein de la liste

Les clés utilisées dans un tableau doivent être uniques parmi leurs voisins. Cependant, elles n'ont pas besoin d'être globalement uniques. On peut utiliser les mêmes clés dans des tableaux différents :

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
```

```

        {sidebar}
        <hr />
        {content}
    </div>
);
}

const posts = [
];
ReactDOM.render(
    <Blog posts={posts} />,
    document.getElementById('root')
);

```

Essayer sur CodePen

Les clés servent d'indicateur à React mais ne sont pas passées à vos composants. Si vous avez besoin de la même valeur dans votre composant, passez-la dans une prop avec un nom différent :

```

const content = posts.map((post) =>
    <Post
        key={post.id}
        id={post.id}
        title={post.title} />
);

```

Dans l'exemple ci-dessus, le composant `Post` peut accéder à `props.id`, mais pas à `props.key`.

Intégrer `map()` dans du JSX

Dans les exemples précédents, nous déclarions séparément la variable `listItems` pour ensuite l'inclure dans le JSX :

```

function NumberList(props) {
    const numbers = props.numbers;
    const listItems = numbers.map((number) =>
        <ListItem key={number.toString()}
                  value={number} />
    );
    return (
        <ul>
            {listItems}
        </ul>
    );
}

```

JSX permet [d'intégrer des expressions quelconques](#) entre accolades. Nous pouvons donc utiliser `map()` directement dans notre code JSX :

```

function NumberList(props) {
    const numbers = props.numbers;
    return (
        <ul>
            {numbers.map((number) =>
                <ListItem key={number.toString()}
                          value={number} />
            )}
        </ul>
    );
}

```

```
    </ul>
  );
}
```

[Essayer sur CodePen](#)

Ça rend parfois le code plus lisible, mais il faut éviter d'en abuser. Comme avec JavaScript, c'est vous qui décidez quand ça vaut le coup d'extraire l'expression dans une variable pour plus de lisibilité. Gardez en tête que si le corps de `map()` est trop profond ou trop riche, c'est sans doute le signe qu'il faudrait [extraire un composant](#).

Optimiser les performances

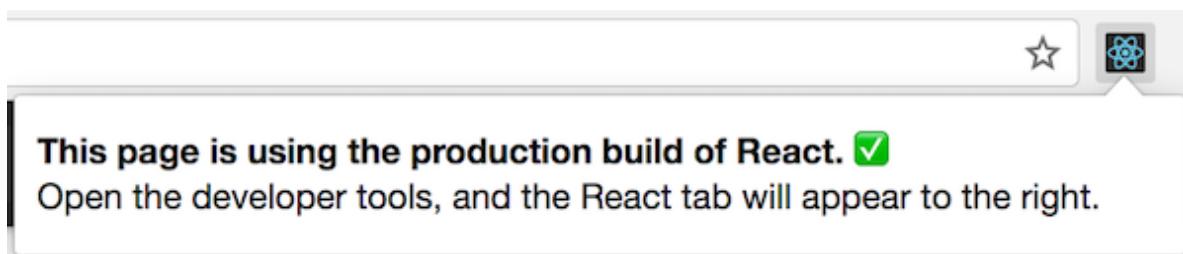
En interne, React fait appel à différentes techniques intelligentes pour minimiser le nombre d'opérations coûteuses sur le DOM nécessaires à la mise à jour de l'interface utilisateur (UI). Pour de nombreuses applications, utiliser React offrira une UI rapide sans avoir à fournir beaucoup de travail pour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

Utiliser la version de production

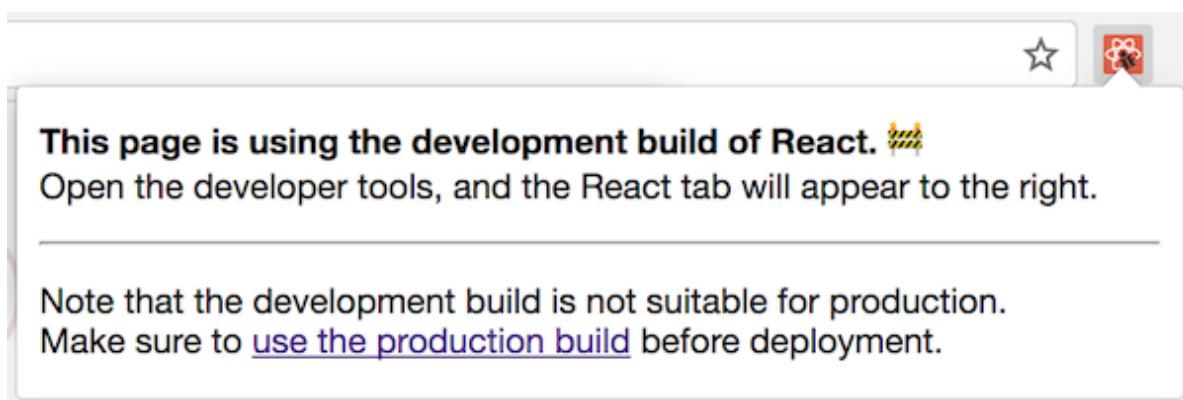
Si vous mesurez ou rencontrez des problèmes de performances dans vos applications React, assurez-vous que vous testez bien la version minifiée de production.

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

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

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

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

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 obfuscué ([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

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 obfuscue 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

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');

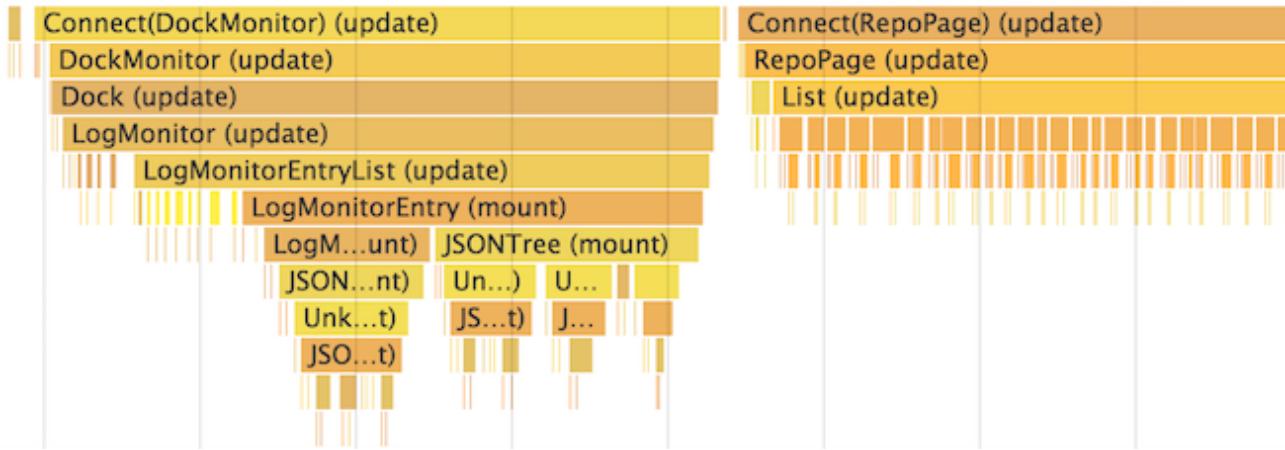
module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

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

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

`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

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

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 noeuds DOM superflus et l'accès excessif aux noeuds 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 noeuds 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 :

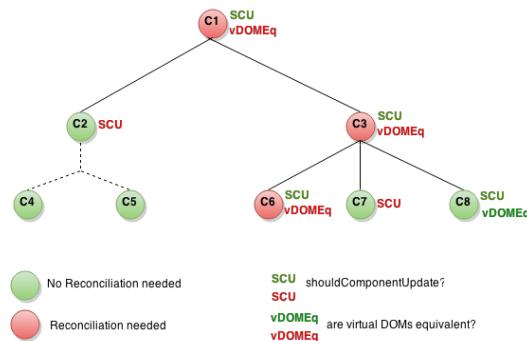
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

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

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

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

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.

Les portails

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

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

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.

React sans ES6

En temps normal, pour définir un composant React, vous utilisez une classe Javascript ordinaire :

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

Si vous n'utilisez pas encore ES6, vous pouvez utiliser le module `create-react-class` :

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function() {
    return <h1>Bonjour, {this.props.name}</h1>;
  }
});
```

À quelques exceptions près, l'API des classes ES6 est similaire à `createReactClass()`.

Déclarer des props par défaut

Dans les fonctions et les classes ES6, `defaultProps` est défini en tant que propriété sur le composant lui-même :

```
class Greeting extends React.Component {
  // ...
}

Greeting.defaultProps = {
  name: 'Marie'
};
```

Avec `createReactClass()`, vous aurez besoin de définir `getDefaults()` en tant que fonction dans l'objet passé en argument :

```
var Greeting = createReactClass({
  getDefaults: function() {
    return {
      name: 'Marie'
    };
  },
  // ...
});
```

Définir l'état initial

Dans les classes ES6, vous pouvez définir l'état local initial en affectant `this.state` dans le constructeur :

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...
}
```

Avec `createReactClass()`, vous devez fournir une méthode `getInitialState` qui renvoie l'état initial :

```
var Counter = createReactClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

Liaison automatique des méthodes

Dans les composants React déclarés en tant que classes ES6, les méthodes suivent la même sémantique que dans toute classe ES6. Ça signifie qu'elles ne vont pas automatiquement lier `this` à l'instance. Pour ce faire, vous devez utiliser `.bind(this)` dans le constructeur :

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Bonjour !'};
    // La ligne ci-dessous est importante !
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
    // Nous pouvons utiliser `this.handleClick` comme gestionnaire d'événements,
    // car il est lié à l'instance courante.
    return (
      <button onClick={this.handleClick}>
        Dis bonjour
      </button>
    );
  }
}
```

Tout ça n'est pas nécessaire avec `createReactClass()`, car elle lie toutes les méthodes :

```
var SayHello = createReactClass({
  getInitialState: function() {
    return {message: 'Bonjour !'};
  },

  handleClick: function() {
    alert(this.state.message);
  },

  render: function() {
```

```

    return (
      <button onClick={this.handleClick}>
        Dis bonjour
      </button>
    );
}
);

```

L'écriture de classes ES6 nécessite un peu plus de code générique pour les gestionnaires d'événements, mais en contrepartie ça améliore légèrement les performances pour les grosses applications.

Si vous n'arrivez pas à tolérer ce code générique, vous pouvez activer dans Babel la proposition de syntaxe **expérimentale Propriétés de classes** :

```

class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Bonjour !'};
  }
  // ATTENTION : cette syntaxe est expérimentale !
  // Recourir à une fonction fléchée va lier la méthode à l'instance :
  handleClick = () => {
    alert(this.state.message);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Dis bonjour
      </button>
    );
  }
}

```

Remarquez bien que la syntaxe ci-dessus est **expérimentale** et qu'elle pourrait être amenée à changer, ou que la proposition de syntaxe ne soit pas intégrée au langage.

Si vous préférez la prudence, quelques options s'offrent à vous :

- Lier les méthodes à l'instance dans le constructeur.
- Utiliser des fonctions fléchées, ex. `onClick={(e) => this.handleClick(e)}`.
- Continuer à utiliser `createClass`.

Mixins

Remarque

ES6 est sorti sans prise en charge des *mixins*. C'est pour cette raison qu'ils ne sont pas pris en charge quand vous utilisez React avec les classes ES6.

Nous avons aussi trouvé pas mal de problèmes dans les bases de code utilisant les mixins, du coup nous recommandons de ne pas les utiliser à l'avenir.

Cette section n'existe qu'à titre de référence.

Des composants très différents les uns des autres peuvent partager des fonctionnalités communes. On parle souvent de [questions transversales](#) (*cross-cutting concerns, NdT*). `createClass` vous permet d'utiliser un système historique de `mixins` pour ça.

Un cas d'usage courant concerne un composant qui veut se mettre à jour à intervalle régulier. C'est facile d'utiliser `setInterval()`, mais il est important de désactiver l'horloge quand vous n'en avez plus besoin afin d'économiser la mémoire. React fournit des [méthodes de cycle de vie](#) qui vous notifient quand un composant est sur le point d'être créé ou détruit. Créons un *mixin* basique qui utilise ces méthodes afin de fournir une fonction `setInterval()` simple d'emploi, qui se nettoiera automatiquement lorsque le composant est détruit.

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var createReactClass = require('create-react-class');

var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // Utiliser le mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Appelle la méthode du mixin
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        Votre composant React tourne depuis {this.state.seconds} secondes.
      </p>
    );
  }
});

ReactDOM.render(
  <TickTock />,
  document.getElementById('example')
);
```

Si un composant utilise plusieurs mixins et que de nombreux mixins définissent la même méthode de cycle de vie (par exemple pour effectuer un nettoyage à la destruction du composant), vous avez la garantie que toutes ces méthodes de cycle de vie seront appelées. Les méthodes définies dans les mixins sont exécutées dans l'ordre dans lequel les mixins ont été listés, suivies par l'appel de la méthode homonyme éventuelle du composant lui-même.

React sans JSX

Vous n'êtes pas obligé·e d'employer JSX pour utiliser React. React sans JSX vous dispense de configurer votre environnement de travail pour gérer la compilation de votre code.

Chaque élément JSX n'est rien de plus que du sucre syntaxique pour `React.createElement(component, props, ...children)`. Par conséquent, tout ce que vous pouvez faire avec JSX peut aussi être réalisé en Javascript brut.

Prenons par exemple ce code écrit avec JSX :

```
class Hello extends React.Component {
  render() {
    return <div>Bonjour ${this.props.toWhat}</div>;
  }
}

ReactDOM.render(
  <Hello toWhat="monde" />,
  document.getElementById('root')
);
```

Il peut être compilé vers ce code qui n'utilise pas JSX :

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

ReactDOM.render(
  React.createElement(Hello, {toWhat: 'monde'}, null),
  document.getElementById('root')
);
```

Si vous voulez voir plus d'exemples de conversion de code JSX en JavaScript brut, vous pouvez essayer [le compilateur Babel en ligne](#).

Le composant peut être soit une chaîne de caractères, soit une sous-classe de `React.Component`, soit une fonction simple.

Si vous n'avez pas envie de taper `React.createElement` à chaque fois, vous pouvez à la place créer un raccourci :

```
const e = React.createElement;

ReactDOM.render(
  e('div', null, 'Bonjour, monde'),
  document.getElementById('root')
);
```

Si vous utilisez un tel raccourci pour `React.createElement`, utiliser React sans JSX devient presque aussi pratique.

Dans le même esprit, vous pouvez aller regarder des projets tels que `react-hyperscript` ou `hyper-script-helpers`, qui proposent une syntaxe encore plus concise.

Réconciliation

React fournit une API déclarative afin que vous n'ayez pas à vous soucier de savoir ce qui change exactement lors de chaque mise à jour. Ça facilite grandement l'écriture d'applications, mais la manière dont React s'y prend n'est pas forcément évidente. Cet article explique les choix que nous avons faits dans l'algorithme de comparaison de façon à rendre prévisibles les mises à jour des composants tout en restant suffisamment rapide pour des applications à hautes performances.

Raisons

Quand vous utilisez React, à chaque instant précis vous pouvez considérer que la fonction `render()` crée un arbre d'éléments React. Lors de la mise à jour suivante de l'état local ou des props, cette fonction `render()` renverra un arbre différent d'éléments React. React doit alors déterminer comment mettre efficacement à jour l'interface utilisateur (UI) pour qu'elle corresponde à l'arbre le plus récent.

Il existe des solutions génériques à ce problème algorithmique consistant à générer le nombre minimal d'opérations pour transformer un arbre en un autre. Néanmoins, [les algorithmes à la pointe de l'état de l'art](#) (en anglais) ont une complexité de l'ordre de $O(n^3)$ où n est le nombre d'éléments dans l'arbre.

Si nous les utilisions dans React, l'affichage de 1 000 éléments nécessiterait environ un milliard d'opérations. C'est beaucoup trop coûteux. React implémente plutôt un algorithme heuristique en $O(n)$ basé sur deux hypothèses :

1. Deux éléments de types différents produiront des arbres différents.
2. Le développeur peut indiquer quels éléments peuvent être stables d'un rendu à l'autre grâce à la prop `key`.

En pratique, ces hypothèses sont valables dans presque tous les cas.

L'algorithme de comparaison

En comparant deux arbres, React va commencer par comparer les éléments racines. Le comportement est différent selon le type des éléments racines.

Éléments de types différents

Chaque fois que les éléments racines ont des types différents, React va détruire l'ancien arbre et reconstruire le nouvel arbre à partir de zéro. Passer de `<a>` à ``, ou de `<Article>` à `<Comment>`, ou de `<Button>` à `<div>` : tous aboutiront à une reconstruction complète.

Lors de la destruction d'un arbre, les anciens noeuds DOM sont détruits. Les instances des composants reçoivent `componentWillUnmount()`. Lors de la construction d'un nouvel arbre, les nouveaux noeuds sont insérés dans le DOM. Les instances de composants reçoivent `UNSAFE_componentWillMount()` puis `componentDidMount()`. Tous les états associés à l'ancien arbre sont perdus.

Tous les composants au-dessous de la racine seront également démontés et leur état détruit. Par exemple, en comparant :

```
<div>
  <Counter />
</div>

<span>
  <Counter />
</span>
```

Ça détrira l'ancien `Counter` puis en remontera un nouveau.

Remarque

Les méthodes suivantes sont considérées dépréciées et vous devriez [les éviter](#) dans vos nouveaux codes :

- `UNSAFE_componentWillMount()`

Éléments DOM de même type

Lors de la comparaison entre deux éléments DOM React de même type, React examine les attributs des deux, conserve le même nœud DOM sous-jacent, et ne met à jour que les attributs modifiés. Par exemple :

```
<div className="before" title="stuff" />
<div className="after" title="stuff" />
```

En comparant ces deux éléments, React sait qu'il ne faut modifier que le `className` du nœud DOM sous-jacent.

Lors d'une mise à jour du `style`, React là aussi sait ne mettre à jour que les propriétés qui ont changé. Par exemple :

```
<div style={{color: 'red', fontWeight: 'bold'}} />
<div style={{color: 'green', fontWeight: 'bold'}} />
```

Lors de la conversion entre les deux éléments, React sait qu'il ne doit modifier que le style `color` et pas `fontWeight`.

Après avoir manipulé le nœud DOM, React applique le même traitement sur les enfants.

Éléments composants de même type

Lorsqu'un composant est mis à jour, l'instance reste la même, afin que l'état soit maintenu d'un rendu à l'autre. React met à jour les props de l'instance de composant sous-jacente pour correspondre au nouvel élément, et appelle `UNSAFE_componentWillReceiveProps()`, `UNSAFE_componentWillUpdate()` et `componentDidUpdate()` dessus.

Ensuite, la méthode `render()` est appelée et l'algorithme de comparaison reprend entre son résultat précédent et le nouveau.

Remarque

Les méthodes suivantes sont considérées dépréciées et vous devriez [les éviter](#) dans vos nouveaux codes :

- `UNSAFE_componentWillReceiveProps()`
- `UNSAFE_componentWillUpdate()`

Traitements récursifs sur les enfants

Par défaut, lorsqu'il traite les enfants d'un nœud DOM, React parcourt simultanément les deux listes d'enfants et génère une modification chaque fois qu'il y a une différence.

Par exemple, lors de l'ajout d'un élément à la fin des enfants, la conversion entre les deux arbres fonctionne bien :

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React fera correspondre les deux arbres `first`, les deux arbres `second`, et insérera l'arbre `third`.

Si vous l'implémentez de façon naïve, l'insertion d'un élément au début aura de moins bonnes performances. Par exemple, la conversion entre ces deux arbres fonctionnera assez mal :

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React va modifier chaque enfant plutôt que de réaliser qu'il pouvait garder les sous-arbres `Duke` et `Villanova` intacts. Cette inefficacité peut être un problème.

Clés

Afin de résoudre ce problème, React prend en charge l'attribut `key`. Quand des enfants ont cette clé, React l'utilise pour faire correspondre les enfants de l'arbre d'origine avec les enfants de l'arbre suivant. Par exemple, l'ajout d'une `key` dans notre exemple inefficace peut rendre la conversion de l'arbre plus efficace :

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

À présent, React sait que l'élément avec la clé `'2014'` est nouveau, et que les éléments avec les clés `'2015'` et `'2016'` ont juste été déplacés.

En pratique, trouver une clé n'est généralement pas difficile. L'élément que vous allez afficher peut déjà disposer d'un identifiant unique, la clé provenant alors de vos données :

```
<li key={item.id}>{item.name}</li>
```

Quand ce n'est pas le cas, vous pouvez ajouter une nouvelle propriété d'identification à votre modèle, ou hacher certaines parties de votre contenu pour générer une clé. La clé n'a besoin d'être unique que parmi ses éléments frères, et non au niveau global.

En dernier recours, vous pouvez utiliser l'index de l'élément dans un tableau comme clé. Cela fonctionne correctement si les éléments ne sont jamais réordonnés et s'il n'y a ni insertion ni suppression, dans le cas contraire ce serait assez lent.

Les tris peuvent également causer des problèmes avec les états des composants quand les index sont utilisés comme des clés. Les instances des composants sont mises à jour et réutilisées en fonction de leur clé. Si la clé est un index, déplacer un élément changera sa clé. En conséquence, l'état local des composants utilisés pour des saisies non-contrôlées peut s'emmêler et être mis à jour de manière inattendue.

Voici [un exemple sur CodePen](#) des problèmes qui peuvent être causés en utilisant des index comme clés. Voilà [une version mise à jour du même exemple montrant comment, en évitant d'utiliser les index comme clés, on résoudra ces problèmes de réordonnement, de tri et d'insertion](#).

Compromis

Rappelez-vous bien que l'algorithme de réconciliation est un détail d'implémentation. React pourrait rafraîchir l'ensemble de l'application à chaque action ; le résultat final serait le même. Pour être clair, rafraîchir dans ce contexte signifie appeler `render` sur tous les composants, ça ne signifie pas que React les démontera et remontera. Il n'appliquera que les différences obtenues en suivant les règles énoncées dans les sections précédentes.

Nous affinons régulièrement les heuristiques afin d'accélérer les cas d'usage courants. Dans l'implémentation actuelle, vous pouvez exprimer le fait qu'un sous-arbre a été déplacé parmi ses frères, mais vous ne pouvez pas dire qu'il a été déplacé ailleurs. L'algorithme va refaire le rendu de l'ensemble du sous-arbre.

Puisque React se repose sur des heuristiques, si les hypothèses derrière celles-ci s'avèrent erronées, ça réduira les performances.

1. L'algorithme n'essaiera pas de faire correspondre des sous-arbres de types de composants différents. Si vous êtes amené·e à alterner entre deux types de composants au rendu très similaire, vous devriez peut-être en faire un type unique. En pratique, nous ne considérons pas ça comme un problème.
2. Les clés doivent être stables, prévisibles et uniques. Des clés instables (comme celles produites par `Math.random()`) entraîneront la re-création superflue de nombreuses instances de composants et de nœuds DOM, ce qui peut dégrader les performances et perdre l'état local des composants enfants.

Éléments DOM

React implémente un système de DOM indépendant des navigateurs pour des raisons de performance et de compatibilité entre navigateurs. Nous en avons profité pour arrondir les angles des implémentations du DOM des navigateurs.

En React, toutes les propriétés et tous les attributs du DOM (y compris les gestionnaires d'événements) doivent être en *camelCase*. Par exemple, l'attribut HTML `tabindex` correspond à l'attribut `tabIndex` en React. Les attributs `aria-*` et `data-*` sont l'exception à la règle, et doivent être en minuscules. Par exemple, vous pouvez garder `aria-label` en tant que `aria-label`.

Différences dans les attributs

Un certain nombre d'attributs diffèrent entre React et HTML :

`checked`

L'attribut `checked` est accepté par les composants `<input>` de type `checkbox` ou `radio`. Vous pouvez l'utiliser pour définir si un composant est coché ou non. C'est utile pour créer des composants contrôlés. L'équivalent non-contrôlé est `defaultChecked`, qui définit l'état coché ou non du composant uniquement lorsqu'il est monté pour la première fois.

`className`

Utilisez l'attribut `className` pour spécifier une classe CSS. Ça vaut pour tous les éléments DOM et SVG tels que `<div>`, `<a>`, et les autres.

Si vous utilisez React avec les Web Components (ce qui est rare), utilisez plutôt l'attribut `class`.

`dangerouslySetInnerHTML`

`dangerouslySetInnerHTML` est l'équivalent React de `innerHTML` dans le DOM des navigateurs. En règle générale, définir le HTML directement depuis le code est risqué car il est trop facile d'exposer vos utilisateurs à une attaque de type [cross-site scripting \(XSS\)](#). C'est pourquoi vous pouvez définir le HTML directement depuis React, mais vous devez taper `dangerouslySetInnerHTML` et passer un objet avec une clé `__html`, pour vous souvenir que c'est dangereux. Par exemple :

```
function createMarkup() {
  return {__html: 'Premier &nbsp; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

`htmlFor`

Puisque `for` est un mot réservé en JavaScript, les éléments React utilisent plutôt `htmlFor`.

onChange

L'événement `onChange` se comporte comme on s'y attend : à chaque fois qu'un champ de formulaire change, cet événement est déclenché. Nous utilisons délibérément un comportement différent de celui des navigateurs car le nom `onChange` est un faux-amis et React s'appuie sur cet événement pour traiter les saisies utilisateurs en temps réel.

selected

Si vous voulez signaler qu'une option est sélectionnée, utilisez plutôt sa valeur dans le `value` de son `<select>`. Pour plus de détails, consultez « [La balise `select`](#) ».

style

Remarque

Certains exemples dans la documentation utilisent `style` par souci de commodité, mais **utiliser l'attribut `style` comme méthode principale pour styler les éléments est généralement déconseillé**. Dans la plupart des cas, vous devriez plutôt utiliser `className` pour référencer des classes définies dans une feuille de style CSS externe. Dans les applications React, on utilise plus fréquemment `style` pour ajouter des styles calculés dynamiquement au moment de l'affichage. Voir également [FAQ : styles et CSS](#).

L'attribut `style` accepte un objet JavaScript avec des propriétés en `camelCase` plutôt qu'une chaîne de caractères CSS. C'est conforme à la propriété JavaScript `style`, plus performant, et évite des failles de sécurité XSS. Par exemple :

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Bonjour, monde !</div>;
}
```

Notez que ces styles ne sont pas automatiquement préfixés. Pour prendre en charge les navigateurs plus anciens vous devez fournir les propriétés de styles correspondantes :

```
const divStyle = {
  WebkitTransition: 'all', // notez le 'W' majuscule ici
  msTransition: 'all' // 'ms' est le seul préfixe fournisseur en minuscules
};

function ComponentWithTransition() {
  return <div style={divStyle}>Ça devrait fonctionner dans tous les
navigateurs</div>;
}
```

Les clés de style sont en `camelCase` pour être cohérentes avec les propriétés des nœuds du DOM (ex. `node.style.backgroundImage`). Les préfixes fournisseurs `hormis` doivent commencer avec une lettre majuscule. C'est pour ça que `WebkitTransition` a un « W » majuscule.

React ajoute automatiquement le suffixe « px » à certaines propriétés numériques de style. Si vous voulez une autre unité que « px », spécifiez la valeur sous forme de chaîne de caractères avec l'unité désirée. Par exemple :

```
// Style résultat : '10px'
<div style={{ height: 10 }}>
  Bonjour, monde !
</div>

// Style résultat : '10%'
<div style={{ height: '10%' }}>
  Bonjour, monde !
</div>
```

Toutes les propriétés de style ne sont pas systématiquement converties en pixels pour autant. Certaines restent sans unité (ex. `zoom`, `order`, `flex`). La liste complète des propriétés sans unité peut-être consultée [ici](#).

suppressContentEditableWarning

Normalement un avertissement apparaît lorsqu'un élément avec des enfants est également marqué comme `contentEditable`, car ça ne fonctionnera pas. Cet attribut supprime cet avertissement. Ne l'utilisez pas à moins que vous ne soyez en train de développer une bibliothèque comme `Draft.js`, qui gère `contentEditable` manuellement.

suppressHydrationWarning

Si vous utilisez le rendu côté serveur de React, normalement un avertissement apparaît lorsque le serveur et le client produisent des contenus différents. Cependant, dans certains cas rares, il est très difficile—voire impossible—de garantir un contenu identique. Par exemple, les horodatages diffèrent généralement entre le serveur et le client.

Si vous définissez `suppressHydrationWarning` à `true`, React ne vous avertira pas des différences dans les attributs et le contenu de cet élément. Ça ne fonctionne qu'à un niveau de profondeur, et c'est conçu comme une échappatoire. N'en abusez pas. Pour en apprendre davantage sur la phase d'hydratation, consultez la [documentation de ReactDOM.hydrate\(\)](#).

value

L'attribut `value` est accepté par les composants `<input>`, `<select>` et `<textarea>`. Vous pouvez l'utiliser pour définir la valeur d'un composant. C'est utile pour créer des composants contrôlés. `defaultValue` est l'équivalent non-contrôlé, qui définit la valeur du composant uniquement lorsqu'il est monté pour la première fois.

Tous les attributs HTML pris en charge

Depuis React 16, tous les attributs standards ou [personnalisés](#) sont pleinement pris en charge.

React a toujours fourni une API de gestion du DOM pensée pour JavaScript. Étant donné que les composants React acceptent autant les props personnalisées que celles liées au DOM, React utilise la convention `camelCase` tout comme les API DOM :

```
<div tabIndex="-1" /> // Tout comme l'API DOM node.tabIndex
<div className="Button" /> // Tout comme l'API DOM node.className
<input readOnly={true} /> // Tout comme l'API DOM node.readOnly
```

Ces props fonctionnent comme les attributs HTML correspondants, à l'exception des cas spéciaux documentés ci-dessus.

Les attributs du DOM pris en charge par React incluent notamment :

```
accept acceptCharset accessKey action allowFullScreen alt autoComplete
autoFocus autoPlay capture cellPadding cellSpacing challenge charSet checked
cite classID className colSpan cols content contentEditable contextMenu controls
controlsList coords crossOrigin data dateTime default defer dir disabled
download draggable encType form formAction formEncType formMethod formNoValidate
formTarget frameBorder headers height hidden high href hrefLang htmlFor
httpEquiv icon id inputMode integrity is keyParams keyType kind label lang list
loop low manifest marginHeight marginWidth max maxLength media mediaGroup method
min minLength multiple muted name noValidate nonce open optimum pattern
placeholder poster preload profile radioGroup readOnly rel required reversed
role rowSpan rows sandbox scope scoped scrolling seamless selected shape size
sizes span spellCheck src srcDoc srcLang srcSet start step style summary
tabIndex target title type useMap value width wmode wrap
```

Dans le même esprit, tous les attributs SVG sont pleinement pris en charge :

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
```

```
 strikethroughPosition strikethroughThickness string stroke strokeDasharray  
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity  
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor  
textDecoration textLength textRendering to transform u1 u2 underlinePosition  
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic  
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY  
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing  
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole  
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlns xmlnsXlink xmlBase  
xmlLang xmlSpace y y1 y2 yChannelSelector z zoomAndPan
```

Vous pouvez également utiliser des attributs personnalisés du moment qu'ils sont intégralement en minuscules.

SyntheticEvent

Ce guide de référence documente l'enrobage `SyntheticEvent` qui fait partie du système d'événements de React. Consultez le guide sur la [gestion d'événements](#) pour en savoir plus.

Aperçu

Vos gestionnaires d'événements recevront des instances de `SyntheticEvent`, un enrobage compatible tous navigateurs autour de l'événement natif du navigateur. Il fournit la même interface que l'événement natif, comprenant notamment `stopPropagation()` et `preventDefault()`, à ceci près que ces événements fonctionnent de façon identique sur tous les navigateurs.

Si pour une raison ou une autre, vous avez besoin de l'événement sous-jacent du navigateur, utilisez l'attribut `nativeEvent` pour le récupérer. Les événements synthétiques diffèrent des événements natifs du navigateur, avec lesquels ils n'ont pas toujours de correspondance directe. Par exemple pour `onMouseLeave`, `event.nativeEvent` référencera un événement `mouseout`. Les correspondances effectives ne font pas partie de l'API publique et sont susceptibles d'évoluer à tout moment. Tous les objets `SyntheticEvent` disposent des attributs suivants :

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```

Remarque

Depuis la version 0.14, renvoyer `false` depuis un gestionnaire d'événements n'interrompt plus la propagation de l'événement. Pour ce faire,appelez explicitement `e.stopPropagation()` ou `e.preventDefault()`, selon le besoin.

Recyclage des événements

Les objets `SyntheticEvent` sont recyclés. En d'autres termes, tout objet `SyntheticEvent` sera réutilisé et ses propriétés seront remises à `null` une fois que la fonction de rappel de l'événement aura été invoquée. React agit ainsi pour améliorer les performances. Par conséquent, vous ne pouvez pas accéder à l'événement de façon asynchrone.

```

function onClick(event) {
  console.log(event); // => objet nullifié.
  console.log(event.type); // => "click"
  const eventType = event.type; // => "click"

  setTimeout(function() {
    console.log(event.type); // => null
    console.log(eventType); // => "click"
  }, 0);

  // Ne fonctionnera pas. this.state.clickEvent ne contiendra que des valeurs à
  // null.
  this.setState({clickEvent: event});

  // Vous pouvez toutefois exporter les propriétés de l'événement.
  this.setState({eventType: event.type});
}

```

Remarque

Si vous souhaitez accéder aux propriétés de l'événement de façon asynchrone, vous devez appeler sa méthode `event.persist()`, ce qui le retirera du système de recyclage, et permettra à votre code de conserver sans problème des références sur l'événement.

Événements pris en charge

React normalise les événements pour qu'ils aient les mêmes propriétés dans tous les navigateurs.

Les gestionnaires d'événements ci-dessous sont déclenchés par un événement durant la phase de propagation. Pour inscrire un gestionnaire d'événements pour la phase de capture, ajoutez le suffixe `Capture` au nom de l'événement ; par exemple, vous utiliserez `onClickCapture` plutôt que `onClick` pour gérer l'événement de clic durant la phase de capture.

- [Événements de presse-papiers](#)
- [Événements de composition](#)
- [Événements du clavier](#)
- [Événements de focus](#)
- [Événements de formulaires](#)
- [Événements génériques](#)
- [Événements de la souris](#)
- [Événements du pointeur](#)
- [Événements de sélection](#)
- [Événements d'interaction tactile](#)
- [Événements visuels](#)
- [Événements de la molette](#)
- [Événements de média](#)
- [Événements d'images](#)
- [Événements d'animation](#)

- Événements de transition
 - Autres événements
-

Référence de l'API

Événements de presse-papiers

Noms des événements :

`onCopy onCut onPaste`

Propriétés :

`DOMDataTransfer clipboardData`

Événements de composition

Noms des événements :

`onCompositionEnd onCompositionStart onCompositionUpdate`

Propriétés :

`string data`

Événements du clavier

Noms des événements :

`onKeyDown onKeyPress onKeyUp`

Propriétés :

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

La propriété `key` peut prendre n'importe quelle valeur documentée dans la [spécification des événements DOM de niveau 3](#).

Événements de focus

Noms des événements :

`onFocus onBlur`

Ces événements de focus fonctionnent sur tous les éléments du DOM de React, et pas uniquement sur les éléments du formulaire.

Propriétés :

`DOMEventTarget relatedTarget`

Événements de formulaires

Noms des événements :

`onChange onInput onInvalid onReset onSubmit`

Pour plus d'informations sur l'événement `onChange`, consultez la documentation sur [les formulaires](#).

Événements génériques

Noms des événements :

`onError onLoad`

Événements de la souris

Noms des événements :

`onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp`

Les événements `onMouseEnter` et `onMouseLeave` se propagent de l'élément qui vient d'être quitté par la souris à celui sur lequel la souris arrive (au lieu d'une propagation classique) et n'ont pas de phase de capture.

Propriétés :

`boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX`

```
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Événements du pointeur

Noms des événements :

```
onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture
onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut
```

Les événements `onPointerEnter` et `onPointerLeave` se propagent de l'élément qui vient d'être quitté par le pointeur à celui sur lequel le pointeur arrive (au lieu d'une propagation classique) et n'ont pas de phase de capture.

Propriétés :

Comme défini par la [spécification W3](#), les événements du pointeur étendent les [événements de la souris](#) avec les propriétés suivantes :

```
number pointerId
number width
number height
number pressure
number tangentialPressure
number tiltX
number tiltY
number twist
string pointerType
boolean isPrimary
```

Une remarque concernant la prise en charge tous navigateurs :

Les événements du pointeur ne sont pas encore pris en charge par tous les navigateurs (au moment de l'écriture de cet article, les navigateurs qui les prennent en charge comprennent Chrome, Firefox, Edge, et Internet Explorer). React ne fournit volontairement pas de *polyfill* pour les autres navigateurs, dans la mesure où un polyfill conforme au standard augmenterait significativement la taille du module `react-dom`.

Si votre application nécessite les événements du pointeur, nous vous conseillons d'ajouter un polyfill tiers pour les prendre en charge.

Événements de sélection

Noms des événements :

```
onSelect
```

Événements d'interaction tactile

Noms des événements :

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

Propriétés :

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

Événements visuels

Noms des événements :

```
onScroll
```

Propriétés :

```
number detail
DOMAbstractView view
```

Événements de la molette

Noms des événements :

```
onWheel
```

Propriétés :

```
number deltaMode
number deltaX
number deltaY
number deltaZ
```

Événements de média

Noms des événements :

```
onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted
onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay
onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend
onTimeUpdate onVolumeChange onWaiting
```

Événements d'images

Noms des événements :

onLoad onError

Événements d'animation

Noms des événements :

onAnimationStart onAnimationEnd onAnimationIteration

Propriétés :

string animationName
string pseudoElement
float elapsedTime

Événements de transition

Noms des événements :

onTransitionEnd

Propriétés :

string propertyName
string pseudoElement
float elapsedTime

Autres événements

Noms des événements :

onToggle

[Go to TOC](#)

Glossaire des termes React

Single-page Application

Une *single-page application* (SPA) est une application qui charge une unique page HTML et toutes les ressources nécessaires (telles que du JavaScript et des CSS) requises pour le fonctionnement de l'application. Aucune interaction sur la page ou les pages ultérieures ne nécessitera un nouvel aller-retour avec le serveur, ce qui signifie que la page n'a pas besoin d'être rafraîchie.

Même si vous pouvez construire une SPA avec React, ce n'est pas obligatoire. React peut aussi être utilisé pour améliorer de petites parties de sites existants en leur apportant une interactivité supplémentaire. Le code écrit en React cohabite paisiblement tant avec le balisage produit par le serveur, au moyen de technologies telles que PHP, qu'avec les autres bibliothèques côté client. En fait, c'est exactement ainsi que React est utilisé chez Facebook.

ES6, ES2015, ES2016, etc.

Ces acronymes font référence aux versions les plus récentes du standard *ECMAScript Language Specification*, dont le langage JavaScript est une implémentation. La version ES6 (également connue sous le nom ES2015) apporte de nombreuses nouveautés par rapport aux versions précédentes, telles que : les fonctions fléchées, les classes, les gabarits de chaînes, les déclarations `let` et `const`... Vous pouvez en apprendre plus sur des versions précises [ici](#).

Compilateurs

Un compilateur JavaScript, souvent appelé *transpileur*, prend du code JavaScript, le transforme et renvoie un code JavaScript de format différent. Le cas d'usage le plus courant consiste à prendre la syntaxe ES6 et à la transformer en une syntaxe que les navigateurs plus anciens sont capables d'interpréter. [Babel](#) est le compilateur le plus utilisé avec React.

Bundlers

Les *bundlers* prennent du code JavaScript et CSS écrit sous forme de modules distincts (souvent par centaines), et les combine pour produire un petit nombre de fichiers plus optimisés pour les navigateurs. Parmi les *bundlers* couramment utilisés pour les applications React, on trouve [Webpack](#) et [Browserify](#).

Gestionnaires de paquets

Les gestionnaires de paquets sont des outils qui vous permettent de gérer les dépendances de votre projet. [npm](#) et [Yarn](#) sont les deux gestionnaires de paquet couramment utilisés pour les applications React. Les deux travaillent en fait avec le même référentiel de modules, géré par npm.

CDN

CDN est l'acronyme de *Content Delivery Network* (*réseau de distribution de contenu*, NdT). Les CDN fournissent des contenus statiques mis en cache via un réseau de serveurs répartis dans le monde entier.

JSX

JSX est une extension syntaxique de JavaScript. C'est un peu comme un langage de gestion de gabarit utilisant du balisage, mais il tire parti de toute la puissance de JavaScript. JSX est compilé en appels à `React.createElement()`, qui renvoient des objets JavaScript nus appelés « éléments React ». Pour découvrir les bases de JSX [consultez cette page de doc](#), et pour l'explorer plus en profondeur vous avez [cette page-ci](#).

React DOM utilise la casse `camelCase` comme convention de nommage des propriétés, au lieu des noms d'attributs HTML. Par exemple, l'attribut `class` devient `className`, dans la mesure où `class` est un mot réservé en JavaScript :

```
const name = 'Clémentine';
ReactDOM.render(
  <h1 className="hello">Je m'appelle {name} !</h1>,
  document.getElementById('root')
);
```

Éléments {#elements}

Les éléments React sont les blocs élémentaires de construction d'applications React. On les confond parfois avec le concept plus répandu de « composants ». Un élément décrit ce que vous voulez voir à l'écran. Les éléments React sont immuables.

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

Habituellement, les éléments ne sont pas utilisés directement, mais renvoyés par les composants.

Composants {#components}

Les composants React sont de petits morceaux de code réutilisables qui renvoient un élément React à afficher dans la page. Dans sa forme la plus simple, un composant React est une bête fonction JavaScript qui renvoie un élément React :

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

Les composants peuvent aussi être basés sur une classe ES6 :

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

Les composants peuvent être scindés en blocs fonctionnels distincts, et utilisés dans d'autres composants. Les composants peuvent renvoyer d'autres composants, des tableaux, des chaînes de caractères et des nombres. Pour décider si une partie de l'UI ferait un bon candidat pour un composant réutilisable, il suffit de se demander si elle apparaît plusieurs fois (`Button`, `Panel`, `Avatar`), ou si elle est suffisamment complexe en elle-même (`App`, `FeedStory`, `Comment`). Les noms de composants devraient toujours démarrer par une majuscule (`<Wrapper/>` pas `<wrapper/>`). Consultez [cette documentation](#) pour de plus amples informations sur le rendu des composants.

`props` `{#props}`

Les `props` sont les entrées d'un composant React. Elles sont passées d'un composant parent à un composant enfant.

Gardez à l'esprit que les `props` sont en lecture seule. Elles ne doivent jamais être modifiées :

```
// Erroné !
props.number = 42;
```

Si vous avez besoin de modifier une valeur en réaction à une saisie utilisateur ou à une réponse réseau, utilisez plutôt `state`.

`props.children` `{#propschildren}`

`props.children` est disponible dans chaque composant. Elle référence le contenu présent entre les balises ouvrante et fermante du composant. Par exemple :

```
<Welcome>Bonjour monde !</Welcome>
```

Le texte `Bonjour monde !` est présent dans la `props.children` du composant `Welcome` :

```
function Welcome(props) {
  return <p>{props.children}</p>;
}
```

Pour les composants définis par des classes, on utilise `this.props.children` :

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

`state` `{#state}`

Un composant a besoin d'un `state` lorsque des données qui lui sont associées évoluent dans le temps. Par exemple, un composant `Checkbox` pourrait avoir besoin d'une donnée d'état local `isChecked`, et un composant `NewsFeed` pourrait vouloir suivre la progression de `fetchedPosts` dans son état local.

La principale différence entre `state` et `props`, c'est que les `props` sont passées depuis le composant parent, alors que `state` est géré en interne par le composant lui-même. Un composant ne peut pas changer ses `props`, mais il peut changer son `state`.

Pour chaque petite donnée qui va changer, un seul composant devrait la « posséder » dans son état local. N'essayez pas de synchroniser les états de plusieurs composants. Préférez [le faire remonter](#) dans leur plus proche ancêtre commun, et faire redescendre l'info via les props aux composants concernés.

Méthodes de cycle de vie {#lifecycle-methods}

Les méthodes de cycle de vie permettent d'implémenter des traitements personnalisés lors des différentes phases d'un composant. Il en existe pour les phases de création et d'insertion dans le DOM ([montage](#)), de mise à jour du composant, et lorsque le composant est démonté et retiré du DOM.

Composants contrôlés ou non-contrôlés

React a deux approches pour la gestion des saisies de formulaire.

Un champ de formulaire dont la valeur est contrôlée par React est appelé un *composant contrôlé*. Quand un utilisateur saisit des données dans un composant contrôlé, un gestionnaire d'événements de modification est déclenché dans votre code, qui décide si la saisie est valide (en ré-affichant le composant avec la valeur mise à jour). Si vous ne ré-affichez pas le champ, sa valeur reste inchangée.

Un *composant non-contrôlé* fonctionne comme les champs habituels non gérés par React. Quand un utilisateur saisit des données dans un champ de formulaire (une zone de texte, une liste déroulante, etc.) l'information mise à jour est reflétée sans que React ait quoi que ce soit à faire. En revanche, cela signifie aussi que vous ne pouvez pas, à la volée, contraindre ou formater les données saisies.

Dans la plupart des cas, vous devriez utiliser des champs contrôlés.

Clés {#keys}

Une « clé » (`key`) est un attribut spécial que vous devez ajouter quand vous créez des tableaux d'éléments. Les clés aident React à identifier quels éléments ont changé, ont été ajoutés ou retirés. Les clés sont à placer sur les éléments du tableau pour leur donner une identité stable.

Les clés doivent être uniques au sein du tableau, mais n'ont pas besoin de l'être pour toute l'application, ou même au sein du composant.

N'utilisez pas quelque chose comme `Math.random()` pour vos clés. Il est critique que les clés aient une « identité stable » d'un rendu à l'autre, pour que React puisse détecter les ajouts, suppressions ou réordonnancements. Dans l'idéal, les clés devraient correspondre à des identifiants stables issus de vos données, comme `post.id`.

Refs {#refs}

React propose un attribut spécial que vous pouvez utiliser pour n'importe quel composant. L'attribut `ref` peut être soit un objet créé par la fonction `React.createRef()`, soit une fonction de rappel, soit encore une chaîne (mais cette dernière forme est dépréciée). Quand l'attribut `ref` est une fonction de rappel, celle-ci reçoit comme argument l'élément du DOM ou l'instance de classe sous-jacents (suivant le type de l'élément). Ça vous permet d'obtenir un accès direct à l'élément du DOM ou à l'instance du composant.

N'abusez pas des refs. Si vous vous retrouvez à souvent recourir à des refs pour « faire que ça marche », vous avez probablement besoin de réviser le [flux de données unidirectionnel](#) de React.

Événements {#events}

La gestion des événements dans React présente quelques différences de syntaxe dans le balisage :

- Les gestionnaires d'événements React sont nommés en `casseCamel`, plutôt qu'en minuscules.
- En JSX, on passe une fonction comme gestionnaire, pas une chaîne de caractères.

Réconciliation {#reconciliation}

Quand les props ou l'état local d'un composant changent, React détermine si une mise à jour effective du DOM est nécessaire en comparant les éléments fraîchement renvoyés et ceux de la passe précédente. Lorsqu'ils ne sont pas égaux, React met à jour le DOM. Ce processus est appelé « réconciliation ».

Pré-requis pour l'environnement JavaScript

React 16 dépend des types de collections `Map` et `Set`. Si vous devez supporter des navigateurs et des appareils plus anciens, qui ne supportent pas ces objets natifs (par exemple IE < 11) ou dont l'implémentation n'est pas conforme aux standards (par exemple IE 11), vous pouvez envisager d'inclure un polyfill global dans votre application, comme `core-js` ou `babel-polyfill`.

L'utilisation d'un polyfill pour supporter React 16 sur les anciens navigateurs internet, grâce à `core-js`, peut ressembler à ceci :

```
import 'core-js/es/map';
import 'core-js/es/set';

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

ReactDOM.render(
  <h1>Bonjour, monde !</h1>,
  document.getElementById('root')
);
```

React dépend aussi de `requestAnimationFrame` (y compris pour les environnements de test).

Vous pouvez utiliser le paquet `raf` pour émuler `requestAnimationFrame` :

```
import 'raf/polyfill';
```

API du profileur

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

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 enveloppée « 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...

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 mémoisation (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.

PureRenderMixin

Note

The `PureRenderMixin` mixin predates `React.PureComponent`. This reference doc is provided for legacy purposes, and you should consider using `React.PureComponent` instead.

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
var createReactClass = require('create-react-class');
createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Example using ES6 class syntax:

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

React.Component

Cette page fournit une référence détaillée de l'API pour la définition de composants React à base de classes. Elle suppose que vous êtes à l'aise avec les concepts fondamentaux de React, tels que les [Composants et props](#), ainsi que l'[État et cycle de vie](#). Si ce n'est pas le cas, lisez ces pages avant de continuer.

Aperçu de l'API

React vous permet de définir vos composants en tant que classes ou fonctions. Les composants définis par des classes fournissent pour le moment davantage de fonctionnalités, qui sont décrites en détail dans cette page. Pour définir un composant React avec une classe, vous devez étendre `React.Component` :

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

La seule méthode que vous devez définir dans une sous-classe de `React.Component` s'appelle `render()`. Toutes les autres méthodes décrites sur cette page sont optionnelles.

Nous vous recommandons fortement de ne pas créer vos propres classes de base pour vos composants. Dans les composants React, la réutilisation de code est obtenue principalement par composition plutôt que par héritage.

Remarque

React ne vous force pas à utiliser la syntaxe de classes ES6. Si vous préférez l'éviter, vous pouvez utiliser à la place le module `create-react-class` ou une autre abstraction maison du même genre. Allez faire un tour sur [React sans ES6](#) pour en apprendre davantage.

Le cycle de vie du composant

Chaque composant a plusieurs « méthodes de cycle de vie » que vous pouvez surcharger pour exécuter du code à des moments précis du processus. **Vous pouvez utiliser ce diagramme de cycle de vie comme aide-mémoire.** Dans la liste ci-dessous, les méthodes de cycle de vie courantes sont indiquées en **gras**. Les autres sont là pour des cas d'utilisation relativement rares.

Montage

Les méthodes suivantes sont appelées dans cet ordre lorsqu'une instance d'un composant est créée puis insérée dans le DOM :

- `constructor()`
- `static getDerivedStateFromProps()`

- `render()`
- `componentDidMount()`

Remarque

Les méthodes suivantes sont considérées dépréciées et vous devriez [les éviter](#) dans vos nouveaux codes :

- `UNSAFE_componentWillMount()`

Mise à jour

Une mise à jour est déclenchée par des changements dans les props ou l'état local. Les méthodes suivantes sont appelées dans cet ordre quand un composant se rafraîchit :

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

Remarque

Les méthodes suivantes sont considérées dépréciées et vous devriez [les éviter](#) dans vos nouveaux codes :

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

Démontage

La méthode suivante est appelée quand un composant est retiré du DOM :

- `componentWillUnmount()`

Gestion d'erreurs

Ces méthodes sont appelées lorsqu'une erreur survient au sein de n'importe quel composant enfant lors de son rendu, dans une méthode de cycle de vie, ou dans son constructeur.

- `static getDerivedStateFromError()`
- `componentDidCatch()`

Autres API

Chaque composant fournit par ailleurs quelques API supplémentaires :

- `setState()`
- `forceUpdate()`

Propriétés de classes

- `defaultProps`
- `displayName`

Propriétés d'instances

- `props`
 - `state`
-

Référence

Méthodes de cycle de vie couramment utilisées

Les méthodes de cette section couvrent la vaste majorité des cas d'utilisation que vous rencontrerez en créant des composants React. **Pour une référence visuelle, jetez un œil à ce diagramme de cycle de vie.**

`render()` {#render}

render()

La méthode `render()` est la seule méthode requise dans une classe de composant.

Lorsqu'elle est appelée, elle examine en général `this.props` et `this.state` et renvoie un des types suivants :

- **Éléments React.** Typiquement créés via [JSX](#). Par exemple, `<div />` et `<MyComponent />` sont des éléments React qui demandent à React de produire, respectivement, un nœud DOM et un autre composant défini par l'utilisateur.
- **Tableaux et fragments.** Ils vous permettent de renvoyer plusieurs éléments racines depuis un rendu. Consultez la documentation des [fragments](#) pour plus de détails.
- **Portails.** Ils permettent d'effectuer le rendu des enfants dans une autre partie du DOM. Consultez la documentation des [portails](#) pour plus de détails.
- **Chaines de caractères et nombres.** Ils deviennent des nœuds textuels dans le DOM.
- **Booléens ou null.** Ils ne produisent rien. (Ça existe principalement pour permettre des motifs de code tels que `return test && <Child />`, où `test` serait booléen.)

La fonction `render()` doit être pure, c'est-à-dire qu'elle ne doit rien changer à l'état local du composant, doit renvoyer le même résultat chaque fois qu'elle est invoquée (dans des conditions identiques), et ne doit pas interagir directement avec le navigateur.

Si vous avez besoin de telles interactions, faites-le plutôt dans `componentDidMount()` ou d'autres méthodes de cycle de vie. S'assurer que `render()` reste pure facilite la compréhension du fonctionnement des composants.

Remarque

```
render() ne sera pas appelée si shouldComponentUpdate() renvoie false.
```

constructor() {#constructor}

```
constructor(props)
```

Si vous n'initialisez pas d'état local et ne liez pas de méthodes, vous n'avez pas besoin d'implémenter votre propre constructeur pour votre composant React.

Le constructeur d'un composant React est appelé avant que celui-ci soit monté. Quand on implémente le constructeur d'une sous-classe de `React.Component`, il faut commencer par appeler `super(props)` avant toute manipulation de `this`. Dans le cas contraire, outre une éventuelle erreur de syntaxe ES6, `this.props` sera `undefined` dans le constructeur, ce qui peut causer des bugs.

Les constructeurs React sont habituellement utilisés pour deux raisons seulement :

- Initialiser l'état local en affectant un objet à `this.state`.
- Lier des méthodes gestionnaires d'événements à l'instance.

Vous ne devez pas appeler `setState()` dans le `constructor()`. Au lieu de ça, si votre composant a besoin d'utiliser l'état local, affectez directement l'état initial à `this.state` dans le constructeur :

```
constructor(props) {
  super(props);
  // Nappelez pas `this.setState()` ici !
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

Le constructeur est le seul endroit où vous devriez affecter directement une valeur à `this.state`. Dans toutes les autres méthodes, utilisez plutôt `this.setState()`.

Évitez aussi de mettre en place des effets de bord ou abonnements dans le constructeur. Dans ces cas-là, préférez `componentDidMount()`.

Remarque

Évitez de copier les props dans l'état local ! C'est une erreur courante :

```
constructor(props) {
  super(props);
  // Ne faites pas ça !
  this.state = { color: props.color };
}
```

Le problème est double : d'une part, c'est superflu (vous pouvez plutôt utiliser directement `this.props.color`), d'autre part ça crée des bugs (les mises à jour de la prop `color` ne seront pas reflétées dans l'état local).

N'utilisez cette approche que si vous avez l'intention d'ignorer les mises à jour de la prop.

Dans un tel cas, il serait judicieux de renommer la `prop` vers quelque chose comme `initialColor` ou `defaultColor`. Vous pouvez ensuite forcer le composant à « réinitialiser » son état interne en `changeant sa key` lorsque c'est nécessaire.

Pour apprendre quelle approche utiliser lorsque vous pensez qu'une partie de votre état doit dépendre des props, lisez notre [article sur l'art d'éviter un état dérivé](#).

componentDidMount() {#componentdidmount}

componentDidMount()

`componentDidMount()` est appelée immédiatement après que le composant est monté (inséré dans l'arbre). C'est ici que vous devriez placer les initialisations qui requièrent l'existence de nœuds du DOM. Si vous avez besoin de charger des données depuis un point d'accès distant, c'est aussi le bon endroit pour déclencher votre requête réseau.

C'est enfin là que vous devriez mettre en place vos abonnements. Si vous en faites, n'oubliez pas de vous désabonner dans `componentWillUnmount()`.

Vous **avez le droit d'appeler `setState()` directement** dans `componentDidMount()`. Ça déclenchera un rendu supplémentaire, mais il surviendra avant que le navigateur ne mette à jour l'écran. Ainsi, vous avez la garantie que même si `render()` est appelée deux fois dans ce cas précis, l'utilisateur ne verra pas l'état intermédiaire. Utilisez toutefois cette façon de faire avec précaution parce qu'elle nuit souvent à la performance. Dans la plupart des cas, vous devriez plutôt pouvoir initialiser l'état local dans le `constructor()`. Ceci étant dit, dans certains cas tels que les boîtes de dialogues et infobulles, qui ont souvent besoin de mesurer un nœud du DOM avant d'afficher quelque chose qui dépend de leur taille ou de leur position, ce second rendu peut s'avérer nécessaire.

componentDidUpdate() {#componentdidupdate}

componentDidUpdate(prevProps, prevState, snapshot)

`componentDidUpdate()` est appelée immédiatement après que la mise à jour a eu lieu. Cette méthode n'est pas appelée pour le rendu initial.

Elle vous donne l'opportunité de travailler sur le DOM une fois que le composant a été mis à jour. C'est aussi un bon endroit pour faire des requêtes réseau, du moment que vous prenez soin de vérifier que les props actuelles concernées diffèrent des anciennes props (dans le sens où une requête réseau est peut-être superflue si les props en question n'ont pas changé).

```
componentDidUpdate(prevProps) {
  // Utilisation classique (pensez bien à comparer les props) :
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

Vous **avez le droit d'appeler `setState()` directement** dans `componentDidUpdate()` mais notez bien que **vous devez l'enrober dans une condition**, comme dans l'exemple ci-dessus, ou vous obtiendrez l'équivalent d'une boucle infinie. Là aussi, vous déclencherez un rendu supplémentaire qui, même s'il n'est pas perceptible par l'utilisateur, peut affecter la performance du composant. Si vous essayez de « refléter » dans l'état local une prop venant de plus haut, voyez si vous ne pouvez pas plutôt utiliser directement la prop. Vous pouvez en apprendre davantage sur [les raisons pour lesquelles copier des props dans l'état local est source de bugs](#).

Si votre composant implémente la méthode de cycle de vie `getSnapshotBeforeUpdate()` (ce qui est rare), la valeur renournée par celle-ci sera passée comme troisième argument `snapshot` à `componentDidUpdate()`. Dans le cas inverse, cet argument sera `undefined`.

Remarque

`componentDidUpdate()` ne sera pas appelée si `shouldComponentUpdate()` renvoie `false`.

`componentWillUnmount()` [\[#componentwillunmount\]](#)

`componentWillUnmount()`

`componentWillUnmount()` est appelée immédiatement avant qu'un composant soit démonté ou détruit. Mettez-y tout nettoyage nécessaire, tel que l'invalidation de minuteurs, l'annulation de requêtes réseau, ou la résiliation d'abonnements effectués dans `componentDidMount()`.

Vous **ne devez pas appeler `setState()` dans `componentWillUnmount()`** car le composant ne sera de toutes façons jamais ré-affiché. Une fois l'instance du composant démontée, elle ne sera plus jamais remontée.

Méthodes de cycle de vie rarement utilisées

Les méthodes de cette section sont liées à des cas d'utilisation peu fréquents. Elles peuvent s'avérer utiles de temps en temps, mais la plupart de vos composants n'en auront sans doute jamais besoin. **Vous pouvez voir la plupart de ces méthodes dans ce diagramme de cycle de vie en cochant la case « Montrer les cycles de vie moins courants » au-dessus.**

`shouldComponentUpdate()` [\[#shouldcomponentupdate\]](#)

`shouldComponentUpdate(nextProps, nextState)`

Utilisez `shouldComponentUpdate()` pour indiquer à React que la sortie d'un composant n'est pas affectée par la modification en cours de l'état local ou des props. Le comportement par défaut consiste à rafraîchir à chaque modification, et pour la vaste majorité des cas vous devriez en rester là.

`shouldComponentUpdate()` est appelée avant le rendu quand de nouvelles props ou un nouvel état local sont reçues. Par défaut, elle renvoie `true`. Cette méthode n'est pas appelée avant le rendu initial ou lorsque `forceUpdate()` est utilisée.

Cette méthode n'existe qu'en tant qu'**optimisation de performance**. Ne vous en servez pas pour « empêcher » un rafraîchissement, car ça finirait par causer des bugs. **Utilisez alors plutôt la classe de base pré définie `PureComponent`**. Celle-ci effectue une comparaison de surface des props et de l'état local, ce qui réduit les risques de sauter une mise à jour nécessaire.

Si vous êtes certain·e de vouloir l'écrire à la main, vous pouvez comparer `this.props` avec `nextProps` et `this.state` avec `nextState`, et renvoyer `false` pour indiquer à React de sauter la mise à jour. Remarquez que renvoyer `false` n'empêche pas les composants fils de se rafraîchir quand *leur* état change.

Nous vous déconseillons de recourir à une comparaison profonde ou à `JSON.stringify()` dans `shouldComponentUpdate()`. Ce sont des techniques coûteuses qui dégraderont les performances.

À l'heure actuelle, si `shouldComponentUpdate()` renvoie `false`, alors `UNSAFE_componentWillUpdate()`, `render()`, et `componentDidUpdate()` ne seront pas appelées. À l'avenir React se réserve le droit de considérer `shouldComponentUpdate()` comme étant « consultatif » plutôt que contraignant, de sorte que renvoyer `false` pourrait tout de même aboutir à un rafraîchissement du composant.

`static getDerivedStateFromProps() {#static-getderivedstatefromprops}`

```
static getDerivedStateFromProps(props, state)
```

`getDerivedStateFromProps` est appelée juste avant d'appeler la méthode `render()`, aussi bien pour le montage initial que lors des mises à jour ultérieures. Elle doit renvoyer un objet qui mette à jour l'état, ou `null` faute de mise à jour.

Cette méthode existe pour **les rares cas** où l'état dépend bien des changements de props au fil du temps. Par exemple, elle peut être utile pour implémenter un composant `<Transition>` qui comparerait les enfants précédents et suivants pour décider lesquels animer en entrée et en sortie.

Dériver l'état entraîne généralement des composants au code verbeux et difficile à suivre. [Assurez-vous qu'une meilleure alternative n'existe pas](#) :

- Si vous avez besoin de **réaliser un effet de bord** (par exemple, charger des données ou dérouler une animation) en réponse à une modification des props, utilisez plutôt la méthode de cycle de vie `componentDidUpdate`.
- Si vous voulez **recalculer des données seulement quand une prop change**, [utilisez plutôt un utilitaire de mémoïsation](#).

- Si vous souhaitez « **réinitialiser** » une partie de l'état local quand une prop change, voyez s'il ne serait pas plutôt judicieux de rendre le composant **pleinement contrôlé** ou **pleinement non-contrôlé** avec une key .

Cette méthode n'a pas accès à l'instance de composant. Si vous le souhaitez, vous pouvez réutiliser du code entre `getDerivedStateFromProps()` et les autres méthodes de la classe en extrayant des fonctions pures appelées avec les props et l'état local du composant, pour les placer hors de la définition de la classe.

Remarquez que cette méthode est déclenchée avant *chaque* rendu, quelle qu'en soit la cause. C'est le contraire de `UNSAFE_componentWillReceiveProps`, qui n'est déclenchée que lorsque votre composant parent vous rafraîchit, et non comme résultat d'un `setState()` local.

`getSnapshotBeforeUpdate() { #getsnapshotbeforeupdate }`

`getSnapshotBeforeUpdate(prevProps, prevState)`

`getSnapshotBeforeUpdate()` est appelée juste avant que le rendu le plus récent ne soit validé, par exemple envoyé au DOM. Elle vous permet de capturer des infos du DOM courant (ex. la position de défilement) avant qu'il ne subisse d'éventuelles modifications. Toute valeur renvoyée par cette méthode de cycle de vie sera passée comme argument à `componentDidUpdate()`.

C'est un cas d'utilisation peu commun, mais il peut survenir dans des UI comme une discussion en ligne qui a besoin de gérer la position de défilement d'une façon spécifique.

Il faut renvoyer une valeur capturée (ou `null`).

Par exemple :

```
embed:react-component-reference/get-snapshot-before-update.js
```

Dans l'exemple ci-dessus, il est important qu'on puisse lire la propriété `scrollHeight` dans `getSnapshotBeforeUpdate`, parce qu'il peut y avoir un délai entre les méthodes de cycle de vie de la « phase de rendu » (comme `render`) et celles de la « phase de commit » (comme `getSnapshotBeforeUpdate` et `componentDidUpdate`).

Périmètres d'erreurs

Les **périmètres d'erreurs** (*error boundaries, NdT*) sont des composants React qui interceptent toute erreur JavaScript survenant dans l'arbre de composants de leurs enfants, loguent ces erreurs, et affichent une UI de remplacement au lieu de l'arbre de composants qui a planté.

Les périmètres d'erreurs capturent les erreurs survenant dans le rendu, les méthodes de cycle de vie, et les constructeurs de tout l'arbre en-dessous d'eux.

Un composant basé classe devient un périmètre d'erreur s'il définit au moins une des méthodes de cycle de vie `static getDerivedStateFromError()` ou `componentDidCatch()`. Mettre à jour votre état local au sein de ces méthodes vous permet d'intercepter une erreur JavaScript non gérée dans l'arbre en-dessous de vous, et d'afficher à la place une UI de remplacement.

N'utilisez les périmètres d'erreurs que pour retomber sur vos pieds lors d'exceptions inattendues ; **ne les utilisez pas comme primitives de contrôle de flux.**

Pour en apprendre davantage, lisez [Gestion d'Erreurs dans React 16](#).

Remarque

Les périmètres d'erreurs n'interceptent que les erreurs dans les composants **en-dessous** d'eux dans l'arbre. Un périmètre d'erreur ne peut pas capturer une erreur survenue en son propre sein.

`static getDerivedStateFromError() {#static-getderivedstatefromerror}`

`static getDerivedStateFromError(error)`

Cette méthode de cycle de vie est appelée après qu'une erreur a été levée par un composant descendant. Elle reçoit l'erreur levée comme paramètre et doit renvoyer une valeur qui mette à jour l'état.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // On met à jour l'état afin que le prochain rendu affiche
    // l'UI de remplacement.
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // Vous pouvez afficher ici n'importe quelle UI de secours
      return <h1>Ça sent le brûlé.</h1>;
    }

    return this.props.children;
  }
}
```

Remarque

`getDerivedStateFromError()` est appelée pendant la « phase de rendu », de sorte que les effets de bord y sont interdits. Si vous en avez besoin, utilisez plutôt `componentDidCatch()`.

`componentDidCatch() {#componentdidcatch}`

`componentDidCatch(error, info)`

Cette méthode de cycle de vie est appelée après qu'une erreur a été levée par un composant descendant. Elle reçoit deux arguments :

1. `error` - L'erreur qui a été levée.
2. `info` - Un objet avec une propriété `componentStack` contenant des informations sur le composant qui a levé l'erreur.

`componentDidCatch()` est appelée durant la « phase de commit », donc les effets de bord y sont autorisés. On peut l'utiliser par exemple pour loguer les erreurs :

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // On met à jour l'état afin que le prochain rendu affiche
    // l'UI de remplacement.
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // Exemple de `componentStack` :
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    logComponentStackToMyService(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // Vous pouvez afficher ici n'importe quelle UI de secours
      return <h1>Ça sent le brûlé.</h1>;
    }

    return this.props.children;
  }
}
```

Remarque

Si une erreur survient, vous pouvez déclencher l'affichage d'une UI de remplacement avec `componentDidCatch()` en appelant `setState`, mais une future version de React dépréciera cette possibilité. Utilisez plutôt `static getDerivedStateFromError()` pour déclencher l'affichage de l'UI de remplacement.

Méthodes de cycle de vie dépréciées

Les méthodes de cycle de vie ci-dessous sont « historiques » (*legacy, NdT*), et à ce titre sont considérées comme dépréciées. Elles fonctionnent encore, mais nous les déconseillons dans tout nouveau code. Vous pouvez en apprendre davantage sur la bonne façon de faire migrer ces méthodes dans [cet article sur notre blog](#).

`UNSAFE_componentWillMount()` **{#unsafe_componentwillmount}**

`UNSAFE_componentWillMount()`

Remarque

Cette méthode de cycle de vie s'appelait à l'origine `componentWillMount`. Ce nom continuera à fonctionner jusqu'à la sortie de React 17. Utilisez le [codemod rename-unsafe-lifecycles](#) pour mettre à jour automatiquement vos composants.

`UNSAFE_componentWillMount()` est appelée juste avant que le montage n'ait lieu. Elle est appelée avant `render()`, de sorte qu'appeler `setState()` de façon synchrone au sein de cette méthode ne déclenchera pas un rendu supplémentaire. Mais généralement, nous conseillons plutôt d'utiliser `constructor()` pour initialiser l'état.

Évitez d'introduire des effets de bord ou d'effectuer des abonnements dans cette méthode. Pour de tels besoins, utilisez plutôt `componentDidMount()`.

C'est la seule méthode de cycle de vie appelée lors d'un rendu côté serveur.

`UNSAFE_componentWillReceiveProps()` **{#unsafe_componentwillreceiveprops}**

`UNSAFE_componentWillReceiveProps(nextProps)`

Remarque

Cette méthode de cycle de vie s'appelait à l'origine `componentWillReceiveProps`. Ce nom continuera à fonctionner jusqu'à la sortie de React 17. Utilisez le [codemod rename-unsafe-lifecycles](#) pour mettre à jour automatiquement vos composants.

Attention

Utiliser cette méthode de cycle de vie aboutit le plus souvent à des bugs de cohérence.

- Si vous aviez besoin de **réaliser un effet de bord** (par exemple, charger des données ou dérouler une animation) en réponse à une modification des props, utilisez plutôt la méthode de cycle de vie `componentDidUpdate`.
- Si vous utilisez `componentWillReceiveProps` pour **recalculer des données seulement quand une prop change**, utilisez plutôt un utilitaire de mémoïsation.
- Si vous utilisez `componentWillReceiveProps` pour « **réinitialiser** » **une partie de l'état local quand une prop change**, voyez s'il ne serait pas plutôt judicieux de rendre le composant **pleinement contrôlé** ou **pleinement non-contrôlé avec une key**.

Pour les autres cas, [suivez les recommandations de notre article de blog sur la dérivation d'état](#).

`UNSAFE_componentWillReceiveProps()` est appelée avant que le composant déjà monté reçoive de nouvelles props. Si vous avez besoin de mettre à jour l'état en réponse à des changements de props (par exemple, pour le réinitialiser), vous pourriez comparer `this.props` et `nextProps`, et déclencher des transitions d'état local en utilisant `this.setState()` au sein de cette méthode.

Remarquez que si un composant parent déclenche le rafraîchissement du vôtre, cette méthode sera appelée même si les props n'ont pas changé. Assurez-vous donc de comparer les valeurs actuelles et futures si vous voulez ne traiter que les modifications.

React n'appelle pas `UNSAFE_componentWillReceiveProps()` avec les props initiales lors du [montage](#). Il ne l'appelle que si au moins une partie des props du composant est susceptible de changer. Appeler `this.setState()` ne déclenche généralement pas `UNSAFE_componentWillReceiveProps()`.

`UNSAFE_componentWillUpdate()` {#unsafe_componentwillupdate}

`UNSAFE_componentWillUpdate(nextProps, nextState)`

Remarque

Cette méthode de cycle de vie s'appelait à l'origine `componentWillUpdate`. Ce nom continuera à fonctionner jusqu'à la sortie de React 17. Utilisez le [codemod rename-unsafe-lifecycles](#) pour mettre à jour automatiquement vos composants.

`UNSAFE_componentWillUpdate()` est appelée juste avant le rendu, quand de nouvelles props ou un nouvel état ont été reçus. C'est l'occasion de faire des travaux préparatoires avant que la mise à jour n'ait lieu. Cette méthode n'est pas appelée avant le rendu initial.

Remarquez que vous ne pouvez pas appeler `this.setState()` ici ; vous ne devez pas non plus faire quoi que ce soit (comme par exemple *dispatcher* une action Redux) qui entraînerait une mise à jour d'un composant React avant que `UNSAFE_componentWillUpdate()` ne se termine.

En général, cette méthode peut être remplacée par `componentDidUpdate()`. Si vous y lisiez des données provenant du DOM (par exemple, la position de défilement), vous pouvez en déplacer le code dans `getSnapshotBeforeUpdate()`.

Remarque

`UNSAFE_componentWillUpdate()` ne sera pas appelée si `shouldComponentUpdate()` renvoie `false`.

Autres API

Contrairement aux méthodes de cycle de vie ci-dessus (que React appelle pour vous), c'est *vous* qui appelez les méthodes ci-dessous depuis vos composants.

Il n'y en a que deux : `setState()` et `forceUpdate()`.

`setState()` {#setstate}

`setState(updater, [callback])`

`setState()` planifie des modifications à l'état local du composant, et indique à React que ce composant et ses enfants ont besoin d'être rafraîchis une fois l'état mis à jour. C'est en général ainsi qu'on met à jour l'interface utilisateur en réaction à des événements ou réponses réseau.

Visualisez `setState()` comme une *demande* plutôt que comme une commande immédiate qui mettrait à jour le composant. Afin d'améliorer la performance perçue, React peut différer son traitement, pour ensuite mettre à jour plusieurs composants en une seule passe. React ne garantit pas que les mises à jour d'état sont appliquées immédiatement.

`setState()` ne met pas toujours immédiatement le composant à jour. Il peut regrouper les mises à jour voire les différer. En conséquence, lire la valeur de `this.state` juste après avoir appelé `setState()` est une mauvaise idée. Utilisez plutôt `componentDidUpdate` ou la fonction de rappel de `setState(setState(updater, callback))`, les deux bénéficiant d'une garantie de déclenchement après que la mise à jour aura été appliquée. Si vous avez besoin de mettre à jour l'état sur base de sa valeur précédente, lisez plus bas comment fonctionne l'argument `updater`.

`setState()` causera toujours un rendu, à moins que `shouldComponentUpdate()` ne renvoie `false`. Si vous y utilisez des objets modifiables et que la logique de rendu conditionnel ne peut pas être implémentée dans `shouldComponentUpdate()`, appeler `setState()` seulement quand le nouvel état diffère du précédent évitera des rafraîchissements superflus.

Le premier argument `updater` est une fonction dont la signature est :

`(state, props) => stateChange`

`state` est une référence à l'état local du composant au moment où cette modification est appliquée. Cet état ne devrait pas être modifié directement. Au lieu de ça, on représente les changements à apporter en construisant un nouvel objet basé sur les données entrantes de `state` et `props`. Par exemple, imaginons que nous voulions incrémenter une valeur dans l'état à raison de `props.step` :

```
this.setState((state, props) => {
  return {counter: state.counter + props.step};
});
```

Tant le `state` que le `props` reçus par la fonction de mise à jour sont garantis à jour au moment de l'appel. La valeur de retour de la fonction est fusionnée (en surface, pas récursivement) avec `state`.

Le second argument de `setState()` est une fonction de rappel optionnelle qui sera exécutée une fois que `setState` est terminé et le composant rafraîchi. D'une façon générale, nous vous recommandons plutôt d'utiliser `componentDidUpdate()` pour ce genre de besoin.

Vous pouvez choisir de passer un objet au lieu d'une fonction comme premier argument à `setState()` :

```
setState(stateChange[, callback])
```

Ça procède à la fusion de surface de `stateChange` dans le nouvel état, par exemple pour ajuster la quantité d'une ligne de commande dans un panier d'achats :

```
this.setState({quantity: 2})
```

Cette forme d'appel à `setState()` reste asynchrone, et des appels répétés au sein du même cycle pourraient être regroupés. Ainsi, si vous tentez d'incrémenter une quantité plus d'une fois dans le même cycle, vous obtiendrez l'équivalent de ceci :

```
Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
)
```

Les appels ultérieurs vont écraser les valeurs des appels précédents du même cycle, de sorte que la quantité ne sera réellement incrémentée qu'une fois. Lorsque l'état suivant dépend de l'état en vigueur, nous vous recommandons de toujours utiliser la forme fonctionnelle du paramètre `updater` :

```
this.setState((state) => {
  return {quantity: state.quantity + 1};
});
```

Pour explorer ce sujet plus en détail, vous pouvez consulter :

- [Le guide État et cycle de vie](#)
- [En profondeur : Quand et pourquoi les appels à `setState\(\)` sont-ils regroupés ?](#) (en anglais)
- [En profondeur : Pourquoi `this.state` ne déclenche-t-il pas une mise à jour immédiate ?](#) (en anglais)

forceUpdate() {#forceupdate}

```
component.forceUpdate(callback)
```

Par défaut, lorsque l'état local ou les props de votre composant changent, ce dernier se rafraîchit. Si votre méthode `render()` dépend d'autres données, vous pouvez indiquer à React que le composant a besoin d'un rafraîchissement en appelant `forceUpdate()`.

Appeler `forceUpdate()` déclenchera le `render()` du composant, en faisant l'impasse sur `shouldComponentUpdate()`. Ça déclenchera les méthodes usuelles de cycle de vie des composants enfants, y compris la méthode `shouldComponentUpdate()` de chaque enfant. React continuera à ne mettre à jour le DOM que si le balisage change.

De façon générale, vous devriez tout faire pour éviter de recourir à `forceUpdate()`, et faire que votre `render()` ne lise que `this.props` et `this.state`.

Propriétés de classes

defaultProps {#defaultprops}

`defaultProps` peut être définie comme propriété sur la classe du composant elle-même, pour définir les valeurs par défaut de props pour cette classe. On s'en sert pour les props `undefined`, mais pas pour celles à `null`. Par exemple :

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

Si `props.color` n'est pas fournie, elle sera définie par défaut à `'blue'` :

```
render() {
  return <CustomButton />; // props.color sera définie à 'blue'
}
```

Si `props.color` est défini à `null`, elle restera à `null` :

```
render() {
  return <CustomButton color={null} />; // props.color reste à `null`
```

displayName {#displayname}

La chaîne de caractères `displayName` est utilisée dans les messages de débogage. La plupart du temps, vous n'avez pas besoin de la définir explicitement parce qu'elle est déduite du nom de la fonction ou classe qui définit le composant. Mais on peut vouloir la définir lorsqu'on veut afficher un nom différent pour des raisons de débogage ou lorsqu'on crée un composant d'ordre supérieur : vous trouverez plus de détails dans [Enrober le nom d'affichage pour faciliter le débogage](#).

Propriétés d'instances

props {#props}

`this.props` contient les props définies par l'appelant de ce composant. Consultez [Composants et props](#) pour une introduction aux props.

Cas particulier : `this.props.children` est spéciale, généralement définie par les balises enfants dans l'expression JSX plutôt que dans la balise du composant lui-même.

state {#state}

L'état local contient des données spécifiques à ce composant, qui sont susceptibles d'évoluer avec le temps. C'est vous qui définissez l'état local, qui devrait être un objet JavaScript brut.

Lorsqu'une valeur n'est utilisée ni par l'affichage ni par le flux de données (par exemple, un ID de minuteur), vous n'avez pas à la mettre dans l'état local. Ce genre de valeurs peuvent être stockées comme champs de l'instance de composant.

Consultez [État et cycle de vie](#) pour de plus amples informations sur l'état local.

Ne modifiez jamais `this.state` directement, car appeler `setState()` par la suite risque d'écraser les modifications que vous auriez apportées. Traitez `this.state` comme s'il était immuable.

ReactDOMServer

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

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

`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.

ReactDOM

Si vous chargez React directement au moyen d'une balise `<script>`, ces API de haut-niveau seront disponibles directement via la variable globale `ReactDOM`. Si vous utilisez ES6 avec `npm`, vous pouvez écrire `import ReactDOM from 'react-dom'`. Si vous utilisez ES5 avec `npm`, utilisez `var ReactDOM = require('react-dom')`.

Aperçu

Le module `react-dom` fournit des méthodes spécifiques au DOM que vous pouvez utiliser au niveau racine de votre appli et comme échappatoire pour travailler hors du modèle React si vous en avez besoin. La plupart des composants ne devraient jamais avoir besoin d'utiliser ce module.

- `render()`
- `hydrate()`
- `unmountComponentAtNode()`
- `findDOMNode()`
- `createPortal()`

Navigateurs pris en charge

React prend en charge tous les navigateurs populaires, y compris Internet Explorer 9 et au-delà, même si certains polyfills sont nécessaires pour les navigateurs plus anciens tels qu'IE 9 et IE 10.

Remarque

Nous ne prenons pas en charge les navigateurs plus anciens qui n'exposent pas les méthodes ES5 mais vous verrez que vos applis peuvent y fonctionner si vous incluez des polyfills tels que `es5-shim` et `es5-sham` dans la page. Ceci dit, vous serez alors livré·e à vous-même en termes de support.

Référence de l'API

`render() {#render}`

ReactDOM.`render(element, container[, callback])`

Affiche un élément React au sein du nœud DOM spécifié par `container` et renvoie une [référence](#) sur le composant (ou renvoie `null` pour les [fonctions composants](#)).

Si l'élément React était déjà affiché dans `container`, cette méthode effectuera plutôt une mise à jour et ne modifiera le DOM que là où c'est strictement nécessaire pour refléter l'élément React à jour.

Si la fonction de rappel optionnelle est fournie, elle sera exécutée après que le composant est affiché ou mis à jour.

Remarque

`ReactDOM.render()` contrôle le contenu du nœud conteneur que vous lui passez. Tout élément DOM existant à l'intérieur sera potentiellement remplacé au premier appel. Les appels ultérieurs utiliseront l'algorithme de différence DOM de React pour des mises à jour efficaces.

`ReactDOM.render()` ne modifie pas le nœud conteneur lui-même (seulement ses enfants). Il peut arriver qu'insérer un composant dans un nœud DOM existant n'en modifie pas les enfants.

`ReactDOM.render()` renvoie pour le moment une référence sur l'instance racine de composant React. Toutefois, manipuler la valeur renvoyée est considéré comme déprécié, et vous devriez vous en abstenir dans la mesure où de futures versions de React pourraient gérer le rendu de façon asynchrone dans certains cas. Si vous avez besoin d'une référence sur l'instance racine de composant React, une meilleure solution consiste à associer une [ref de rappel](#) à l'élément racine.

Utiliser `ReactDOM.render()` pour hydrater un conteneur rendu côté serveur est une pratique dépréciée qui disparaîtra avec React 17. Utilisez plutôt la méthode `hydrate()`.

hydrate() {#hydrate}

`ReactDOM.hydrate(element, container[, callback])`

Similaire à `render()`, mais sert à hydrater un conteneur dont le HTML a déjà été produit par `ReactDOMServer`. React tentera alors d'associer les gestionnaires d'événements au balisage existant.

React s'attend à ce que le balisage produit soit identique entre le serveur et le client. Il peut gérer des différences de contenu textuel mais vous devriez considérer toute erreur de correspondance comme un bug et la corriger. En mode développement, React vous avertit de telles erreurs lors de l'hydratation. Nous ne garantissons notamment pas que les écarts d'attributs seront correctement résolus. C'est important pour les performances car dans la plupart des applications les erreurs de correspondance sont rares et faire une validation trop fine du balisage serait sans doute inutilement coûteux.

Si un attribut ou contenu textuel sur un élément précis est forcément différent entre le serveur et le client (par exemple, un horodatage), vous pouvez empêcher cet avertissement en ajoutant `suppressHydrationWarning={true}` à l'élément. Ça ne marche toutefois qu'à un niveau de profondeur et c'est considéré comme une échappatoire. N'en abusez pas. À moins qu'il ne s'agisse de contenu textuel, React ne tentera de toutes façons pas de résoudre la différence qui pourrait rester incohérente jusqu'aux prochaines mises à jour.

Si vous faites volontairement un rendu différent entre le serveur et le client, vous pouvez procéder à un rendu en deux passes. Les composants qui ont un rendu serveur différent peuvent lire une variable d'état local du style `this.state.isClient`, que vous définirez à `true` au sein de `componentDidMount()`. Ainsi,

la première passe de rendu affichera le même contenu que celui du serveur, évitant les écarts, mais une passe supplémentaire surviendra de façon synchrone juste après l'hydratation. Remarquez que cette approche ralentira l'initialisation de vos composants car ils devront faire un double rendu au démarrage : ne l'employez qu'avec précaution.

Souvenez-vous que les connexions lentes ont un impact sur l'expérience utilisateur. Le code JavaScript peut être chargé très longtemps après l'affichage HTML initial de sorte que si vous affichez quelque chose de différent lors du rendu côté client, la transition risque de désorienter l'utilisateur. Ceci dit, si vous vous y prenez bien, il peut tout de même être utile de produire d'abord une « ossature » de l'application sur le serveur et de n'ajouter certains éléments visuels supplémentaires qu'une fois côté client. Pour apprendre à faire ça sans souffrir d'écarts de balisage, relisez le paragraphe précédent.

`unmountComponentAtNode()` **{#unmountcomponentatnode}**

`ReactDOM.unmountComponentAtNode(container)`

Retire un composant React monté du DOM et nettoie ses gestionnaires d'événements et son état local. Si aucun composant n'était monté sur ce conteneur, appeler cette fonction ne fait rien. Renvoie `true` si un composant a bien été démonté et `false` si aucun composant ne nécessitait de démontage.

`findDOMNode()` **{#finddomnode}**

Remarque

`findDOMNode` est une échappatoire utilisée pour accéder au nœud du DOM associé au composant. La plupart du temps, nous déconseillons le recours à cette échappatoire parce qu'elle rompt l'abstraction du composant. [Elle est d'ailleurs désormais dépréciée en mode strict.](#)

`ReactDOM.findDOMNode(component)`

Si le composant a été monté dans le DOM, cette méthode renvoie l'élément DOM (natif au navigateur) qui a servi de point de montage. Ça peut être utile pour lire des valeurs issues du DOM, telles que la valeur d'un champ de formulaire, ou pour effectuer des mesures de tailles sur les nœuds du DOM. **La plupart du temps, préférez associer une ref au nœud du DOM, et évitez complètement `findDOMNode`.**

Quand le rendu d'un composant produit `null` ou `false`, `findDOMNode` renvoie `null`. Quand ça produit une chaîne de caractère, `findDOMNode` renvoie le nœud DOM textuel qui contient cette valeur. Depuis React 16, un composant peut renvoyer un fragment avec de multiples enfants, auquel cas `findDOMNode` renverra le nœud DOM correspondant au premier enfant non-vide.

Remarque

`findDOMNode` ne fonctionne que sur des composants montés (c'est-à-dire des composants qui ont été placés dans le DOM). Si vous tentez de l'appeler sur un composant qui n'a pas encore été monté (par exemple en appelant `findDOMNode()` dans le `render()` initial d'un composant), une exception sera levée.

`findDOMNode` ne peut pas être utilisée sur des fonctions composants.

`createPortal()` **#createportal}**

ReactDOM.`createPortal`(child, container)

Crée un portail. Les portails fournissent un moyen d'afficher des enfants dans un nœud du DOM à l'extérieur de la hiérarchie DOM du composant.

L'API haut-niveau de React

L'objet `React` est le point d'entrée de la bibliothèque React. Si vous chargez React depuis une balise `<script>`, ces API de haut-niveau sont disponibles depuis l'objet global `React`. Si vous utilisez npm avec la syntaxe ES6, vous pouvez écrire : `import React from 'react'`. Si vous utilisez npm avec la syntaxe ES5, vous pouvez écrire : `var React = require('react')`.

Aperçu

Composants

Les composants React vous permettent de découper une interface utilisateur (*UI, pour User Interface, NdT*) en blocs indépendants et réutilisables, ce qui vous permet de concevoir chaque partie en isolation. Un composant React peut être défini en étendant les classes `React.Component` ou `React.PureComponent`.

- `React.Component`
- `React.PureComponent`

Si vous n'utilisez pas les classes ES6, vous pouvez utiliser le module `create-react-class` à la place. Lisez [React sans ES6](#) pour plus de détails.

Un composant React peut aussi être défini via une fonction que l'on pourra choisir d'enrober avec `React.-memo` :

- `React.memo`

Créer des éléments React

Nous vous recommandons d'[utiliser JSX](#) pour définir à quoi ressemblera votre UI. Tous les éléments JSX ne sont que du sucre syntaxique qui cache des appels à `React.createElement()`. Si vous utilisez JSX, vous ne devriez pas avoir besoin d'appeler les méthodes suivantes :

- `createElement()`
- `createFactory()`

Lisez [React sans JSX](#) pour plus de détails.

Transformer des éléments

`React` propose plusieurs API pour manipuler les éléments :

- `cloneElement()`
- `isValidElement()`
- `React.Children`

Fragments

`React` fournit également un composant particulier pour réaliser le rendu de plusieurs éléments sans avoir à les envelopper avec un autre.

- `React.Fragment`

Refs

- `React.createRef`
- `React.forwardRef`

Suspense

`Suspense` permet aux composants « d'attendre » quelque chose avant de s'afficher. Pour le moment, `Suspense` ne prend en charge qu'un seul cas d'utilisation : [le chargement dynamique de composants avec `React.lazy`](#). À l'avenir, il permettra de gérer d'autres cas d'utilisations tels que le chargement de données distantes.

- `React.lazy`
- `React.Suspense`

Hooks

Les *Hooks* sont une nouveauté de React 16.8. Ils vous permettent d'utiliser les états et d'autres fonctionnalités de React sans avoir à écrire de classes. Les *Hooks* disposent de [leur propre documentation](#) et leur API est détaillée à part :

- [Hooks de base](#)
 - `useState`
 - `useEffect`
 - `useContext`
- [Hooks supplémentaires](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`

Référence de l'API

`React.Component` [{#reactcomponent}](#)

`React.Component` est la classe de base utilisée pour créer des composants React avec la syntaxe des [classes ES6](#) :

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

Rendez-vous sur [la page de référence de l'API `React.Component`](#) pour voir la liste complète des méthodes et propriétés de la classe de base `React.Component`.

`React.PureComponent` [\(#reactpurecomponent\)](#)

`React.PureComponent` est similaire à `React.Component`. La seule différence est que `React.Component` n'implémente pas la méthode `shouldComponentUpdate()`, alors que `React.PureComponent` l'implémente en réalisant une comparaison de surface de l'état et des propriétés.

Si la fonction `render()` d'un de vos composants React produit un rendu identique pour le même état et les mêmes propriétés, le faire étendre `React.PureComponent` devrait améliorer les performances dans certains cas.

Remarque

La méthode `shouldComponentUpdate()` de `React.PureComponent` réalise une simple comparaison de surface. Avec des données complexes, elle peut produire des faux négatifs si la structure de données subit des changements profonds. Ne créez des composants avec `PureComponent` que si vous avez des états et des props simples, et le cas échéant utilisez `forceUpdate()` si vous savez que vos données ont changé en profondeur. Vous pouvez aussi envisager d'utiliser des [objets immuables](#) pour simplifier la comparaison rapide de données imbriquées.

De plus, la méthode `shouldComponentUpdate()` de `React.PureComponent` ignore la mise à jour des propriétés de tout l'arbre des composants enfants. Assurez-vous donc que tous les composants enfants sont également « purs ».

`React.memo` [\(#reactmemo\)](#)

```
const MyComponent = React.memo(function MyComponent(props) {
  /* Faire le rendu en utilisant les props */
});
```

`React.memo` est un [composant d'ordre supérieur](#).

Si vous avez un composant qui affiche toujours le même résultat pour un même jeu de propriétés, vous pouvez l'enrober avec `React.memo`, ce qui mémoisera le résultat et devrait augmenter les performances dans certains cas. Cela signifie que React sautera le rafraîchissement du composant en réutilisant son dernier rendu en date.

`React.memo` ne se préoccupe que des modifications de props. Si votre fonction composant enrobée par `React.memo` utilise un Hook `useState` ou `useContext` dans son implémentation, des changements d'état local ou de contexte entraîneront tout de même un nouveau rendu.

Par défaut, seule une comparaison de surface des props sera faite. Si vous voulez gérer cette comparaison vous-même, vous pouvez fournir une fonction de comparaison personnalisée en deuxième argument.

```
function MyComponent(props) {
  /* Faire le rendu en utilisant les props */
}
function areEqual(prevProps, nextProps) {
  /*
    Renvoie `true` si passer l'objet nextProps à la fonction de rendu
    produira le même résultat que de lui passer l'objet prevProps.
    Renvoie `false` dans le cas contraire.
  */
}
export default React.memo(MyComponent, areEqual);
```

Cette méthode n'est qu'un outil d'**optimisation des performances**. Ne vous y fiez pas pour « empêcher » un rendu car cela peut causer des bugs.

Remarque

Contrairement à la méthode `shouldComponentUpdate()` des composants créés à l'aide de classes, la fonction `areEqual` renvoie `true` si les props sont égales et `false` dans le cas contraire. C'est donc l'inverse de `shouldComponentUpdate`.

createElement() [\[#createelement\]](#)

```
React.createElement(
  type,
  [props],
  [...children]
)
```

Cette méthode crée et renvoie un nouvel **élément React** du type indiqué. L'argument `type` peut être au choix : une chaîne contenant un nom de balise (tel que `'div'` ou `'span'`), un type de **composant React** (une classe ou une fonction), ou encore un **fragment React**.

Tout code écrit avec **JSX** sera converti de manière à utiliser `React.createElement()`. Normalement vous ne devriez pas appeler `React.createElement()` si vous utilisez JSX. Lisez **React sans JSX** pour en savoir plus.

cloneElement() {#cloneelement}

```
React.cloneElement(  
  element,  
  [props],  
  [...children]  
)
```

Cette méthode clone et renvoie un nouvel élément en utilisant `element` comme point de départ. L'élément obtenu aura les props de l'élément originel augmentées par une fusion de surface des nouvelles props. Les nouveaux éléments enfants (`children`) remplaceront les anciens. Les `key` et `ref` issues de l'élément original seront préservées.

`React.cloneElement()` est quasiment équivalent à :

```
<element.type {...element.props} {...props}>{children}</element.type>
```

Cependant elle préserve les `ref`. Concrètement, ça signifie que si vous avez un enfant avec une `ref` associée, vous ne la volerez pas accidentellement à votre ancêtre. Vous aurez la même `ref` associée au nouvel élément.

Cette API a été introduite pour remplacer la méthode dépréciée `React.addons.cloneWithProps()`.

createFactory() {#createfactory}

```
React.createFactory(type)
```

Cette méthode renvoie une fonction qui produit des éléments React d'un type donné. Comme pour `React.createElement()`, l'argument `type` peut être au choix : une chaîne contenant un nom de balise (tel que `'div'` ou `'span'`), un type de [composant React](#) (une classe ou une fonction), ou encore un type de [fragment React](#).

Cette fonction d'aide est historique et nous vous encourageons plutôt à utiliser JSX ou directement `React.createElement()`.

Normalement vous ne devriez pas appeler `React.createFactory()` si vous utilisez JSX. Lisez [React sans JSX](#) pour en savoir plus.

isValidElement() {#isvalidelement}

```
React.isValidElement(object)
```

Cette méthode vérifie qu'un objet est bien un élément React. Elle renvoie `true` ou `false`.

React.Children {#reactchildren}

`React.Children` fournit des utilitaires pour interagir avec la structure de données opaque de `this.props.children`.

React.Children.map {#reactchildrenmap}

```
React.Children.map(children, function[(thisArg)])
```

Cette méthode exécute une fonction sur chacun des enfants directs contenus dans `children` avec le `this` en vigueur transmis dans l'argument `thisArg`. Si `children` est un tableau, il sera parcouru et la fonction sera appelée sur chacun des enfants du tableau. Si `children` est `null` ou `undefined`, la méthode renverra `null` ou `undefined` plutôt qu'un tableau.

Remarque

Si `children` est un `Fragment` il sera traité comme un unique enfant et ne sera pas parcouru.

React.Children.forEach {#reactchildrenforeach}

```
React.Children.forEach(children, function[(thisArg)])
```

Même chose que `React.Children.map()` mais sans renvoyer de tableau.

React.Children.count {#reactchildrencount}

```
React.Children.count(children)
```

Cette méthode renvoie le nombre total de composants présents dans `children`, ce total étant égal au nombre de fois qu'une fonction de rappel passée à `map` ou `forEach` serait exécutée.

React.Children.only {#reactchildrenonly}

```
React.Children.only(children)
```

Cette méthode vérifie que `children` n'a qu'un seul enfant (un élément React) et le renvoie. Si ce n'est pas le cas elle lèvera une erreur.

Remarque

`React.Children.only()` n'accepte pas la valeur de retour de `React.Children.map()` car il s'agit d'un tableau et non d'un élément React.

React.Children.toArray {#reactchildrentoarray}

```
React.Children.toArray(children)
```

Cette méthode renvoie la structure de donnée opaque de `children` sous la forme d'un tableau linéarisé où chaque enfant est doté d'une `key`. C'est utile si vous voulez manipuler une collection d'enfants dans votre méthode de rendu, en particulier si vous voulez réorganiser ou découper `this.props.children` avant de le passer à d'autres éléments.

Remarque

`React.Children.toArray()` change les `key` pour préserver la sémantique des tableaux imbriqués pendant la linéarisation des enfants. Pour cela, `toArray` préfixe chaque `key` dans le tableau qui sera renvoyé de manière à ce que la `key` de chaque élément soit associée au tableau originel qui les contient.

React.Fragment {#reactfragment}

Le composant `React.Fragment` vous permet de renvoyer plusieurs éléments depuis une méthode `render()` sans avoir à créer un élément DOM supplémentaire :

```
render() {
  return (
    <React.Fragment>
      Du texte.
      <h2>Un en-tête</h2>
    </React.Fragment>
  );
}
```

Vous pouvez également l'utiliser via la syntaxe raccourcie `<></>`. Pour plus d'information, lisez [React v16.2.0: Improved Support for Fragments](#) (en anglais).

React.createRef {#reactcreateref}

`React.createRef` crée une `ref` qui peut être associée à des éléments React via l'attribut `ref`.

`embed:16-3-release-blog-post/create-ref-example.js`

React.forwardRef {#reactforwardref}

`React.forwardRef` crée un composant React qui transfère la valeur de l'attribut `ref` qu'il reçoit à un autre composant plus bas dans l'arbre. Cette technique est assez inhabituelle mais elle est particulièrement utile dans deux cas :

- [Transférer une référence à un composant DOM](#)
- [Transférer une référence au sein d'un composant d'ordre supérieur](#)

`React.forwardRef` prend une fonction de rendu comme argument. React appellera cette fonction avec deux arguments `props` et `ref`. Cette fonction devrait renvoyer un nœud React.

`embed:reference-react-forward-ref.js`

Dans l'exemple ci-avant, la fonction de rendu passée à `React.forwardRef` recevra comme deuxième argument la `ref` initialement fournie à l'élément `<FancyButton ref={ref}>`. Cette fonction va alors transférer la `ref` à l'élément `<button ref={ref}>`.

En conséquence, après que React aura attaché la `ref`, `ref.current` pointera directement vers l'instance de l'élément DOM de `<button>`.

Pour en savoir plus, lisez [Transférer les refs](#)

React.lazy {#reactlazy}

`React.lazy()` vous permet de définir un composant qui sera chargé dynamiquement. Cela aide à réduire la taille du fichier initial en reportant à plus tard le chargement des composants inutiles lors du rendu initial.

Vous pouvez apprendre comment l'utiliser en lisant [la documentation sur la découpe du code](#). Vous voudrez peut-être aussi jeter un œil à [cet article](#), qui explique comment l'utiliser en détail.

```
// Ce composant est chargé dynamiquement
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

Notez bien que l'affichage d'un composant `lazy` a besoin d'un composant `<React.Suspense>` plus haut dans l'arbre de rendu. C'est de cette manière que vous pouvez spécifier un indicateur de chargement.

Remarque

Utiliser `React.lazy` avec un `import()` dynamique requiert une prise en charge des *Promises* par l'environnement JS. Vous aurez donc besoin d'un *polyfill* pour IE11 et inférieurs.

React.Suspense {#reactsuspense}

`React.Suspense` vous permet de définir un indicateur de chargement pour le cas où certains composants plus bas dans l'arbre de rendu ne seraient pas encore prêts à être affichés. Pour le moment le **seul** cas d'usage pris en charge par `<React.Suspense>`, c'est le chargement différé de composants via `React.lazy` :

```
// Ce composant est chargé dynamiquement
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // Affiche <Spinner> jusqu'à ce que <OtherComponent> soit chargé
    <React.Suspense fallback={<Spinner />}>
      <div>
        <OtherComponent />
      </div>
    </React.Suspense>
  );
}
```

Tout ça est détaillé dans [notre guide sur la découpe du code](#). Remarquez que les composants `lazy` peuvent être profondément enfouis dans l'arbre des descendants de `Suspense` —ils n'ont pas besoin d'être enveloppés individuellement. La bonne pratique consiste à placer un `<Suspense>` aux endroits où vous souhaitez voir un indicateur de chargement, et à utiliser `lazy()` partout où vous voulez découper votre code.

Bien que ce ne soit pas le cas pour le moment, nous prévoyons d'étendre les capacités de `Suspense` pour qu'il puisse gérer d'autre scénarios tel que le chargement de données. Vous pourrez en savoir plus en jetant un coup d'œil à [notre feuille de route](#).

Remarque

`React.lazy()` et `<React.Suspense>` ne sont pas encore pris en charge par `ReactDOMServer`. C'est une limitation connue qui devrait être résolue à l'avenir.

Moteur de rendu de test

Importation

```
import TestRenderer from 'react-test-renderer'; // ES6
const TestRenderer = require('react-test-renderer'); // ES5 avec npm
```

Aperçu

Ce paquet fournit un moteur de rendu (*dans la suite de cet article, pour des raisons de concision, nous employons le terme générique anglais renderer sans italiques, NdT*) capable de produire un rendu de composants React sous forme d'objets JavaScript nus, sans dépendre du DOM ou d'un environnement mobile natif.

Fondamentalement, ce paquet facilite la création d'un instantané de la hiérarchie produite par un renderer React DOM ou React Native (similaire à un arbre DOM) sans recourir à un navigateur ou à [jsdom](#).

Exemple :

```
import TestRenderer from 'react-test-renderer';

function Link(props) {
  return <a href={props.page}>{props.children}</a>;
}

const testRenderer = TestRenderer.create(
  <Link page="https://www.facebook.com/">Facebook</Link>
);

console.log(testRenderer.toJSON());
// { type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ] }
```

Vous pouvez utiliser la fonctionnalité de test par instantanés (*snapshot testing, NdT*) de Jest pour sauvegarder automatiquement une copie de l'arbre JSON obtenu dans un fichier, puis vérifier dans vos tests qu'il n'a pas changé : [vous trouverez plus de détails ici](#).

Vous pouvez également explorer le résultat pour trouver des noeuds spécifiques et vérifier vos attentes les concernant.

```
import TestRenderer from 'react-test-renderer';

function MyComponent() {
  return (
    <div>
      <SubComponent foo="bar" />
      <p className="my">Salut</p>
    </div>
  )
}

function SubComponent() {
  return (
```

```

    <p className="sub">Sous-Composant</p>
  );
}

const testRenderer = TestRenderer.create(<MyComponent />);
const testInstance = testRenderer.root;

expect(testInstance.findByType(SubComponent).props.foo).toBe('bar');
expect(testInstance.findByProps({className: "sub"}).children).toEqual(['Sous-Composant']);

```

TestRenderer

- `TestRenderer.create()`
- `TestRenderer.act()`

Instance de `TestRenderer` `{#testrenderer-instance}`

- `testRenderer.toJSON()`
- `testRenderer.toTree()`
- `testRenderer.update()`
- `testRenderer.unmount()`
- `testRenderer.getInstance()`
- `testRenderer.root`

TestInstance

- `testInstance.find()`
- `testInstance.findByType()`
- `testInstance.findByProps()`
- `testInstance.findAll()`
- `testInstance.findAllByType()`
- `testInstance.findAllByProps()`
- `testInstance.instance`
- `testInstance.type`
- `testInstance.props`
- `testInstance.parent`
- `testInstance.children`

Référence de l'API

`TestRenderer.create()` `{#testrenderercreate}`

```
TestRenderer.create(element, options);
```

Crée une instance de `TestRenderer` avec l'élément React passé en argument. Ça n'utilise pas un véritable DOM, mais ça ne l'empêche pas de produire l'arbre intégral des composants en mémoire pour pouvoir vérifier vos attentes dessus. Renvoie une [instance de `TestRenderer`](#).

TestRenderer.act() {#testrendereract}

`TestRenderer.act(callback);`

De la même manière que l'utilitaire `act()` de `react-dom/test-utils`, `TestRenderer.act` prépare un composant permettant la vérification d'attentes. Utilisez cette version de `act()` pour englober les appels à `TestRenderer.create` et `testRenderer.update`.

```
import {create, act} from 'react-test-renderer';
import App from './app.js'; // Le composant testé

// Fait le rendu du component
let root;
act(() => {
  root = create(<App value={1}>/);
});

// Exprime des attentes sur la racine
expect(root.toJSON()).toMatchSnapshot();

// Met à jour avec des props différentes
act(() => {
  root.update(<App value={2}>);
});

// Exprime des attentes sur la racine
expect(root.toJSON()).toMatchSnapshot();
```

testRenderer.toJSON() {#testrenderertojson}

`testRenderer.toJSON()`

Renvoie un objet représentant l'arbre obtenu. Cet arbre contient uniquement les nœuds spécifiques à la plateforme comme `<div>` ou `<View>` avec leur props mais ne contient aucun composant écrit par l'utilisateur. C'est pratique pour le [test par instantanés](#).

testRenderer.toTree() {#testrenderertotree}

`testRenderer.toTree()`

Renvoie un objet représentant l'arbre obtenu. Contrairement à `toJSON()`, la représentation y est plus détaillée et contient les composants écrits par l'utilisateur. Vous n'aurez probablement pas besoin de cette méthode à moins que vous n'écriviez votre propre bibliothèque de vérification d'attentes construite au-dessus du renderer de test.

testRenderer.update() {#testrendererupdate}

`testRenderer.update(element)`

Effectue à nouveau le rendu de l'arbre en mémoire, avec un nouvel élément racine. Ça simule une mise à jour de l'arbre React à la racine. Si le nouvel élément a le même type et la même clé que l'élément précédent, l'arbre sera mis à jour ; dans le cas contraire, un nouvel arbre sera re-monté.

testRenderer.unmount() {#testrendererunmount}`testRenderer.unmount()`

Démonte l'arbre en mémoire, en déclenchant les événements de cycle de vie appropriés.

testRenderer.getInstance() {#testrenderergetinstance}`testRenderer.getInstance()`

Renvoie l'instance correspondant à l'élément racine, s'il est disponible. Ça ne marchera pas si l'élément racine est une fonction composant car celles-ci n'ont pas d'instance.

testRenderer.root {#testrendererroot}`testRenderer.root`

Renvoie l'objet « instance de test » racine, qui est utile pour faire des vérifications d'attentes à propos de nœuds spécifiques dans l'arbre. Vous pouvez l'utiliser pour trouver d'autres « instances de test » présentes plus profond dans l'arbre.

testInstance.find() {#testinstancefind}`testInstance.find(test)`

Trouve une unique « instance de test » descendante pour laquelle `test(testInstance)` renvoie `true`. Si `test(testInstance)` ne renvoie pas `true` pour exactement une « instance de test », une erreur est levée.

testInstance.findByType() {#testinstancefindbytype}`testInstance.findByType(type)`

Trouve une unique « instance de test » avec le `type` donné. Si il n'y a pas exactement une « instance de test » avec le `type` donné, une erreur est levée.

testInstance.findByProps() {#testinstancefindbyprops}`testInstance.findByProps(props)`

Trouve une unique « instance de test » avec les `props` données. Si il n'y a pas exactement une « instance de test » avec les `props` données, une erreur est levée.

testInstance.findAll() {#testinstancefindall}`testInstance.findAll(test)`

Trouve toutes les « instances de test » descendantes pour lesquelles `test(testInstance)` renvoie `true`.

testInstance.findAllByType() {#testinstancefindallbytype}

```
testInstance.findAllByType(type)
```

Trouve toutes les « instances de test » descendantes avec le `type` donné.

testInstance.findAllByProps() {#testinstancefindallbyprops}

```
testInstance.findAllByProps(props)
```

Trouve toutes les « instances de test » descendantes avec les `props` données.

testInstance.instance {#testinstanceinstance}

```
testInstance.instance
```

L'instance de composant correspondant à cette « instance de test ». Ça n'est disponible que pour les composants à base de classe, vu que les fonctions composants n'ont pas d'instances. Ça correspond à la valeur de `this` au sein du composant donné.

testInstance.type {#testinstancetype}

```
testInstance.type
```

Le type de composant correspondant à cette « instance de test ». Par exemple, un composant `<Button />` a pour type `Button`.

testInstance.props {#testinstanceprops}

```
testInstance.props
```

Les props correspondant à cette « instance de test ». Par exemple, un composant `<Button size="small" />` a comme props `{size: 'small'}`.

testInstance.parent {#testinstanceparent}

```
testInstance.parent
```

L'« instance de test » parente de cette « instance de test ».

testInstance.children {#testinstancechildren}

```
testInstance.children
```

Les « instances de test » enfants de cette « instance de test ».

Idées

Vous pouvez passer la fonction `createNodeMock` à `TestRenderer.create` comme option pour créer des simulations personnalisées de refs. `createNodeMock` accepte l'élément courant et doit retourner une imitation de ref. C'est utile quand vous testez un composant qui utilise des refs.

```
import TestRenderer from 'react-test-renderer';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.input = null;
  }
  componentDidMount() {
    this.input.focus();
  }
  render() {
    return <input type="text" ref={el => this.input = el} />
  }
}

let focused = false;
TestRenderer.create(
  <MyComponent />,
  {
    createNodeMock: (element) => {
      if (element.type === 'input') {
        // simule une fonction focus
        return {
          focus: () => {
            focused = true;
          }
        };
      }
      return null;
    }
);
expect(focused).toBe(true);
```

Les refs et le DOM

Les refs fournissent un moyen d'accéder aux nœuds du DOM ou éléments React créés dans la méthode de rendu.

Dans le flux de données habituel de React, les `props` constituent le seul moyen pour des composants parents d'interagir avec leurs enfants. Pour modifier un enfant, vous le rafraîchissez avec de nouvelles props. Ceci dit, dans certains cas vous aurez besoin de modifier un enfant de façon impérative, hors du flux de données normal. L'enfant à modifier pourrait être une instance d'un composant React ou un élément DOM. Dans les deux cas, React vous fournit une échappatoire.

Quand utiliser les refs

Voici quelques cas d'usages tout trouvés pour les refs :

- Gérer le focus, la sélection du texte, ou la lecture de média.
- Lancer des animations impératives.
- S'interfacer avec des bibliothèques DOM tierces.

Evitez d'utiliser les refs pour tout ce qui peut être fait déclarativement.

Par exemple, au lieu d'exposer les méthodes `open()` et `close()` dans un composant `Dialog`, vous pouvez lui passer une prop `isOpen`.

N'abusez pas des refs

Vous serez peut-être enclin·e à toujours commencer par une ref pour « faire le boulot » dans votre appli. Si tel est le cas, examinez d'un œil critique votre hiérarchie de composants pour déterminer lesquels sont censés posséder l'état. Vous remarquerez souvent que l'état serait mieux géré plus haut dans la hiérarchie. Voyez le guide [Faire remonter l'état](#) pour des exemples.

Remarque

Les exemples ci-dessous ont été mis à jour pour utiliser l'API `React.createRef()` introduite dans React 16.3. Si vous utilisez une version précédente de React, nous recommandons d'utiliser [les refs avec fonctions de rappel](#) à la place.

Créer des refs

Les refs sont créées en utilisant `React.createRef()` et attachées aux éléments React via l'attribut `ref`. Les refs sont souvent affectées à une propriété d'instance quand un composant est construit et peuvent donc être référencées à travers le composant.

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

Accéder aux refs

Quand une ref est passée à un élément dans `render`, une référence au nœud devient accessible via l'attribut `current` de la ref.

```
const node = this.myRef.current;
```

La valeur de la ref change suivant le type de nœud :

- Quand l'attribut `ref` est utilisé sur un élément HTML, la `ref` créée dans le constructeur avec `React.createRef()` reçoit l'élément DOM sous-jacent dans sa propriété `current`.
- Quand l'attribut `ref` est utilisé sur un composant de classe personnalisée, l'objet `ref` reçoit l'instance du composant créée dans son `current`.
- **Vous ne pouvez pas utiliser l'attribut `ref` sur les fonctions composants** parce qu'elles n'ont pas d'instance.

Les exemples ci-dessous illustrent les différences.

Ajouter une ref à un élément du DOM

Ce code utilise une `ref` pour stocker une référence à un nœud du DOM :

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // Crée une référence pour stocker l'élément DOM textField
    this.textField = React.createRef();
    this.focusTextField = this.focusTextField.bind(this);
  }

  focusTextField() {
    // Donne explicitement le focus au champ texte en utilisant l'API DOM native.
    // Remarque : nous utilisons `current` pour cibler le nœud DOM
    this.textField.current.focus();
  }

  render() {
    // Dit à React qu'on veut associer la ref `textField` créée
    // dans le constructeur avec le `<input>`.
    return (
      <div>
        <input
          type="text"
          ref={this.textField} />
        <input
          type="button"
          value="Donner le focus au champ texte"
    
```

```

        onClick={this.focusTextInput}
      />
    </div>
  );
}
}

```

React affectera l'élément DOM à la propriété `current` quand le composant sera monté, et la remettra à `null` lorsqu'il sera démonté. La `ref` est mise à jour avant le déclenchement des méthodes de cycle de vie `componentDidMount` et `componentDidUpdate`.

Ajouter une ref à un composant à base de classe

Si on voulait enrober le `CustomTextInput` ci-dessus pour simuler un clic immédiatement après le montage, on pourrait utiliser une ref pour accéder au champ personnalisé et appeler manuellement sa méthode `focusTextInput` :

```

class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focusTextInput();
  }

  render() {
    return (
      <CustomTextInput ref={this.textInput} />
    );
  }
}

```

Il faut noter que ça ne fonctionne que si `CustomTextInput` est déclaré comme une classe :

```

class CustomTextInput extends React.Component {
  // ...
}

```

Les refs et les fonctions composants

Par défaut, **vous ne pouvez pas utiliser l'attribut `ref` sur les fonctions composants** parce qu'elles n'ont pas d'instance.

```

function MyFunctionComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  render() {
    // Ça ne fonctionnera pas !
    return (

```

```

        <MyFunctionComponent ref={this.TextInput} />
    );
}
}

```

Si vous voulez permettre à vos utilisateurs de passer une `ref` à votre fonction composant, vous pouvez utiliser `forwardRef` (peut-être combiné à un `useImperativeHandle`), ou vous pouvez convertir votre composant pour être à base de classe.

Vous pouvez néanmoins **utiliser l'attribut `ref` dans une fonction composant**, tant que vous vous référez à un élément DOM ou un composant à base de classe :

```

function CustomTextInput(props) {
  // TextInput doit être déclaré ici pour que la ref puisse s'y référer
  const TextInput = useRef(null);

  function handleClick() {
    TextInput.current.focus();
  }

  return (
    <div>
      <input
        type="text"
        ref={TextInput} />
      <input
        type="button"
        value="Donner le focus au champ texte"
        onClick={handleClick}
      />
    </div>
  );
}

```

Exposer les refs DOM aux composants parents

Dans de rares cas, un composant parent pourrait vouloir accéder aux nœuds DOM d'un enfant. C'est généralement déconseillé car ça brise l'encapsulation, mais c'est parfois utile pour gérer le focus, mesurer les dimensions ou la position d'un nœud DOM enfant.

Même si vous pourriez [ajouter une ref à un composant enfant](#), ce n'est pas une solution idéale car vous n'obtiendriez qu'une instance de composant plutôt qu'un nœud DOM. De plus, ça ne fonctionnerait pas avec les fonctions composants.

Si vous utilisez React 16.3 ou une version ultérieure, nous recommandons d'utiliser [le transfert de refs](#) pour ce genre de cas. **Le transfert de refs permet à un composant de choisir d'exposer une ref à un de ses enfants comme étant la sienne.** Vous trouverez un exemple détaillé de la façon d'exposer un nœud DOM enfant à un composant parent [dans la documentation du transfert de refs](#).

Si vous utilisez React 16.2 ou une version antérieure, ou si vous avez besoin de plus de flexibilité que ce que permet le transfert de refs, vous pouvez utiliser [cette approche alternative](#) et passer explicitement une ref via une autre prop.

Autant que possible, nous déconseillons d'exposer les nœuds DOM, mais ça peut être une échappatoire utile. Remarquez que cette approche exige la modification du code du composant enfant. Si vous n'avez pas cette possibilité, votre seule option consiste à utiliser `findDOMNode()`, mais c'est déconseillé et déprécié en mode strict.

Refs avec fonctions de rappel

React propose une autre façon de définir des refs appelée « refs avec fonctions de rappel », qui permet un contrôle plus fin sur l'affectation et le nettoyage des refs.

Plutôt que de passer un attribut `ref` créé par `createRef()`, vous pouvez passer une fonction. La fonction récupère l'instance du composant React ou l'élément du DOM HTML comme argument, qui peut être stocké et accédé depuis ailleurs.

L'exemple ci-dessous implémente une approche fréquente : utiliser la fonction de rappel `ref` pour stocker une référence à un nœud DOM dans une propriété d'instance.

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);

    this.textInput = null;

    this.setTextInputRef = element => {
      this.textInput = element;
    };

    this.focusTextInput = () => {
      // Donne le focus au champ texte en utilisant l'API DOM native.
      if (this.textInput) this.textInput.focus();
    };
  }

  componentDidMount() {
    // Focus automatique sur le champ au montage
    this.focusTextInput();
  }

  render() {
    // Utilise la fonction de rappel `ref` pour stocker une référence à l'élément
    // DOM du champ texte dans une propriété d'instance (ex. this.textInput)
    return (
      <div>
        <input
          type="text"
          ref={this.setTextInputRef}
        />
        <input
          type="button"
          value="Donner le focus au champ texte"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

React appellera la fonction de rappel `ref` avec l'élément DOM quand le composant sera monté, puis avec `null` quand il sera démonté. Les refs sont toujours mises à jour avant l'exécution de `componentDidMount` et `componentDidUpdate`.

Vous pouvez passer une fonction de rappel de ref d'un composant à l'autre comme vous le feriez avec les objets refs créés par `React.createRef()`.

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  render() {
    return (
      <CustomTextInput
        inputRef={el => this.inputElement = el}
      />
    );
  }
}
```

Dans l'exemple ci-dessus, `Parent` passe sa fonction de rappel de ref dans la propriété `inputRef` du `CustomTextInput`, et `CustomTextInput` passe la même fonction dans l'attribut spécial `ref` à l'`<input>`. Au final, `this.inputElement` dans `Parent` recevra le nœud DOM correspondant à l'élément `<input>` dans `CustomTextInput`.

API historique : refs textuelles

Si vous avez travaillé avec React par le passé, vous avez peut-être l'habitude d'une ancienne API où l'attribut `ref` était une chaîne de caractères du genre `"textInput"`, et le nœud DOM était accessible via `this.refs.textInput`. Nous recommandons de ne plus utiliser cette approche en raison de [plusieurs problèmes](#) ; elle est dépréciée et **sera probablement supprimée dans une version future**.

Remarque

Si vous utilisez actuellement la syntaxe `this.refs.textInput` pour accéder aux refs, nous vous conseillons d'utiliser soit [l'approche avec fonction de rappel](#) soit [l'API `createRef`](#).

Limitations des refs avec fonctions de rappels

Si la fonction de rappel `ref` est définie à la volée, elle sera appelée deux fois à chaque mise à jour, d'abord avec `null` puis avec l'élément DOM. C'est parce qu'une nouvelle instance de la fonction est créée à chaque affichage, et React a besoin de nettoyer l'ancienne ref avant d'affecter la nouvelle. Vous pouvez éviter ça en définissant la fonction de rappel `ref` comme une méthode liée de la classe, même si ça ne devrait pas être gênant la plupart du temps.

[Go to TOC](#)

Canaux de sortie

React dépend d'une communauté open source florissante pour signaler des bugs, ouvrir des *pull requests* et [soumettre des RFC](#). Afin d'encourager des retours efficaces nous partageons parfois des builds spéciaux de React qui comportent des fonctionnalités encore officieuses.

Ce document sera surtout pertinent pour les développeur·se·s qui travaillent sur des frameworks, des bibliothèques ou de l'outillage de développement. Les personnes qui utilisent React principalement pour construire des applications destinées à des utilisateurs finaux ne devraient pas avoir à se préoccuper de nos canaux de sortie.

Chaque canal de sortie React est conçu pour des cas d'usages bien spécifiques :

- **Latest** fournit les versions stables de React, conformes à semver. C'est ce que vous obtenez en installant React depuis npm. C'est le canal que vous utilisez déjà aujourd'hui. **Utilisez ce canal pour toute application React visant des utilisateurs finaux.**
- **Next** est basée sur la branche `master` du dépôt de code source de React. Considérez-le comme des versions candidates de la prochaine version semver mineure. Utilisez-le pour faire des tests d'intégration entre React et des projets tiers.
- **Experimental** comprend les API et fonctionnalités expérimentales qu'on ne trouve pas dans les versions stables. C'est là aussi basé sur la branche `master`, mais des drapeaux de fonctionnalités supplémentaires ont été activés. Utilisez ce canal pour essayer des fonctionnalités à venir en amont de leur sortie officielle.

Toutes ces versions sont publiées sur npm, mais seul *Latest* utilise [une gestion sémantique des versions](#). Les pré-versions (celles des canaux *Next* et *Experimental*) ont des versions dont l'identifiant est un hash basé sur leur contenu, par exemple `0.0.0-1022ee0ec` pour *Next* et `0.0.0-experimental-1022ee0ec` pour *Experimental*.

Latest est le seul canal de sortie officiellement autorisé pour des applications visant les utilisateurs finaux. Les sorties des canaux *Next* et *Experimental* sont fournies dans une optique de tests uniquement, et leur comportement est susceptible de varier d'une sortie à l'autre. Elles ne respectent pas le protocole semver que nous utilisons pour les sorties du canal *Latest*.

En publiant les pré-versions sur le même référentiel que pour les versions stables, nous pouvons tirer parti des nombreux outils qui se basent sur npm pour fonctionner, tels que [unpkg](#) et [CodeSandbox](#).

Canal **Latest** {#latest-channel}

Le canal *Latest* est utilisé pour les versions stables de React. Il correspond à l'étiquette `latest` sur npm. C'est le canal recommandé pour toute appli React proposée à de véritables utilisateurs.

Si vous n'êtes pas sûr·e du canal que vous devriez utiliser, alors c'est *Latest*. Si vous développez avec React, c'est ce que vous utilisez déjà.

Vous pouvez vous attendre à ce que les mises à jour sur *Latest* soient extrêmement stables. On y suit un protocole sémantique de gestion des versions. Vous pouvez en apprendre davantage sur nos engagements en termes de stabilité et de migration incrémentielle en consultant notre [politique de gestion des versions](#).

Canal *Next* {#next-channel}

Le canal *Next* est un canal de pré-version qui piste la branche `master` du dépôt React. Nous utilisons les pré-versions du canal *Next* comme des versions candidates pour le canal *Latest*. Vous pouvez imaginer *Next* comme un sur-ensemble de *Latest*, mis à jour plus fréquemment.

L'ampleur des changements entre la version *Next* la plus récente et la dernière version *Latest* est à peu près du même ordre que ce que vous trouveriez entre deux versions semver mineures. En revanche, **le canal *Next* ne respecte pas la gestion sémantique des versions**. Vous êtes susceptible d'y rencontrer des ruptures de compatibilité ascendante d'une version à la suivante.

N'utilisez pas les pré-versions dans des applications pour utilisateurs finaux.

Les sorties de *Next* sont publiées avec l'étiquette `next` sur npm. Les numéros de versions sont générés sur la base d'un hash de leur contenu, par exemple `0.0.0-1022ee0ec`.

Utiliser le canal *Next* pour des tests d'intégration

Le canal *Next* est conçu pour permettre des tests d'intégration entre React et d'autres projets qui se basent dessus.

Toutes les modifications apportées à React passent par une phase poussée de tests internes avant d'être publiées. Cependant, l'écosystème React recèle une myriade d'environnements et de configurations, et il nous est parfaitement impossible de tester chaque combinaison.

Si vous êtes l'auteur·e d'une solution tierce pour React (framework, bibliothèque, outil de développement ou d'autres projets de type infrastructurel), vous pouvez nous aider à préserver la stabilité de React pour vos utilisateurs et la communauté React toute entière en exécutant régulièrement votre suite de tests avec les dernières modifications en date. Si cela vous intéresse, suivez ces étapes :

- Mettez en place une tâche périodique au sein de votre plate-forme préférée d'intégration continue. On trouve ce type d'exécutions périodiques tant dans [CircleCI](#) que dans [Travis CI](#).
- Dans la tâche, mettez à jour les modules React à partir de la version la plus récente du canal *Next*, grâce à l'étiquette `next` sur npm. Avec la CLI de npm :

```
npm update react@next react-dom@next
```

Ou avec Yarn :

```
yarn upgrade react@next react-dom@next
```

- Exécutez votre suite de tests avec les modules ainsi mis à jour.

- Si tout fonctionne, super ! Vous pouvez vous attendre à ce que votre projet continue à fonctionner avec la prochaine version mineure de React.
- Si quelque chose casse de façon inattendue, merci de nous le signaler en [créant un ticket](#).

Le projet Next.js (promis, on n'a pas fait exprès !) utilise cette approche. Vous pouvez consulter leur [configuration CircleCI](#) à titre d'exemple.

Canal *Experimental* {#experimental-channel}

Tout comme *Next*, le canal *Experimental* est un canal de pré-versions qui piste la branche `master` du dépôt React. Mais contrairement à *Next*, les sorties sur *Experimental* comportent des fonctionnalités et API qui ne sont pas encore prêtes à être diffusées plus largement.

En général, une mise à jour sur *Next* est accompagnée par une mise à jour correspondante sur *Experimental*. Elles sont basées sur la même révision de source, mais sont construites avec des jeux distincts de drapeaux de fonctionnalités.

Les versions sur *Experimental* peuvent différer considérablement de celles sur *Next* et *Latest*. **N'utilisez pas les pré-versions *Experimental* dans les applications pour utilisateurs finaux.** Vous pouvez vous attendre à des ruptures fréquentes de compatibilité ascendante dans les sorties du canal *Experimental*.

Les sorties de *Experimental* sont publiées avec l'étiquette `experimental` sur npm. Les numéros de versions sont générés sur la base d'un hash de leur contenu, par exemple `0.0.0-experimental-1022ee0ec`.

Que trouve-t-on dans une sortie *Experimental* ?

Les fonctionnalités expérimentales ne sont pas encore prêtes à être livrées à un large public, et sont susceptibles de changer radicalement d'ici leur finalisation. Certaines expériences n'aboutiront peut-être même jamais—c'est justement pour tester la viabilité de changements proposés que nous avons ces expériences.

Par exemple, si le canal *Experimental* avait existé quand nous avons annoncé les Hooks, nous aurions sorti les Hooks sur le canal *Experimental* plusieurs semaines avant qu'ils deviennent disponibles dans *Latest*.

Vous trouverez peut-être utile de lancer des tests d'intégration avec *Experimental*. Libre à vous. Ceci dit, gardez bien à l'esprit que *Experimental* est encore moins stable que *Next*. **Nous ne garantissons aucune stabilité d'une version de *Experimental* à une autre.**

Comment en apprendre davantage sur les fonctionnalités expérimentales ?

Les fonctionnalités expérimentales ne sont pas nécessairement documentées. Le plus souvent, les expériences ne sont documentées qu'une fois qu'elles sont sur le point d'être livrées dans *Next* ou *Latest*.

Si une fonctionnalité n'est pas documentée, elle peut toutefois être accompagnée d'une [RFC](#).

Lorsque nous sommes prêts à annoncer de nouvelles expériences, nous en parlons sur le [blog React](#), mais ça ne veut pas dire que nous annonçons chaque expérience.

Vous pouvez toujours vous référer à l'[historique](#) de notre dépôt public sur GitHub pour une liste exhaustive des modifications.

Props de rendu

Le terme « prop de rendu » (*render prop, NdT*) fait référence à une technique qui consiste à partager du code entre des composants React en utilisant une prop dont la valeur est une fonction.

Un composant avec une prop de rendu prend une fonction qui renvoie un élément React et l'appelle au lieu d'implémenter sa propre logique de rendu.

```
<DataProvider render={data => (
  <h1>Bonjour {data.target}</h1>
)}>
```

Des bibliothèques telles que [React Router](#), [Downshift](#) et [Formik](#) utilisent ces props de rendu.

Dans cette page, nous verrons en quoi les props de rendu sont pratiques, et comment vous pouvez écrire les vôtres.

Utiliser les props de rendu pour des questions transversales

Les composants sont l'unité de base de réutilisation de code dans React, mais il n'est pas toujours évident de partager l'état ou le comportement contenu dans un composant avec d'autres composants qui auraient besoin de ce même état.

Par exemple, le composant suivant piste la position de la souris dans une application web :

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        <h1>Déplacez votre souris sur l'écran !</h1>
        <p>La position actuelle de la souris est ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

Lorsque le curseur se déplace sur l'écran, le composant affiche ses coordonnées (x,y) dans un élément `<p>`.

La question qui se pose maintenant est : comment pouvons-nous réutiliser ce comportement dans un autre composant ? En d'autres termes, si un autre composant a besoin de connaître la position du curseur, pouvons-nous encapsuler ce comportement pour pouvoir facilement le partager avec ce composant ?

Puisque les composants sont l'unité de réutilisation de code dans React, essayons de refactoriser un peu le code pour pouvoir utiliser un composant `<Mouse>` qui encapsule le comportement dont nous avons besoin ailleurs.

```
// Le composant <Mouse> encapsule le comportement dont nous avons besoin...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        {/* ...mais comment afficher autre chose qu'un <p> ? */}
        <p>La position actuelle de la souris est ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <>
        <h1>Déplacez votre souris sur l'écran !</h1>
        <Mouse />
      </>
    );
  }
}
```

Le composant `<Mouse>` encapsule maintenant tous les comportements associés à l'écoute des événements `mousemove` et à la sauvegarde de la position (x, y) du curseur, mais il n'est pas encore tout à fait réutilisable.

Par exemple, supposons que nous avons un composant `<Cat>` qui affiche une image de chat pourchassant une souris sur l'écran. Nous pourrions utiliser une prop `mouse`, comme dans `<Cat mouse={{ x, y }}>`, pour transmettre au composant les coordonnées de la souris pour qu'il sache où positionner l'image sur l'écran.

Au premier essai, vous tenteriez peut-être d'afficher `<Cat>` dans la méthode `render` de `<Mouse>`, comme ceci :

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class MouseWithCat extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        /*
          Nous pourrions simplement remplacer le <p> par un <Cat> ici...
          mais nous devrions alors créer un composant séparé
          <MouseWithSomethingElse> chaque fois que nous voudrions l'utiliser.
          <MouseWithCat> n'est donc pas si réutilisable que ça pour le moment.
        */
        <Cat mouse={this.state} />
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Déplacez votre souris sur l'écran !</h1>
        <MouseWithCat />
      </div>
    );
  }
}
```

Cette approche fonctionnera dans notre cas particulier, mais nous n'avons pas atteint notre objectif qui consiste à vraiment encapsuler le comportement de façon réutilisable. En effet, chaque fois que nous aurons besoin de la position de la souris pour un cas d'utilisation différent, nous devrons créer un nouveau composant (pour ainsi dire un autre `<MouseWithCat>`) spécifique à ce cas.

C'est là que la prop de rendu entre en scène : au lieu d'écrire en dur un composant `<Cat>` dans un composant `<Mouse>`, et changer le résultat de l'affichage, nous pouvons créer `<Mouse>` avec une prop qui prendra une fonction permettant de déterminer dynamiquement ce qui doit être affiché : une prop de rendu.

```

class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        /*
          Au lieu de fournir une représentation statique de ce qu'affiche <Mouse>,
          utilisez la prop `render` pour déterminer dynamiquement quoi afficher.
        */
        {this.props.render(this.state)}
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Déplacez votre souris sur l'écran !</h1>
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}/>
      </div>
    );
  }
}

```

Dorénavant, au lieu de cloner le composant `<Mouse>` et d'écrire en dur quelque chose d'autre dans sa méthode `render` pour s'adapter à un cas d'utilisation, nous fournissons une prop `render` que `<Mouse>` peut utiliser pour déterminer dynamiquement quoi afficher.

Plus concrètement, **une prop de rendu est une prop de type fonction qu'un composant utilise pour savoir quoi afficher.**

Cette technique facilite énormément le partage d'un comportement. Pour récupérer ce comportement, il suffit d'afficher un `<Mouse>` avec une prop `render` qui lui dit quoi afficher avec les coordonnées (x, y) actuelles du curseur.

Un point intéressant à noter concernant les props de rendu est que vous pouvez implémenter la plupart des **composants d'ordre supérieur** (*Higher-Order Components, ou HOC, NdT*) en utilisant un composant classique avec une prop de rendu. Par exemple, si vous préférez avoir un HOC `withMouse` au lieu d'un composant `<Mouse>`, vous pouvez facilement en créer un en utilisant un composant `<Mouse>` avec une prop de rendu :

```
// Si, pour une raison ou une autre, vous voulez vraiment utiliser un HOC, vous
// pouvez
// facilement en créer un en utilisant un composant classique avec une prop de
// rendu !
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}>
      );
    }
  }
}
```

Utiliser une prop de rendu rend donc possible l'utilisation des deux méthodes.

Utiliser d'autres props que `render` `{#using-props-other-than-render}`

Il est important de se rappeler que ce n'est pas parce que la technique s'appelle « props de rendu » qu'il est obligatoire d'utiliser une prop appelée `render` pour la mettre en œuvre. En fait, *n'importe quelle prop utilisée par un composant pour savoir quoi renvoyer est techniquement une « prop de rendu ».*

Même si les exemples ci-dessus utilisent `render`, nous pourrions tout aussi simplement utiliser la prop `children` !

```
<Mouse children={mouse => (
  <p>La position de la souris est {mouse.x}, {mouse.y}</p>
)}>
```

Et rappelez-vous, la propriété `children` n'a en fait pas besoin d'être nommée dans la liste des « attributs » de votre élément JSX. Au lieu de ça, vous pouvez l'utiliser directement *dans* l'élément !

```
<Mouse>
  {mouse => (
    <p>La position de la souris est {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

Vous pouvez découvrir l'utilisation de cette technique dans l'API [react-motion](#).

Comme cette technique est un peu inhabituelle, vous aurez probablement envie de préciser que `children` devrait être une fonction dans vos `propTypes` au moment de concevoir votre API de cette façon.

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

Limitations

Soyez prudent·e lors de l'utilisation de props de rendu avec React.PureComponent

Utiliser une prop de rendu peut contrebalancer l'avantage apporté par l'utilisation de `React.PureComponent` si vous créez la fonction dans une méthode `render`. C'est dû au fait que la comparaison superficielle des props renverra toujours `false` pour les nouvelles props, et que dans ce cas chaque `render` générera justement une nouvelle valeur pour la prop de rendu.

Pour revenir sur l'exemple de notre composant `<Mouse>`, si `Mouse` étendait `React.PureComponent` au lieu de `React.Component`, notre exemple ressemblerait à ça :

```
class Mouse extends React.PureComponent {
  // Même implémentation que plus haut...
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Déplacez votre souris sur l'écran !</h1>

        /*
          Il ne faut pas faire ça ! La valeur de la prop `render`
          sera différente à chaque rendu.
        */
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}>
      </div>
    );
  }
}
```

Dans cet exemple, à chaque fois que `<MouseTracker>` s'affiche, une nouvelle fonction est générée comme valeur de la prop `<Mouse render>`, ce qui annule l'optimisation recherchée à la base quand on a fait en sorte que `<Mouse>` étende `React.PureComponent` !

Pour éviter ce problème; vous pouvez parfois définir la prop comme une méthode d'instance, de la façon suivante :

```
class MouseTracker extends React.Component {  
  // Définie comme une méthode d'instance, `this.renderTheCat` se réfèrera  
  // toujours à la *même* fonction quand nous l'utiliserons dans le rendu.  
  renderTheCat(mouse) {  
    return <Cat mouse={mouse} />;  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>Déplacez votre souris sur l'écran !</h1>  
        <Mouse render={this.renderTheCat} />  
      </div>  
    );  
  }  
}
```

Dans les cas où vous ne pourriez pas définir la prop de manière statique (ex. parce qu'elle utilise des valeurs locales à `render`), `<Mouse>` devrait plutôt étendre `React.Component`.

Le rendu des éléments

Les éléments sont les blocs élémentaires d'une application React.

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

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

Contrairement aux éléments DOM d'un navigateur, les éléments React sont de simples objets peu coûteux à créer. React DOM se charge de mettre à jour le DOM afin qu'il corresponde aux éléments React.

Remarque

On pourrait confondre les éléments avec le concept plus répandu de « composants ». Nous présenterons les composants dans la [prochaine section](#). Les éléments représentent la base des composants, aussi nous vous conseillons de bien lire cette section avant d'aller plus loin.

Afficher un élément dans le DOM

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

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

Nous parlons de nœud DOM « racine » car tout ce qu'il contient sera géré par React DOM.

Les applications développées uniquement avec React ont généralement un seul nœud DOM racine. Si vous intégrez React dans une application existante, vous pouvez avoir autant de nœuds DOM racines isolés que vous le souhaitez.

Pour faire le rendu d'un élément React dans un nœud DOM racine, passez les deux à la méthode `ReactDOM.render()` :

```
embed:rendering-elements/render-an-element.js
```

[Essayer sur CodePen](#)

Cet exemple de code affichera « Bonjour, monde » sur la page.

Mettre à jour un élément affiché

Les éléments React sont [immuables](#). Une fois votre élément créé, vous ne pouvez plus modifier ses enfants ou ses attributs. Un élément est comme une image d'un film à un instant T : il représente l'interface utilisateur à un point précis dans le temps.

Avec nos connaissances actuelles, la seule façon de mettre à jour l'interface utilisateur est de créer un nouvel élément et de le passer à `ReactDOM.render()`.

Prenons l'exemple de cette horloge :

```
embed:rendering-elements/update-rendered-element.js
```

[Essayer dans CodePen](#)

À chaque seconde, nous appelons `ReactDOM.render()` depuis une fonction de rappel passée à `setInterval()`.

Remarque

En pratique, la plupart des applications React n'appellent `ReactDOM.render()` qu'une seule fois. Dans les prochaines sections, nous apprendrons comment encapsuler un tel code dans des [composants à état](#).

Nous vous conseillons de lire les sujets abordés dans l'ordre car ils s'appuient l'un sur l'autre.

React met à jour le strict nécessaire

React DOM compare l'élément et ses enfants avec la version précédente, et applique uniquement les mises à jour DOM nécessaires pour refléter l'état voulu.

Vous pouvez vérifier ce comportement en inspectant le [dernier exemple](#) avec les outils de développement 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>
```

L'inspecteur montrant des mises à jour atomiques

Même si nous créons à chaque seconde un élément décrivant l'arborescence complète de l'interface utilisateur, seul le nœud texte dont le contenu a été modifié est mis à jour par React DOM.

L'expérience nous montre que réfléchir à quoi devrait ressembler une interface utilisateur à un moment donné plutôt que de réfléchir à comment elle devrait évoluer permet d'éliminer toute une catégorie de bugs.

État et cycle de vie

Cette page présente les concepts d'état local et de cycle de vie dans un composant React. Vous pouvez trouver [la référence d'API des composants ici](#).

Prenons l'exemple de l'horloge dans [une des sections précédentes](#). Dans [Le rendu des éléments](#), nous avons appris une seule façon de mettre à jour l'interface utilisateur (UI). On appelle `ReactDOM.render()` pour changer la sortie rendue :

```
function tick() {
  const element = (
    <div>
      <h1>Bonjour, monde !</h1>
      <h2>Il est {new Date().toLocaleTimeString()}</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

[Essayer sur CodePen](#)

Dans cette section, nous allons apprendre à faire un composant `Clock` vraiment réutilisable et isolé. Il mettra en place son propre minuteur et se mettra à jour tout seul à chaque seconde.

Nous commençons par isoler l'apparence de l'horloge :

```
function Clock(props) {
  return (
    <div>
      <h1>Bonjour, monde !</h1>
      <h2>Il est {props.date.toLocaleTimeString()}</h2>
    </div>
  );
}

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

setInterval(tick, 1000);
```

[Essayer sur CodePen](#)

Cependant, il manque une contrainte cruciale : le fait que la `Clock` mette en place le minuteur et mette à jour son interface utilisateur devrait être un détail d'implémentation de la `Clock`.

Idéalement, on veut écrire ceci une seule fois et voir la `Clock` se mettre à jour elle-même :

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

Pour implémenter ça, on a besoin d'ajouter un « état local » au composant `Horloge`.

L'état local est similaire aux props, mais il est privé et complètement contrôlé par le composant.

Convertir 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-y une méthode vide appelée `render()`.
3. Déplacez le corps de la fonction dans la méthode `render()`.
4. Remplacez `props` par `this.props` dans le corps de la méthode `render()`.
5. Supprimez la déclaration désormais vide de la fonction.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.props.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

[Essayer sur CodePen](#)

Le composant `Clock` est maintenant défini comme une classe au lieu d'une fonction.

La méthode `render` sera appelée à chaque fois qu'une mise à jour aura lieu, mais tant que l'on exploite le rendu de `<Clock />` dans le même nœud DOM, une seule instance de la classe `Clock` sera utilisée. Cela nous permet d'utiliser des fonctionnalités supplémentaires telles que l'état local et les méthodes de cycle de vie.

Ajouter un état local à une classe

Nous allons déplacer la `date` des props vers l'état en trois étapes :

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

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

2. Ajoutez un constructeur de classe qui initialise `this.state` :

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

  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Notez que l'on passe `props` au constructeur de base :

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

Les composants à base de classe devraient toujours appeler le constructeur de base avec `props`.

3. Supprimez la prop `date` de l'élément `<Clock />` :

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

Nous rajouterons plus tard le code du minuteur dans le composant lui-même.

Le résultat ressemble à ceci :

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

  render() {
    return (
      <div>
```

```

        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
    </div>
);
}

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

```

[Essayer sur CodePen](#)

Ensuite, nous allons faire en sorte que le composant `Clock` mette en place son propre minuteur et se mette à jour toutes les secondes.

Ajouter des méthodes de cycle de vie à une classe

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

Nous voulons [mettre en place un minuteur](#) quand une `Horloge` apparaît dans le DOM pour la première fois. Le terme React « montage » désigne cette phase.

Nous voulons également [nettoyer le minuteur](#) quand le DOM produit par l'`Horloge` est supprimé. En React, on parle de « démontage ».

Nous pouvons déclarer des méthodes spéciales sur un composant à base de classe pour exécuter du code quand 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>Bonjour, monde !</h1>
                <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
            </div>
        );
    }
}

```

On les appelle des « méthodes de cycle de vie ».

La méthode `componentDidMount()` est exécutée après que la sortie du composant a été injectée dans le DOM. C'est un bon endroit pour mettre en place le minuteur :

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

Notez qu'on a enregistré l'ID du minuteur directement sur `this (this.timerID)`.

Alors que `this.props` est mis en place par React lui-même et que `this.state` a un sens bien spécial, vous pouvez très bien ajouter manuellement d'autres champs sur la classe si vous avez besoin de stocker quelque chose qui ne participe pas au flux de données (comme un ID de minuteur).

Nous allons détruire le minuteur dans la méthode de cycle de vie `componentWillUnmount()` :

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

Enfin, nous allons implémenter une méthode appelée `tick()` que le composant `Clock` va exécuter toutes les secondes.

Elle utilisera `this.setState()` pour planifier une mise 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>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

```

    }
}

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

```

Essayer sur CodePen

Maintenant l'horloge se met à jour toutes les secondes.

Récapitulons ce qui se passe et l'ordre dans lequel les méthodes sont invoquées :

1. Quand `<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 contenant l'heure actuelle. Nous mettrons cet état à jour par la suite.
2. React appelle ensuite la méthode `render()` du composant `Clock`. C'est comme cela que React découvre ce qu'il faut afficher à l'écran. React met ensuite à jour le DOM pour correspondre à la sortie de la méthode `render()` du composant `Clock`.
3. Quand la sortie de la `Clock` est insérée dans le DOM, React appelle la méthode de cycle de vie `componentDidMount()`. À l'intérieur, le composant `Clock` demande au navigateur de mettre en place un minuteur pour appeler la méthode `tick()` du composant une fois par seconde.
4. Chaque seconde, le navigateur appelle la méthode `tick()`. À l'intérieur, le composant `Clock` planifie une mise à jour de l'interface utilisateur en appelant `setState()` avec un objet contenant l'heure actuelle. Grâce à l'appel à `setState()`, React sait que l'état a changé, et invoque à nouveau la méthode `render()` pour savoir ce qui devrait être affiché à l'écran. Cette fois, la valeur de `this.state.date` dans la méthode `render()` est différente, la sortie devrait donc inclure l'heure mise à jour. React met à jour le DOM en accord avec cela.
5. Si le composant `Clock` finit par être retiré du DOM, React appellera la méthode de cycle de vie `componentWillUnmount()` pour que le minuteur soit arrêté.

Utiliser l'état local correctement

Il y'a trois choses que vous devriez savoir à propos de `setState()`.

Ne modifiez pas l'état directement

Par exemple, ceci ne déclenchera pas un rafraîchissement du composant :

```
// Erroné
this.state.comment = 'Bonjour';
```

À la place, utilisez `setState()` :

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

Le seul endroit où vous pouvez affecter `this.state`, c'est le constructeur.

Les mises à jour de l'état peuvent être asynchrones

React peut grouper plusieurs appels à `setState()` en une seule mise à jour pour des raisons de performance.

Comme `this.props` et `this.state` peuvent être mises à jour de façon asynchrone, vous ne devez pas vous baser sur leurs valeurs pour calculer le prochain état.

Par exemple, ce code peut échouer pour mettre à jour un compteur :

```
// Erroné
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

Pour remédier à ce problème, utilisez la seconde forme de `setState()` qui accepte une fonction à la place d'un objet. Cette fonction recevra l'état précédent comme premier argument et les props au moment de la mise à jour comme second argument :

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

Nous avons utilisé une [fonction fléchée](#) ci-dessus, mais une fonction normale marche aussi :

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

Les mises à jour de l'état sont fusionnées

Quand vous invoquez `setState()`, React fusionne les objets que vous donnez avec l'état actuel.

Par exemple, votre état peut contenir plusieurs variables indépendantes :

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

Ensuite, vous pouvez les mettre à jour indépendamment avec des appels séparés à `setState()` :

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
```

```

        posts: response.posts
    });
});

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

```

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

Les données descendantes

Ni parent ni enfant ne peuvent savoir si un certain composant est à état ou non, et ne devraient pas se soucier de savoir s'il est défini par une fonction ou une classe.

C'est pourquoi on dit souvent que l'état est local ou encapsulé. Il est impossible d'y accéder depuis un autre composant.

Un composant peut choisir de passer son état à ses enfants via des props :

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

Le composant `FormattedDate` reçoit la `date` dans ses props et ne sait pas si elle vient de l'état de la `Clock`, des props de la `Clock`, ou a été tapée à la main :

```

function FormattedDate(props) {
  return <h2>Il est {props.date.toLocaleTimeString()}</h2>;
}

```

[Essayer sur CodePen](#)

On appelle souvent cela un flux de données « du haut vers le bas » ou « unidirectionnel ». Un état local est toujours possédé par un composant spécifique, et toute donnée ou interface utilisateur dérivée de cet état ne peut affecter que les composants « en-dessous » de celui-ci dans l'arbre de composants.

Si vous imaginez un arbre de composants comme une cascade de props, chaque état de composant est une source d'eau supplémentaire qui rejoint la cascade à un point quelconque, mais qui coule également vers le bas.

Pour démontrer que tous les composants sont réellement isolés, nous pouvons créer un composant `App` qui affiche trois `<Clock>`s :

```

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

```

```
    );
}

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

[Essayer sur CodePen](#)

Chaque `Clock` met en place son propre minuteur et se met à jour indépendamment.

Dans une application React, le fait qu'un composant soit à état ou non est considéré comme un détail d'implémentation du composant qui peut varier avec le temps. Vous pouvez utiliser des composants sans état à l'intérieur de composants à état, et vice-versa.

La validation de types statique

Les systèmes de typage statique tels que [Flow](#) et [TypeScript](#) parviennent à identifier certains types de problèmes avant même d'exécuter votre code. Ils peuvent aussi améliorer le processus de travail des développeurs en y ajoutant des assistances telles que l'auto-complétion. C'est pourquoi nous recommandons l'utilisation de Flow ou TypeScript au lieu des [PropTypes](#) pour les grandes bases de code source.

Flow

[Flow](#) est un système de typage statique pour votre code JavaScript. Il est développé chez Facebook et souvent utilisé avec React. Il permet d'annoter les variables, fonctions et composants React avec une syntaxe spéciale de typage, et de déceler préventivement les erreurs. Vous pouvez lire [cette introduction à Flow](#) pour en apprendre les bases.

Pour utiliser Flow, vous devrez :

- Ajouter Flow comme dépendance dans votre projet.
- Vous assurer que la syntaxe Flow soit bien enlevée du code, une fois compilé.
- Ajouter les annotations de typage et lancer Flow pour les vérifier.

Nous allons vous expliquer ces étapes en détail ci-dessous.

Ajouter Flow à un projet

Premièrement, naviguez vers le dossier de votre projet dans le terminal (ou l'invite de commandes). Vous allez devoir exécuter la commande suivante :

Si vous utilisez [Yarn](#) :

```
yarn add --dev flow-bin
```

Si vous utilisez [npm](#) :

```
npm install --save-dev flow-bin
```

Cette commande installe la dernière version de Flow dans votre projet.

Maintenant, ajoutez `flow` à la section `"scripts"` de votre `package.json` afin de pouvoir l'utiliser dans votre terminal :

```
{
  // ...
  "scripts": {
    "flow": "flow",
    // ...
  },
  // ...
}
```

Enfin, exécutez l'une des commandes suivantes :

Si vous utilisez [Yarn](#) :

```
yarn run flow init
```

Si vous utilisez [npm](#) :

```
npm run flow init
```

Cette commande va créer un fichier de configuration Flow qu'il va falloir ajouter à la gestion de versions de votre code source.

Retirer la syntaxe Flow du code compilé

Flow est une extension du langage JavaScript avec une syntaxe spéciale pour les annotations de type. Cependant les navigateurs ne connaissent pas cette syntaxe. Nous devons donc nous assurer qu'elle ne finisse pas dans le *bundle* de JavaScript compilé qui sera à terme envoyé au navigateur.

Le procédé exact pour y arriver dépend des outils que vous utilisez pour compiler votre code source JavaScript.

Create React App

Si vous avez utilisé [Create React App](#) pour initialiser votre projet, félicitations ! Dans ce scénario, les annotations Flow sont déjà retirées par défaut, donc vous n'avez rien à faire de plus pour cette étape.

Babel

Remarque

Ces instructions *ne sont pas adaptées* aux utilisateurs de Create React App. Bien que Create React App utilise Babel en interne, il est déjà configuré pour comprendre Flow. Ne suivez cette étape que si vous n'utilisez *pas* Create React App.

Si vous avez configuré Babel manuellement pour votre projet, il vous faudra ajouter un *preset* spécial pour prendre en charge Flow.

Si vous utilisez [Yarn](#), exécutez :

```
yarn add --dev @babel/preset-flow
```

Si vous utilisez [npm](#), exécutez :

```
npm install --save-dev @babel/preset-flow
```

Ensuite, ajoutez le preset `flow` à votre [configuration Babel](#). Par exemple, si vous configurez Babel avec le fichier `.babelrc`, ça pourrait ressembler à ceci :

```
{
  "presets": [
    "@babel/preset-flow",
    "react"
  ]
}
```

De cette façon, vous pourrez utiliser la syntaxe Flow dans votre code.

Remarque

Il n'est pas nécessaire d'avoir le preset `react` afin d'utiliser Flow, mais ils sont souvent utilisés ensemble. Flow comprend déjà quant à lui la syntaxe JSX.

Autres configurations

Si vous n'utilisez ni Create React App, ni Babel, vous pouvez utiliser [flow-remove-types](#) pour enlever les annotations de type.

Exécuter Flow

Si vous avez bien suivi les instructions ci-dessus, vous devriez être prêt·e à exécuter Flow pour la première fois.

Si vous utilisez [Yarn](#), exécutez :

```
yarn flow
```

Si vous utilisez [npm](#), exécutez :

```
npm run flow
```

Vous devriez voir apparaître un message similaire à celui-ci :

```
No errors!
✨ Done in 0.17s.
```

Ajouter des annotations de type Flow

Par défaut, Flow ne vérifie que les fichiers qui contiennent cette annotation :

```
// @flow
```

Elle est habituellement placée au début du document. Essayez de l'ajouter à quelques fichiers dans votre projet, et exécutez `yarn flow` ou `npm run flow` pour voir si Flow a déjà découvert des problèmes.

Il existe aussi [une option](#) pour forcer Flow à vérifier *tous* les fichiers, même sans l'annotation. Ça peut être trop pour des projets existants, mais raisonnable pour un nouveau projet si vous souhaitez le typer avec Flow de façon intégrale.

Vous êtes paré·e ! Nous vous conseillons d'aller voir les ressources suivantes pour en apprendre davantage sur Flow :

- Documentation Flow : les annotations de type
- Documentation Flow : éditeurs de code
- Documentation Flow : React
- Le linting avec Flow

TypeScript

TypeScript est un langage de programmation développé par Microsoft. C'est un sur-ensemble typé de JavaScript, et il fournit son propre compilateur. Étant un langage typé, TypeScript peut trouver des erreurs et bugs lors de la compilation, bien avant que l'application ne soit déployée. Vous trouverez plus d'informations sur l'utilisation de TypeScript avec React [ici](#).

Pour utiliser TypeScript, vous devez :

- Ajouter la dépendance TypeScript dans votre projet
- Configurer les options du compilateur TypeScript
- Utiliser les extensions appropriées pour vos fichiers
- Ajouter les définitions de type pour les bibliothèques que vous utilisez

Voyons ça plus en détail.

Utiliser TypeScript avec Create React App

Create React App prend déjà en charge TypeScript.

Pour créer un **nouveau projet** avec la prise en charge de TypeScript, exécutez :

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

Vous pouvez aussi l'ajouter à **un projet Create React App existant**, [comme documenté ici](#).

Remarque

Si vous utilisez Create React App, vous pouvez **sauter le reste de cette page**. Elle décrit l'installation manuelle qui ne s'applique pas aux utilisateurs de Create React App.

Ajouter TypeScript à un projet

On commence par exécuter une commande dans le terminal.

Si vous utilisez [Yarn](#) :

```
yarn add --dev typescript
```

Si vous utilisez `npm` :

```
npm install --save-dev typescript
```

Félicitations ! Vous venez d'installer la dernière version de TypeScript dans votre projet. L'installation de TypeScript rend disponible la commande `tsc`. Avant de traiter la configuration, ajoutons `tsc` à la section `"scripts"` de notre `package.json` :

```
{
  // ...
  "scripts": {
    "build": "tsc",
    // ...
  },
  // ...
}
```

Configurer le compilateur TypeScript

Le compilateur ne nous aidera pas tant qu'on ne lui aura pas dit quoi faire. En TypeScript, ces règles sont définies dans un fichier spécial appelé `tsconfig.json`. Pour générer ce fichier :

Si vous utilisez `Yarn`, exécutez :

```
yarn run tsc --init
```

Si vous utilisez `npm`, exécutez :

```
npx tsc --init
```

Allez voir le `tsconfig.json` qui vient d'être généré : vous voyez qu'il y a de nombreuses options qu'on peut utiliser pour configurer le compilateur. Pour une explication détaillée des options, voyez [cette page](#).

Parmi les nombreuses options, nous allons regarder `rootDir` et `outDir`. Conformément à son rôle, le compilateur accepte des fichiers TypeScript et génère des fichiers JavaScript. Mais nous ne voudrions pas confondre les fichiers sources avec le résultat de la compilation.

Nous allons régler ça en deux étapes :

- D'abord, réorganisons notre projet. Nous allons placer tout notre code source à l'intérieur du dossier `src`.

```
└── package.json
└── src
  └── index.ts
└── tsconfig.json
```

- Ensuite, nous allons dire au compilateur où se trouve notre code source et où déposer le résultat.

```
// tsconfig.json
{
  "compilerOptions": {
```

```
// ...
  "rootDir": "src",
  "outDir": "build"
// ...
},
}
```

Génial ! Désormais, quand on exécute notre script, le compilateur va déposer le JS généré dans le dossier `build`. Le [TypeScript React Starter](#) fournit un `tsconfig.json` avec un bon ensemble de règles pour vous aider à démarrer.

En règle générale, vous ne souhaitez pas enregistrer le JS généré dans votre système de gestion de versions, aussi veillez à bien ajouter le dossier de sortie dans votre `.gitignore`.

Extensions de fichiers

Avec React, vous écrivez probablement vos composants dans un fichier `.js`. En TypeScript, il existe deux extensions de fichier : `.ts` est l'extension par défaut, tandis que `.tsx` est une extension spéciale pour les fichiers qui contiennent du JSX.

Exécuter TypeScript

Si vous avez bien suivi les instructions ci-dessus, vous devriez pouvoir exécuter TypeScript pour la première fois.

Si vous utilisez [Yarn](#) :

```
yarn build
```

Si vous utilisez [npm](#) :

```
npm run build
```

Si aucun message ne s'affiche, ça veut dire que tout a bien fonctionné.

Définitions de types

Afin de pouvoir afficher les erreurs et conseils des autres modules, le compilateur a besoin de fichiers de déclarations. Un fichier de déclarations contient toutes les informations de typage d'une bibliothèque. Ça nous permet d'utiliser dans notre projet des bibliothèques JS telles que celles que l'on trouve sur npm.

Il y a deux façons principales d'obtenir les déclarations de types d'une bibliothèque :

Inclure – La bibliothèque fournit déjà son fichier de déclarations. Ça nous arrange, il nous suffira d'installer la bibliothèque et nous pourrons l'utiliser sans tarder. Pour déterminer si une bibliothèque contient ses types, cherchez un fichier `index.d.ts` dans son projet. Certaines bibliothèques l'annoncent dans leur `package.json` avec le champ `typings` ou `types`.

DefinitelyTyped – DefinitelyTyped est un énorme référentiel de déclarations pour les bibliothèques qui n'incluent pas leur fichier de déclarations. Les déclarations sont faites par tout le monde et gérées par Microsoft et des contributeurs en logiciel libre. React par exemple n'inclut pas de fichier de déclarations, mais on peut l'obtenir sur DefinitelyTyped. Pour ce faire, exécutez cette commande dans votre terminal.

```
# yarn
yarn add --dev @types/react

# npm
npm i --save-dev @types/react
```

Déclarations locales – Parfois, un module que vous souhaitez utiliser ne contient pas de déclarations et n'est pas non plus disponible sur DefinitelyTyped. Dans ce cas, on peut créer un fichier de déclarations local. Créez un fichier `declarations.d.ts` à la racine de votre dossier source. Une déclaration simple pourrait ressembler à ceci :

```
declare module 'querystring' {
  export function stringify(val: object): string
  export function parse(val: string): object
}
```

Vous êtes maintenant prêt·e à écrire du code ! Nous vous conseillons de jeter un coup d'œil aux ressources suivantes pour en apprendre davantage sur TypeScript :

- Documentation TypeScript : les types de base
- Documentation TypeScript : migrer depuis JavaScript
- Documentation TypeScript : React et Webpack

Reason

Reason n'est pas un nouveau langage, c'est une nouvelle syntaxe et une boîte à outils par-dessus le langage reconnu OCaml. Reason donne à OCaml une syntaxe qui plaira aux développeurs JavaScript, et s'intègre bien avec les processus de développement basés sur npm et Yarn que les développeurs utilisent quotidiennement.

Reason est développé chez Facebook, qui l'utilise pour certains de ses produits tels que Messenger. Il est encore relativement expérimental mais a des intégrations React dédiées maintenues par Facebook et par une communauté dynamique.

Kotlin

Kotlin est un langage au typage statique développé par JetBrains. Ses plateformes cibles sont entre autres la JVM, Android, LLVM et JavaScript.

JetBrains développe et maintient certains outils spécialement pour la communauté React : des intégrations React ainsi que Create React Kotlin App. Ce dernier est conçu pour vous aider à commencer à construire des applications React avec Kotlin sans avoir à vous farcir une configuration de construction.

Autres langages

Remarquez qu'il existe d'autres langages au typage statique qui compilent vers JavaScript et sont donc compatibles avec React. Par exemple, [F#/Fable](#) avec [elmish-react](#). Allez voir leurs sites respectifs pour plus d'informations et n'hésitez pas à ajouter sur cette page d'autres langages au typage statique fonctionnant avec React !

[Go to TOC](#)

Le mode strict

`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'impacteront 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

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
```

Exemple de message d'avertissement sur les méthodes dépréciées

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

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

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.

Détecter l'API dépréciée de Contexte

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>

Avertissement du mode strict à l'utilisation de l'API dépréciée de Contexte

Lisez la documentation de la nouvelle API de Contexte pour faciliter la migration vers cette nouvelle version.

Environnements de test

Ce document détaille les facteurs susceptibles d'affecter votre environnement, et fournit des conseils pour quelques scénarios.

Harnais de test

Les harnais de test tels que [Jest](#), [mocha](#) ou [ava](#) vous permettent d'écrire des suites de tests en utilisant du JavaScript classique, et les exécutent dans le cadre de votre processus de développement. Qui plus est, les suites de tests peuvent être exécutées au sein de votre intégration continue.

- Jest est largement compatible avec les projets React, en offrant des fonctionnalités telles que l'isolation des [modules](#) et des [horloges](#), ainsi qu'une prise en charge de [jsdom](#). **Si vous utilisez Create React App, Jest est fourni de base avec des réglages par défaut très utiles.**
- Les bibliothèques comme [mocha](#) sont bien adaptées à une utilisation au sein de véritables navigateurs, et pourraient vous aider pour des tests ayant explicitement besoin de ce type de contexte d'exécution.
- Les tests de bout en bout sont surtout pertinents pour des scénarios plus longs impliquant plusieurs pages successives, et requièrent une [mise en place distincte](#).

Simuler une surface d'affichage

Les tests s'exécutent souvent dans un environnement qui n'est pas doté d'une véritable surface d'affichage (comme le serait un navigateur). Pour ces environnements-là, nous vous conseillons de simuler un navigateur avec [jsdom](#), une implémentation légère de navigateur qui s'exécute dans un processus Node.js.

La plupart du temps, jsdom se comporte comme un navigateur classique, à ceci près qu'il ne fournit pas des aspects comme [la mise en page et la navigation](#). Ça reste utile pour la plupart des tests de composants basés web, dans la mesure où il s'exécute bien plus vite que lorsque vous devez démarrer un navigateur pour chaque test. Qui plus est, il s'exécute dans le même processus que vos tests, de sorte qu'il est facile d'écrire du code qui examine le DOM obtenu et vérifie son contenu.

Tout comme un véritable navigateur, jsdom nous permet de modéliser les interactions utilisateurs ; vos tests peuvent déclencher des événements sur les nœuds du DOM, puis observer le DOM et vérifier les effets de bord de ces actions ([exemple](#)).

Une large part des tests d'interface utilisateur (UI) peuvent être écrits de cette façon : en utilisant Jest comme harnais, en réalisant le rendu avec jsdom, et en spécifiant les interactions utilisateurs sous forme de séquences d'événements au sein du navigateur, à l'aide de la fonction utilitaire [act\(\)](#) ([exemple](#)). Par exemple, de très nombreux tests de React lui-même sont écrits ainsi.

Si vous écrivez une bibliothèque qui doit tester principalement des comportements spécifiques aux navigateurs, et requiert donc des comportements natifs du navigateur tels que la mise en page ou de véritables champs de saisie, il vous faudra alors plutôt une solution comme [mocha](#).

Dans un environnement où vous *ne pouvez pas* simuler un DOM (par exemple en testant des composants React Native depuis Node.js), vous pourriez opter pour des [utilitaires de simulation d'événements](#), qui simulent les interactions avec les éléments. Une autre option consisterait à utiliser la fonction utilitaire `fireEvent` fournie par `@testing-library/react-native`.

Les frameworks comme [Cypress](#), [puppeteer](#) et [webdriver](#) sont quant à eux utiles pour exécuter des [tests de bout en bout](#).

Simuler des fonctions

Lorsqu'on écrit des tests, on aimerait les isoler des parties de notre code qui n'ont pas d'équivalent dans notre environnement de test (par exemple pour vérifier l'état de `navigator.onLine` alors qu'on est en Node.js). Les tests peuvent aussi surveiller certaines fonctions, et observer l'interaction entre ces fonctions et d'autres parties du test. Il est alors utile de pouvoir simuler sélectivement ces fonctions en les remplaçant par des versions adaptées à nos tests.

C'est particulièrement vrai pour le chargement de données. Il est généralement préférable d'utiliser de « fausses » données pour nos tests, afin d'éviter la lenteur et la fragilité d'une récupération depuis de véritables points d'accès API ([exemple](#)). On gagne ainsi en fiabilité pour nos tests. Des bibliothèques comme [Jest](#) et [sinon](#), entre autres, permettent de simuler les fonctions. Pour des tests de bout en bout, il peut être un peu plus délicat de simuler complètement le réseau, mais vous voudrez sans doute utiliser de véritables points d'accès API dans ces tests-là de toutes façons.

Simuler des modules

Certains composants ont des dépendances à des modules qui peuvent ne pas bien fonctionner dans un environnement de test, ou ne sont pas essentiels à nos tests. Il peut alors être utile de les simuler en les remplaçant par des versions appropriées ([exemple](#)).

Dans Node.js, les harnais comme Jest [permettent la simulation de modules](#). Vous pouvez aussi recourir à des bibliothèques telles que `mock-require`.

Simuler des horloges

Vos composants pourront parfois utiliser des fonctionnalités sensibles au temps, telles que `setTimeout`, `setInterval`, ou `Date.now`. Dans des environnements de test, il peut être pratique de les remplacer par des versions qui vous permettent de faire manuellement « avancer » le temps. C'est super pour garantir que nos tests s'exécutent rapidement ! Les tests basés sur des horloges seraient toujours exécutés dans l'ordre, juste plus vite ([exemple](#)). La plupart des frameworks, y compris [Jest](#), [sinon](#) et [lolex](#), vous permettent de simuler les horloges dans vos tests.

Parfois, vous voudrez sans doute ne pas simuler les horloges, par exemple si vous testez une animation, ou une interaction avec un point d'accès sensible au temps (tel qu'une API imposant une limitation de trafic). Les bibliothèques qui simulent les horloges vous permettent d'activer ou désactiver ces interceptions par test ou par suite de tests, de sorte que vous pouvez choisir explicitement dans quels tests ces simulations sont exploitées.

Tests de bout en bout

Les tests de bout en bout sont utiles pour tester des scénarios plus longs, en particulier s'ils sont critiques à votre activité (comme des paiements ou des inscriptions). Pour de tels tests, vous voudrez sans doute tester comment un véritable navigateur affiche votre appli entière, charge les données depuis de véritables points d'accès API, utilise les sessions et les cookies, navigue au travers de divers liens... Vous voudrez sans doute aussi vérifier non seulement l'état du DOM, mais les données sous-jacentes (par exemple pour vérifier si des mises à jour ont bien été persistées en base).

Dans un tel scénario, vous utiliserez probablement un framework comme [Cypress](#) ou une bibliothèque telle que [puppeteer](#) afin de pouvoir naviguer à travers de multiples routes et vérifier les effets de bord non seulement au sein du navigateur, mais dans votre couche serveur également.

Recettes de test

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

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/test-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

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

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

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

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);
      }}
      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

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

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

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();

```

Pas trouvé votre bonheur ?

Si nous avons oublié de couvrir un scénario courant, n'hésitez pas à nous en faire part sur notre [outil de suivi de tickets](#) pour le site web de la documentation.

react

[Go to TOC](#)

Aperçu des tests

- "community/testing.html"

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

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

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

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

- [Les recettes](#) : une compilation d'approches éprouvées pour l'écriture de tests visant des composants React.
- [Les environnements](#) : les considérations à examiner lorsque vous mettez en place un environnement de test de composants React.

Penser en React

- 'blog/2013/11/05/thinking-in-react.html'
- 'docs/thinking-in-react-zh-CN.html'

React est, à notre avis, la meilleure façon de créer des applis web vastes et performantes en JavaScript. Il a très bien tenu le coup pour nous, à Facebook et Instagram.

L'un des nombreux points forts de React, c'est la façon dont il vous fait penser aux applis pendant que vous les créez. Dans ce document, nous vous guiderons à travers l'élaboration avec React d'un tableau de données de produits proposant filtrage et recherche.

Commençons par une maquette

Imaginez que nous avons déjà une API JSON et une maquette de notre designer. La maquette ressemble à ceci :

<input type="text" value="Search..."/>
<input type="checkbox"/> Only show products in stock
Name Price
Sporting Goods
Football \$49.99
Baseball \$9.99
Basketball \$29.99
Electronics
iPod Touch \$99.99
iPhone 5 \$399.99
Nexus 7 \$199.99

Maquette

Notre API JSON renvoie des données qui ressemblent à ceci :

```
[ ];
```

Étape 1 : décomposer l'interface utilisateur en une hiérarchie de composants

Pour commencer, dessinez des cases autour de chaque composant (et sous-composant) sur la maquette, et attribuez un nom à chacune. Si vous travaillez avec un designer, il se peut qu'elle l'ait déjà fait, alors allez lui parler ! Ses noms de calques Photoshop pourraient devenir les noms de vos composants React !

Mais comment savoir quelles parties devraient disposer de leurs propres composants ? Utilisez les mêmes techniques que lorsque vous décidez de créer une nouvelle fonction ou un nouvel objet. L'une de ces techniques est le [principe de responsabilité unique](#), qui stipule qu'un composant ne devrait idéalement faire qu'une seule chose. S'il finit par grossir, il devrait être décomposé en sous-composants plus petits.

Comme vous affichez souvent un modèle de données JSON à un utilisateur, vous constaterez que si votre modèle a été correctement construit, votre interface utilisateur (et donc la structure de vos composants) correspondra aisément. En effet, l'interface utilisateur (UI) et les modèles de données tendent à adhérer à la même *architecture d'information*. Séparez votre UI en composants, où chaque composant représente juste un élément de votre modèle de données.

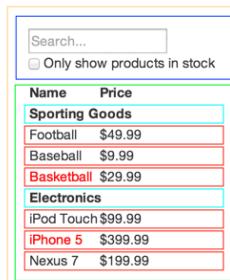


Schéma des composants

Vous pouvez voir que nous avons cinq composants dans notre petite appli. Nous avons mis en italiques les données que chaque composant représente.

1. **FilterableProductTable (orange)** : contient l'intégralité de l'exemple
2. **SearchBar (bleu)** : reçoit toutes les *données saisies par l'utilisateur*
3. **ProductTable (vert)** : affiche et filtre la *collection de données* en fonction des *données saisies par l'utilisateur*
4. **ProductCategoryRow (turquoise)** : affiche un titre pour chaque *catégorie*
5. **ProductRow (rouge)** : affiche une ligne pour chaque *produit*

Si vous regardez `ProductTable`, vous verrez que l'en-tête du tableau (contenant les titres "Name" et "Price") n'a pas son propre composant. C'est une question de préférence, et honnêtement les deux se valent. Dans cet exemple, nous l'avons laissé au sein de `ProductTable` car il fait partie de l'affichage de la *collection de données*, qui est de la responsabilité de `ProductTable`. Cependant, si cet en-tête devenait complexe (par exemple, si nous devions ajouter des options de tri), il deviendrait logique d'en faire son propre composant `ProductTableHeader`.

Maintenant que nous avons identifié les composants dans notre maquette, organisons-les en hiérarchie. Les composants qui apparaissent dans un autre composant sur la maquette doivent apparaître comme enfants dans cette hiérarchie :

- `FilterableProductTable`
 - `SearchBar`
 - `ProductTable`
 - `ProductCategoryRow`
 - `ProductRow`

Étape 2 : construire une version statique avec React

Maintenant que vous avez votre hiérarchie de composants, il est temps d'implémenter votre appli. La façon la plus simple consiste à construire une version qui prend votre modèle de données et affiche une UI inerte. Il est préférable de découpler ces processus, car la construction d'une version statique nécessite beaucoup de code et aucune réflexion, alors qu'ajouter de l'interactivité demande beaucoup de réflexion et peu de code. Nous verrons pourquoi.

Pour créer une version statique de votre appli qui affiche votre modèle de données, vous devrez créer des composants qui en réutilisent d'autres et transmettent les données au moyen des *props*. Les *props* sont un moyen de transmettre des données de parent à enfant. Si vous êtes à l'aise avec le concept d'*état local*, **n'utilisez pas d'état local du tout** pour construire cette version statique. L'état local est réservé à l'interactivité, c'est-à-dire aux données qui évoluent dans le temps. Comme il s'agit d'une version statique de l'appli, vous n'en avez pas besoin.

Vous pouvez construire l'appli en partant de l'extérieur ou de l'intérieur. En d'autres termes, vous pouvez aussi bien commencer par construire les composants les plus hauts dans la hiérarchie (dans notre cas, `FilterableProductTable`), que par ceux les plus bas (`ProductRow`). Dans des exemples plus simples, il est généralement plus facile de partir de l'extérieur, et sur des projets plus importants, il est plus facile de partir de l'intérieur et d'écrire les tests au fil de la construction.¹⁹⁷

Afin de construire correctement votre appli, vous devez d'abord penser à l'état modifiable minimal dont votre appli a besoin. La règle est simple : *ne vous répétez pas* (*Don't Repeat Yourself, aussi désigné par l'acronyme DRY, NdT*). Déterminez la représentation la plus minimale possible de l'état dont votre appli a besoin, et calculez le reste à la demande. Par exemple, si vous construisez une liste de tâches, gardez un tableau des tâches sous la main ; pas besoin d'une variable d'état pour le compteur. Au lieu de ça, quand vous voulez afficher le nombre de tâches, prenez la longueur du tableau de tâches.

Pensez à toutes les données de notre application. On a :

- La liste des produits
- Le texte de recherche saisi par l'utilisateur
- La valeur de la case à cocher
- La liste filtrée des produits

Passons-les en revue pour déterminer lesquelles constituent notre état. Posez-vous ces trois questions pour chaque donnée :

1. Est-elle passée depuis un parent via les props ? Si oui, ce n'est probablement pas de l'état.
2. Est-elle figée dans le temps ? Si oui, ce n'est probablement pas de l'état.
3. Pouvez-vous la calculer en vous basant sur le reste de l'état ou les props de votre composant ? Si oui, ce n'est pas de l'état.

La liste des produits est passée via les props, ce n'est donc pas de l'état. Le texte de recherche et la case à cocher semblent être de l'état puisqu'ils changent avec le temps et ne peuvent être calculés à partir d'autre chose. Enfin, la liste filtrée des produits ne constitue pas de l'état puisqu'elle peut être calculée en combinant la liste originale des produits avec le texte de recherche et la valeur de la case à cocher.

Au final, notre état contient :

- Le texte de recherche saisi par l'utilisateur
- La valeur de la case à cocher

- Trouvez leur plus proche ancêtre commun (un composant unique, au-dessus de tous les composants qui ont besoin de cette partie de l'état dans la hiérarchie).
- L'ancêtre commun ou un autre composant situé plus haut dans la hiérarchie devrait posséder cette portion d'état.
- Si vous ne trouvez pas de composant logique pour posséder cette partie de l'état, créez-en un exprès pour ça, et ajoutez-le quelque part dans la hiérarchie au-dessus de l'ancêtre commun.

Utilisons cette stratégie pour notre application :

- `ProductTable` doit filtrer la liste des produits en fonction de l'état et `SearchBar` doit afficher l'état du texte de recherche et de la case à cocher.
- Leur ancêtre commun est `FilterableProductTable`.
- Conceptuellement, il est logique que le texte du filtre et la valeur de la case à cocher soient dans `FilterableProductTable`

Parfait, nous avons donc décidé que `FilterableProductTable` possèdera notre état. Tout d'abord, ajoutez une propriété d'instance `this.state = {filterText: '', inStockOnly: false}` dans le `constructor` de `FilterableProductTable` pour refléter l'état initial de votre application. Ensuite, passez `filterText` et `inStockOnly` à `ProductTable` et `SearchBar` via leurs props. Enfin, utilisez ces props pour filtrer les lignes dans `ProductTable` et définir les valeurs des champs du formulaire dans `SearchBar`.

Vous pouvez commencer à voir comment votre application se comportera : définissez `filterText` à "ball" et rafraîchissez votre appli. Vous verrez que le tableau de données est correctement mis à jour.

Et c'est tout

Avec un peu de chance, vous avez maintenant une idée de la façon de penser la construction de vos composants et applis en React. Bien que ça demande peut-être un peu plus de code que vous n'en avez l'habitude, souvenez-vous que le code est lu beaucoup plus souvent qu'il n'est écrit, et que ce type de code, modulaire et explicite, est moins difficile à lire. Plus vous écrirez de composants, plus vous apprécierez cette clarté et cette modularité, et avec la réutilisation du code, le nombre de vos lignes de code commencera à diminuer.



Validation de types avec 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

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

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

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`.

Composants non-contrôlés

Dans la plupart des cas, pour implémenter des formulaires, nous recommandons d'utiliser des [composants contrôlés](#). Dans un composant contrôlé, les données du formulaire sont gérées par le composant React. L'alternative est le composant non-contrôlé, où les données sont gérées par le DOM.

Au lieu d'écrire un gestionnaire d'événements pour chaque mise à jour de l'état d'un composant non-contrôlé, vous pouvez [utiliser une ref](#).

Par exemple, ce code accepte une saisie de nom dans un composant non-contrôlé :

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('Un nom a été envoyé : ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Nom :
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Envoyer" />
      </form>
    );
  }
}
```

[Essayer sur CodePen](#)

Puisqu'un composant non-contrôlé garde la source de vérité dans le DOM, il est parfois plus simple d'intégrer du code React à base de composants non-contrôlés avec du code non-React. Le code peut également être légèrement plus concis si vous souhaitez pondre un truc vite fait. Autrement, vous devriez utiliser les composants contrôlés.

Si dans une situation donnée il n'est pas toujours évident de savoir quel type de composant utiliser, [cet article sur les champs contrôlés ou non-contrôlés](#) vous sera peut-être utile.

Valeurs par défaut

Dans le cycle de vie des rendus React, l'attribut `value` des éléments du formulaire va écraser la valeur du DOM. Avec un composant non-contrôlé, vous souhaiterez plus souvent spécifier la valeur initiale et laisser les mises à jours suivantes non-contrôlées. Dans ces cas-là, vous pouvez spécifier un attribut `defaultValue` plutôt que `value`.

```

render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Nom :
        <input
          defaultValue="Thierry"
          type="text"
          ref={this.input} />
      </label>
      <input type="submit" value="Envoyer" />
    </form>
  );
}

```

Dans la même série, `<input type="checkbox">` et `<input type="radio">` ont un attribut `defaultChecked`, et `<select>` et `<textarea>` ont aussi `defaultValue`.

La balise `input type="file"` {#the-file-input-tag}

En HTML, un `<input type="file">` permet à l'utilisateur de sélectionner un ou plusieurs fichiers depuis son appareil, et de les téléverser vers un serveur ou de les manipuler en JavaScript grâce à l'[API File](#).

```
<input type="file" />
```

`<input type="file" />` est toujours un composant non-contrôlé en React, puisque sa valeur ne peut être définie que par l'utilisateur et non programmatiquement.

Vous devriez utiliser l'[API File](#) pour interagir avec les fichiers. L'exemple suivant montre comment créer une [ref sur le noeud DOM](#) pour accéder aux fichiers lors de l'envoi du formulaire :

embed:[uncontrolled-components/input-type-file.js](#)

[Essayer sur CodePen](#)

[Go to TOC](#)

Web Components

React et les Web Components sont conçus pour résoudre des problèmes différents. Les Web Components offrent une encapsulation forte pour des composants réutilisables, tandis que React fournit une bibliothèque déclarative qui permet au DOM de rester synchronisé avec vos données. Les deux objectifs sont complémentaires. En tant que développeur·se, vous êtes libre d'utiliser React dans vos Web Components, ou bien d'utiliser des Web Components dans React, ou encore les deux à la fois.

La plupart des utilisateurs de React n'utilisent pas les Web Components, mais vous voudrez peut-être le faire, en particulier si vous utilisez des composants tiers d'interface utilisateur (UI) écrits à l'aide de Web Components.

Utiliser des Web Components dans React

```
class HelloMessage extends React.Component {
  render() {
    return <div>Bonjour <x-search>{this.props.name}</x-search> !</div>;
  }
}
```

Remarque

Les Web Components exposent souvent une API impérative. Par exemple, un Web Component `video` peut exposer les fonctions `play()` et `pause()`. Pour accéder à l'API impérative d'un Web Component, vous devez utiliser une référence pour interagir directement avec le nœud du DOM. Si vous utilisez des Web Components tiers, la meilleure solution consiste à écrire un composant React qui se comporte comme un enrobage pour votre Web Component.

Les événements émis par un Web Component peuvent ne pas se propager correctement à travers un arbre de composants de React. Vous devrez attacher manuellement les gestionnaires d'événements afin de gérer ces événements au sein de vos composants React.

Une source habituelle de confusion réside dans l'utilisation par les Web Components de `class` au lieu de `className`.

```
function BrickFlipbox() {
  return (
    <brick-flipbox class="demo">
      <div>Face</div>
      <div>Arrière</div>
    </brick-flipbox>
  );
}
```

Utiliser React dans vos Web Components

```
class XSearch extends HTMLElement {
  connectedCallback() {
    const mountPoint = document.createElement('span');
    this.attachShadow({ mode: 'open' }).appendChild(mountPoint);

    const name = this.getAttribute('name');
    const url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
    ReactDOM.render(<a href={url}>{name}</a>, mountPoint);
  }
}
customElements.define('x-search', XSearch);
```

Remarque

Ce code **ne fonctionnera pas** si vous transformez vos classes avec Babel. Consultez la discussion à ce sujet sur [ce ticket](#). Intégrer l'adaptateur `custom-elements-es5-adapter` préalablement au chargement de vos Web Components afin de résoudre ce problème.

Colophon

This book is created by using the following sources:

- React - Français
- GitHub source: [reactjs/fr.reactjs.org/content](https://reactjs.fr.reactjs.org/content)
- Created: 2022-11-28
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>