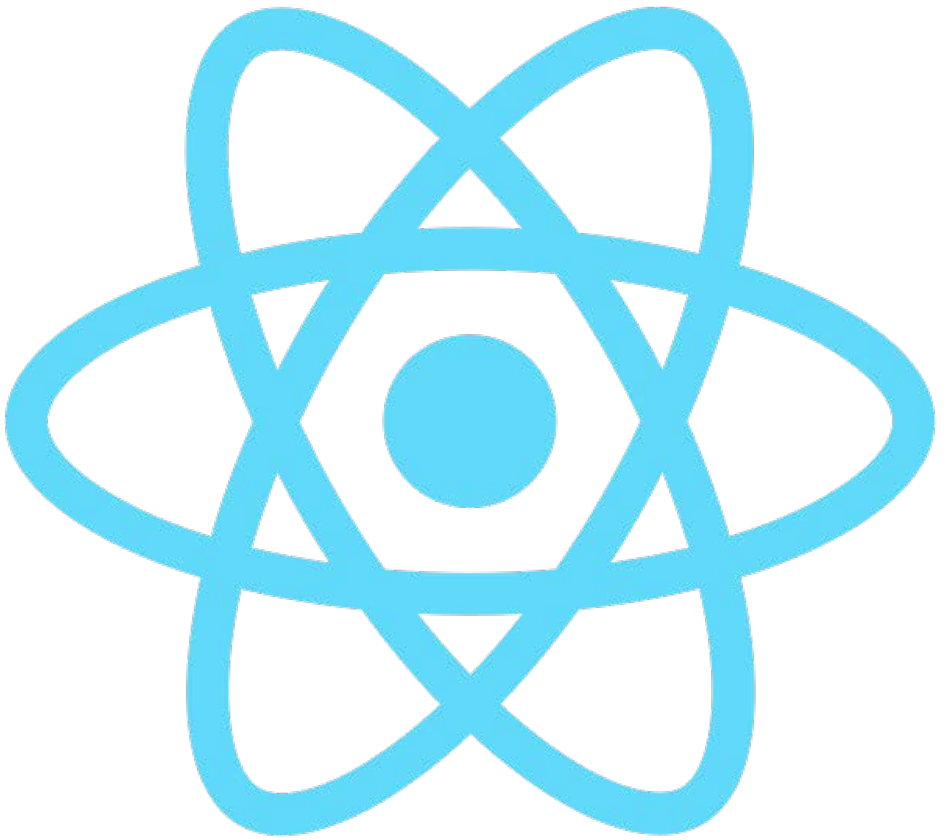
REACT JS

SUPPORT DE FORMATION "REALISEZ UNE APPLICATION WEB AVEC REACT.JS"



Réalisez une application web avec React.js

React.js est devenu une référence incontournable pour le développement d'expériences utilisateurs riches dans le navigateur web, y compris sur mobiles.

Ce cours vise à donner de solides bases sur React.js en explorant l'ensemble de ses concepts et possibilités, pour faciliter ensuite l'exploration du très vaste écosystème qui gravite autour.

Nous allons commencer par découvrir les concepts-clés de React.js et par mettre en place un environnement de travail performant. Pas à pas, nous

explorerons les fondamentaux du framework avant d'en dégager les subtilités et la puissance.

Ce cours vise particulièrement à démonter les pièges classiques que rencontrent les débutants - et même certains confirmés - sur React.js, et à mettre en lumière les meilleures pratiques établies chaque fois que possible. C'est la raison pour laquelle un volet entier sera consacré à la mise en place de tests automatisés des composants React.js.

Objectifs pédagogiques :

- Être en mesure d'expliquer les concepts fondamentaux de React, et ce qui le différencie d'autres frameworks
- Mettre en place un projet avec Create React App (CRA)
- Créer des composants React complets avec la syntaxe JavaScript ES2015 et l'extension JSX
- Gérer des formulaires avec ou sans contrôle de saisie
- Tester ses composants React

Prérequis :

Ce cours nécessite de maîtriser les bases de la programmation avec JavaScript, ainsi que des concepts de la programmation orienté objet. Si ce n’est pas le cas, vous pouvez suivre ce cours !

Découvrez l'utilité et les concepts clés de React

Bienvenue ! Dans cette partie, nous allons découvrir React et mettre en place notre environnement de développement afin d'effectuer correctement les exercices et les démonstrations du cours.

À quoi sert un framework front-end ?

Et tout d’abord, qu’est-ce que c’est ? On appelle **framework front-end** tout ensemble de classes, fonctions et utilitaires qui nous **facilite la création d'applications** riches pour les navigateurs (et, de plus en plus, pour les mobiles).

Un tel framework vise à nous isoler - nous qui développons des applications web - des
différences techniques entre les navigateurs, et à nous **éviter de réinventer la roue** pour tous les besoins classiques de nos applications : **gestion de l'interface utilisateur**, des événements, du DOM, des formulaires, de l'évolution dans le temps des données manipulées par l'interface, etc.

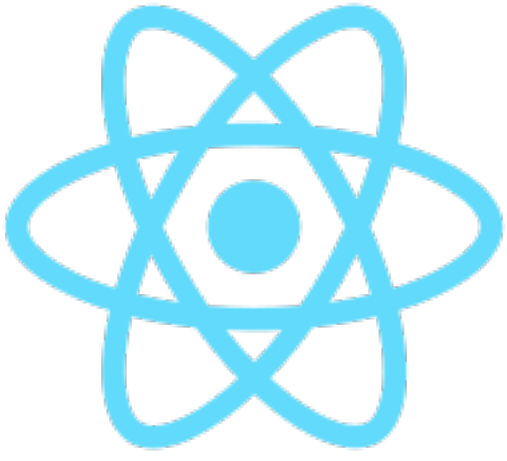
Dans cet « espace » des frameworks front-end (ou simplement « frameworks front »), on trouve pêle-mêle Angular, Ember, React, Vue.js et bien d'autres (par exemple Marko, Mavo, Backbone,...).

React

React est sans doute le framework front-end le plus populaire aujourd'hui, même s'il préfère se positionner comme une **bibliothèque** (*library*) plutôt que comme un framework. En effet, React se concentre sur le cœur du problème : la gestion de l'interface utilisateur. Les autres briques applicatives, comme le routage côté client, le stockage des données, etc. sont laissées aux innombrables solutions complémentaires de son écosystème (par exemple, React-Router, Redux, Redux-Offline...).

Framework ou bibliothèque ? Il existe quelques tentatives de distinguer formellement l'un de l'autre ; par exemple, une explication populaire dit qu'un framework appelle notre propre code, là où c'est notre code qui appelle celui d'une bibliothèque. Mais en pratique, il arrive fréquemment que les deux se mêlent, comme dans le cas de React.

Pour moi, la frontière est plus subtile, mais j'aurais tendance à dire que oui, React reste un framework, même s'il est beaucoup plus léger que des frameworks « qui veulent tout faire », comme Angular, Aurelia ou Ember...



Le logo de React

React est un projet **open-source** , désormais distribué sous la très populaire licence MIT et piloté par Facebook, dont tous les produits web et mobiles reposent sur cette technologie. Autour d'une équipe principale composée d'une vingtaine de personnes, gravitent **plus de mille contributeurs** occasionnels. En octobre 2017, **plus de 20 000 modules** sur npm (le principal référentiel de modules de développement, surtout orienté JavaScript, qui contient quelques 650 000 modules au total) avaient un rapport à l'écosystème React.

La plupart des « gros sites » connus du web ont migré leur interface web - et souvent leurs applications mobiles - vers React et React Native (son pendant pour la création d'applications natives iOS et Android). **Airbnb**, Algolia, Atlassian, Bit.ly, Campaign Monitor, Citymapper, CloudFlare, Codecademy, Coursera, **Dailymotion**, Discord, **Dropbox**, **Facebook** évidemment, Housing.com, IMDb, **Instagram**, KISSmetrics, **Netflix**, **Paypal**, Tesla, **Twitter**, Walmart, **Wordpress**, **Yahoo!** et des centaines d'autres en sont quelques exemples

L'édition 2017 du JavaScript Developer Survey montre bien cette domination : plus de la moitié des répondants ont déjà utilisé React et sont prêts à recommencer, tandis qu'un bon quart souhaite d'y mettre ! C'est davantage que tous les autres principaux frameworks réunis…

Une approche basée composants

React a remis sur le devant de la scène l'approche basée **composants** qui

avait déjà dominé le monde du développement d'applications riches de bureau (sur Windows principalement), au travers de solutions comme Delphi à l'époque.

Le web avait été conçu initialement pour des **documents**, et non pour des applications. Cela se ressentait dans le fonctionnement de HTML, de CSS et du DOM, de sorte qu'à force d'ajouter des rustines et de tenter d'y faire rentrer au chausse-pied une gestion applicative, on obtenait finalement des applications dont le code source ressemblait à un immense **plat de spaghetti sauce jQuery** : incompréhensibles, très ardues à maintenir et à faire évoluer, bourrées de bugs et d'effets de bord. Cet excellent didacticiel comparé de 2015 montre bien ce phénomène (même s'il souffre d'un compte JSBin débordé pour ses démos…)

React a fait **voler en éclat les dogmes établis du développement web**, tels que la séparation stricte entre la structure (HTML), l'aspect (CSS) et le comportement (JS), classiquement écrits dans des fichiers bien distincts, pour revenir à la notion fondamentale de composant autonome, cohérent et complet.

Notez que tous les « concurrents » de React - Angular, Ember, Vue.js ou autres - rejoignent cette philosophie. Leurs approches diffèrent essentiellement dans la syntaxe et les détails d'implémentation. Leurs écosystèmes, outillages et nombres de déploiements majeurs sont en revanche beaucoup plus réduits que pour React.

Encapsulation

Les composants React se comportent vis-à-vis du reste de l'application comme une **boîte noire**, avec une interface de programmation externe (une **API**) **clairement définie**. Ils contiennent en eux tout le nécessaire à leur bon fonctionnement : **la structure, les styles, le comportement**.

Le fait de rassembler dans un même fichier source ces trois volets, connectés par une **syntaxe concise et familière** (baptisée **JSX**, que nous verrons en détail dans la prochaine partie), facilite considérablement

le développement. En effet, pour comprendre les changements d'aspect et le comportement d'un composant, plus besoin de fouiller dans de multiples fichiers plus globaux en essayant de « connecter les points » de l'un à l'autre !

Notez que cette encapsulation facilite aussi l'écriture de **tests automatiques** pour les composants.

Composition

React nous encourage à structurer nos interfaces comme une **arborescence de composants**, et à rendre ces composants éminemment **réutilisables**. La majorité de nos composants sont eux-mêmes créés en **combinant d'autres composants** plus simples : c'est le principe fondamental de la **composition**, un outil primordial de structuration de code et d'application.

« Les données descendent, l'état remonte »

Lorsque l'on programme des comportements complexes dans une interface utilisateur, une des plus grosses sources de complication et de difficulté réside dans la gestion de **l'état applicatif** : ce sont les données manipulées par l'interface qui doivent évoluer au fil des manipulations en respectant une série de règles métier précises. Ainsi, certaines saisies sont invalides, d'autres sont inter-dépendantes, etc.

Réciproquement, lorsque les données évoluent, l'interface est censée réagir pour refléter le nouvel état applicatif. Par exemple, l'absence de données obligatoires *doit* interdire l'accès à certaines actions, des saisies invalides *doivent* provoquer un affichage d'erreur, une saisie a *besoin* d'être formatée au fil de l'eau, etc.

Afin d'éviter que tous les composants ne finissent par être étroitement connectés entre eux (appelé *couplage fort*, et constituant un gros problème de maintenance), React nous force à suivre un principe simple : *« les données descendent, l'état remonte »*.

Chaque composant est seul responsable des informations qu'il fournit à ses composants fils. Lorsque ceux-ci ont besoin de signaler une évolution de l'état applicatif, ils font alors remonter cette demande de composant parent en composant parent, jusqu'à atteindre un niveau suffisamment haut pour contenir tous les autres composants intéressés.

Les deux mécanismes techniques employés pour cela sont les **props** et **l'état local**. Nous verrons ces notions en détail dans la troisième partie de ce cours, « Créez des composants complets ».

Le DOM virtuel

React a inauguré la notion de DOM virtuel : React lui-même ne manipule pas directement le DOM du navigateur. Cela serait coûteux en performance et empêcherait de décliner cette approche sur d'autres supports tels que les applications mobiles natives, les terminaux textuels, les fichiers PDF, etc.

À la place, React nous fait décrire un **DOM virtuel**, absolument distinct du DOM des navigateurs. Au moment venu il **réconcilie** ce DOM virtuel avec la couche de rendu réelle (par exemple, le DOM du navigateur, ou, si on est côté serveur, la production du texte HTML à renvoyer côté client), en prenant soin de minimiser le nombre d'opérations nécessaires.

Ce système permet **d'excellentes performances**. Cependant, à mon sens, son véritable avantage est qu'il nous permet d'utiliser React dans de nombreux contextes, et pas seulement au sein du navigateur même. Le **pré-rendu côté serveur** et les **applications natives mobiles** sont les deux autres cas de figure les plus intéressants.

Dans le prochain chapitre, nous découvrirons *Create-React-App*, un outil de développement spécialement conçu pour mettre facilement le pied à l'étrier et pouvoir attaquer le développement avec React en un rien de temps.

Démarrez facilement avec Create-React-App

À quoi sert Create-React-App (« CRA ») ?

Create-React-App est un outil pour faciliter le développement d'applications web fondées sur React. L'idée est de générer automatiquement un squelette applicatif et de masquer la complexité potentielle de configuration des briques techniques associées : gestion de JavaScript moderne (ES2015+), bundling de notre application (avec Webpack), serveur de développement, génération de fichiers de production optimisés, etc.

Il est par ailleurs facile de mettre à jour automatiquement l'outil lui-même, et le jour où on en atteint les limites, « d'ouvrir la boîte » pour gérer soi-même chaque aspect technique. Au fil des versions, CRA a permis de gérer de plus en plus de besoins, comme en témoigne son guide utilisateur grandissant sans cesse.

Installations

CRA est écrit en Node et nécessite Node 6+. Node.js est un environnement d'exécution JavaScript installable partout, qui permet d'écrire n'importe quel type de programme en JavaScript : outil en ligne de commande, serveur, application desktop, internet des objets (IoT), et j'en passe. À l'heure où j'écris ces lignes, la version stable de Node est la 8.6.0. Assurez-vous d'avoir une version suffisamment récente.

Node

Node peut être installé de nombreuses façons. Commencez par ouvrir un terminal (sous Windows, si vous n'avez pas le Windows Subsystem for Linux installé, vous pouvez utiliser cette bonne vieille **Invite de Commandes**, `cmd.exe`) et vérifier si Node est déjà installé :

```
node --version
v8.6.0
```

Si vous obtenez plutôt un message d'erreur, ou un numéro de version antérieur à 6.x, installez la dernière version de Node.

Même si elles datent parfois un peu, les instructions de cet autre cours vous aideront à installer Node,

npm 5.3+

npm est le gestionnaire de modules de Node, utilisé pour de nombreux autres types de contenus. C'est à la fois un outil en ligne de commande (la commande `npm`) et un référentiel en ligne de modules, interrogé par cet outil. On y trouve plus de 650 000 modules couvrant à peu près tous les

besoins imaginables. La commande permet notamment d'installer et de mettre à jour les modules que l'on sélectionne pour nos projets.

Indépendamment de la version livrée par défaut avec Node, **vous pouvez toujours installer le dernier npm** si vous êtes sur une version de Node pas trop ancienne.

npm sort très régulièrement des mises à jour qui gagnent en performance et en fonctionnalités. Assurez-vous d'utiliser la version 5 - au minimum, l'idéal étant la version 5.3 ou au-delà - de npm. Pour cela, une fois Node installé, exécutez la commande suivante dans votre terminal :

```
npm install --global npm
```

Si vous utilisez un système Linux ou OSX et que vous obtenez un message d'erreur, celui-ci est probablement lié à un problème de droits. Vous pouvez régler rapidement le problème en préfixant cette commande par `sudo` . Une meilleure pratique est d'utiliser un gestionnaire de versions Node ou de configurer un préfixe d'installation dans votre configuration

Vérifiez ensuite :

```
npm --version
5.5.1
```

Terminal ? Un terminal est un programme qui propose, dans des fenêtres et onglets, des **lignes de commandes**, des programmes au sein desquels vous interagissez avec la machine en tapant des commandes textuelles plutôt qu'en cliquant sur des boutons et en tapant dans des champs de saisie.

Les lignes de commandes sont les premières interfaces visuelles de l'informatique, bien avant les interfaces graphiques de type Windows, OSX ou KDE. Cela ne signifie pourtant pas qu'elles sont moins puissantes : souvent, une saisie en ligne de commande ira plus vite que de nombreuses manipulations graphiques… voire sera la seule approche possible !

Windows ne propose que deux terminaux, pas très avancés au demeurant : Invite de Commande (*Command Prompt*), fourni par le programme `cmd.exe` , et PowerShell, fourni par `powershell.exe` . À partir de la version 10, vous pouvez installer le *Windows Subsystem for Linux* (WSL), et bénéficier ainsi de toutes les lignes de commandes Linux, notamment Bash et zsh.

Sur OSX, on a par défaut le programme Terminal, caché dans les Utilitaires, qui fait déjà très bien le boulot.

Sur Linux, on a pléthore de terminaux ; apprenez-en les bases !

Create-React-App

Enfin, terminons par l'installation de CRA. Le plus simple, et qui est recommandé, est de l'installer comme une commande globale, ensuite utilisable de n'importe où. Pour cela, ouvrez si besoin un terminal et tapez la commande ci-après (la première ligne) :

```
npm install --global create-react-app
…
+ create-react-app@1.4.1
added 107 packages in 19.638s
```

(vos chiffres, versions et durées pourront différer)

À nouveau, si vous constatez un problème de droits sur OSX ou Linux, relisez l'avertissement précédent.

Créer notre premier squelette d'application

Voilà ! Il est temps à présent de créer notre première application React !

Demandons à CRA de créer le squelette de notre application, qui s'appellera *Memory*. Ouvrez (si ce n'est déjà fait) un terminal, placez-vous si besoin dans un dossier de votre choix (commande `cd`), puis saisissez la commande `create-react-app memory` , comme ci-dessous :

```
create-react-app memory

Creating a new React app in /Users/tdd/perso/delicious-insights/ocr/memory.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...


…
✨  Done in 28.96s.

Success! Created memory at /Users/tdd/perso/delicious-insights/ocr/memory
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can’t go back!

We suggest that you begin by typing:

  cd memory
  yarn start

Happy hacking!
```

Comme vous pouvez le voir, l’affichage final est très sympathique, et il nous donne immédiatement quelques suggestions pour la suite.

Les fichiers produits

En examinant le contenu du dossier memory ainsi créé, nous voyons qu’en dehors du gros dossier *node_modules*, où toutes les dépendances ont été installées, on trouve déjà pas mal de choses :

```
.
├── README.md
├── package.json
```

```
├── public
|   ├── favicon.ico
|   ├── index.html
|   └── manifest.json
├── src
|   ├── App.css
|   ├── App.js
|   ├── App.test.js
|   ├── index.css
|   ├── index.js
|   ├── logo.svg
|   └── registerServiceWorker.js
└── yarn.lock
```

On y trouve le point d'entrée de l'application (`src/index.js`), un premier composant (`src/App.js`) avec ses styles à part et ses tests, ainsi qu'une page de support pour le tout (`public/index.html`). On y aperçoit également des éléments plus avancés qui sortent largement du cadre de ce cours (le *service worker*, le manifeste applicatif PWA...).

Seuls `src/index.js` et `public/index.html` sont exigés par CRA ; tout le reste est à notre gré. Par ailleurs, l'ensemble des fichiers sources (JS, CSS, etc.) que nous créerons devront être dans `src/` , sans quoi le Webpack utilisé en interne ne les verra pas.

Les commandes disponibles

Le message de fin de génération nous propose diverses commandes, dont `start` , `test` et `build` . On y trouve également la commande `yarn` , qui est une alternative à la commande officielle `npm` . Yarn est un client alternatif, mais ses avantages se sont considérablement réduits depuis la sortie de npm 5, qui a quant à elle d'autres forces uniques. Dans ce cours, nous resterons avec le npm officiel.

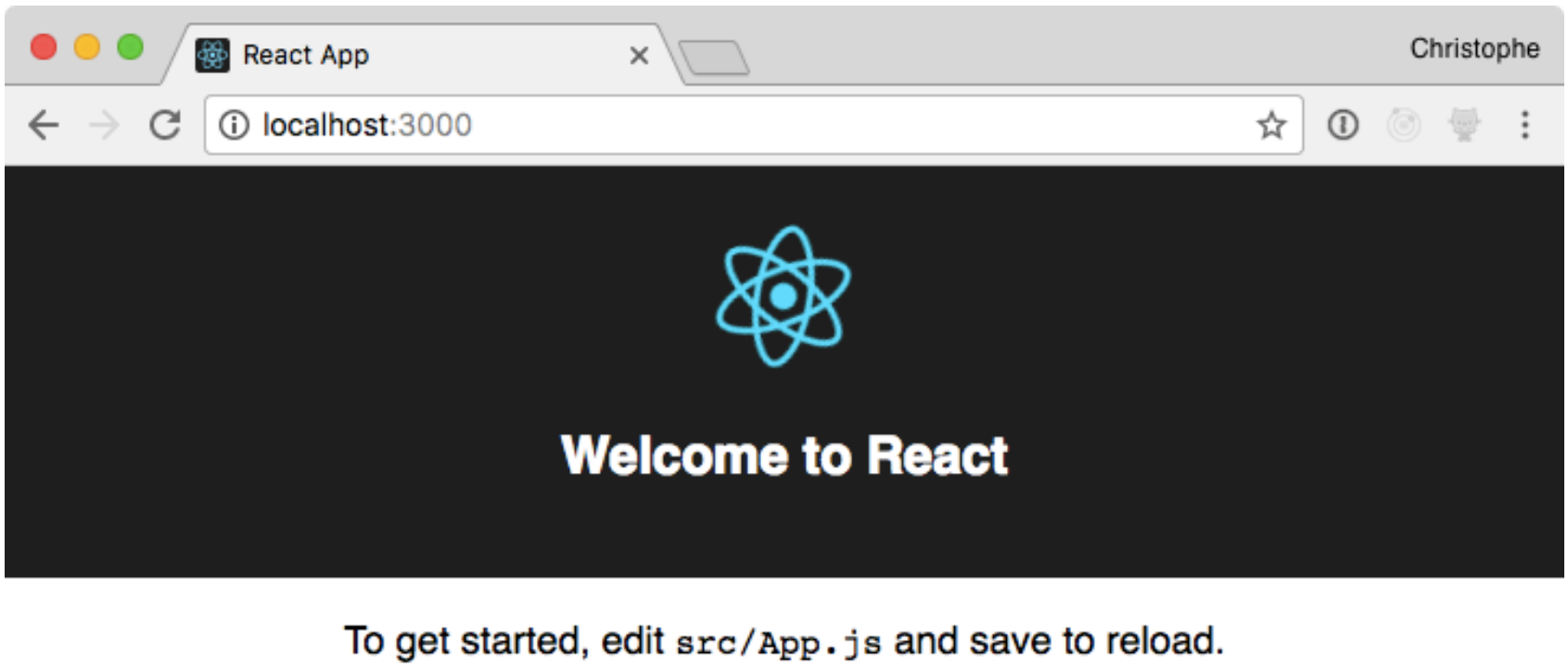
Voyons ce que donne notre squelette au démarrage. Pour cela, il faudra lancer la commande `npm start` au sein d'une ligne de commande, **depuis le répertoire dans lequel a été créé l'application**.

```
npm start
```

```
…

Compiled successfully!

You can now view memory in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://192.168.1.15:3000/

Note that the development build is not optimized.
To create a production build, use yarn build.
```

(Votre IP sera probablement différente de 192.168.1.15)

Notez que notre application de base a dû s'ouvrir automatiquement dans notre navigateur *(si ce n'est pas le cas, ouvrez un nouvel onglet dans votre navigateur et saisissez l'URL indiquée par la commande dans le terminal, normalement* `http://localhost:3000/` *)* :



Notre squelette applicatif ouvert dans le navigateur

Voilà qui est prometteur !

Le langage JavaScript ayant connu de récentes évolutions dont nous allons avoir besoin, nous allons dans le prochain chapitre faire un tour rapide des

nouveautés syntaxiques apportées par ES2015 ("ES6"). Nous verrons également les syntaxes plus récentes encore (voire même futures) prises en charge par CRA.

Modernisez votre JavaScript avec ES2015

L’univers React, comme ceux de la plupart des frameworks front-end, utilise désormais intensément des **avancées récentes de JavaScript**. Ces avancées sont apparues à partir de 2015, dans la version du langage appelée ES2015 (également connue sous le nom « ES6 »).

Peut-être n’avez-vous pas l’habitude de cette syntaxe : nous allons donc jeter un rapide coup d’œil sur les principales nouveautés que vous rencontrerez fréquemment à l’usage.

Classes

Les composants React peuvent être définis sous forme de fonctions ou de classes. ES2015 fournit du « sucre syntaxique » qui facilite considérablement la **définition de classes**, comparée à l'approche traditionnelle en JavaScript.

Vous n'y connaissez pas grand chose en Programmation Orientée Objet (POO) ? Vous pouvez faire un petit détour utile par ce joli cours pour remédier à ça. Pour le moment, sachez juste qu'en POO, on manipule des *objets* qui représentent des entités de notre programme (des composants, des documents, des utilisateurs…), et qu'une *classe* est en quelque sorte le plan de construction pour une catégorie d'objets (et l'usine qui les fabrique).

Cette syntaxe est comparable à celle trouvée dans de nombreux langages courants. Voici un exemple :

```
class Screen extends Component {

  constructor (props) {
    super(props)
    this.state = { loginState: 'logged-out' }
  }


  render () {
    // …
  }
}
```

Faites attention aux points suivants :

- Le mot-clé `class` permet de définir un **corps de classe**, dont les éléments (constructeur, méthodes, etc.) n'ont pas besoin d'être séparés par des virgules, contrairement à ce qui se passe dans les littéraux objets.

- Il est facile de spécialiser une classe par héritage à l'aide du mot-clé `extends`

- Le mot-clé `constructor` permet de définir le constructeur de la classe ; au sein du constructeur, `super(…)` permet d'appeler le constructeur hérité, ce qui est d'ailleurs **obligatoire** si la classe en spécialise une autre avec `extends` (et doit impérativement précéder toute manipulation de `this`).

Notez bien : si vous écrivez une classe qui en spécialise une autre avec `extends` , et que votre classe a son propre constructeur, celui-ci doit impérativement appeler le constructeur parent avec `super(…)` , et ce *avant* de manipuler `this` de quelque façon que ce soit.

C'est une obligation syntaxique dans la spec : votre code JavaScript ne devrait pas être valide sans cela. En pratique, la plupart des moteurs JS (hors Babel, qui justement tourne « sous le capot » dans Create React App et dans la plupart des projets) attendront l'invocation du `new` pour détecter le souci, et lever une `ReferenceError` plutôt qu'une `SyntaxError` . En tous les cas, ça ne marchera pas *in fine*.

- Les méthodes doivent être déclarées avec la nouvelle syntaxe de « méthodes concises », également disponible dans les littéraux objets : `nom (paramètres) { … }` , contrairement à ce que l'on faisait auparavant dans les littéraux objets, comme par exemple `nom: function (paramètres) { … }` .

Vous pouvez retrouver tous ces éléments dans la documentation.

Fonctions fléchées

Une syntaxe plus concise

Le premier avantage des fonctions fléchées est la concision de leur syntaxe.

Traditionnellement, une fonction se déclare avec le mot-clé `function` suivi de la signature puis du bloc de fonction, au sein duquel tout `return`

doit être explicite. Par exemple :

```
const adults = [], minors = []

people.forEach(function (person) {
  if (person.age >= 18) {
    adults.push(person)
  } else {
    minors.push(person)
  }
})


people.map(function (person) {
  return person.firstName
})
```

Les fonctions fléchées ne nécessitent pas de mot-clé déclaratif. Lorsqu'elles sont intégralement constituées d'une expression à renvoyer, les accolades du bloc de fonction peuvent être omises, ainsi que le mot-clé `return` . Entre la signature et le bloc ou l'expression, on trouve une *fat arrow* (un nom pas terrible, je vous l'accorde !), à savoir `=>` .

```
const adults = [], minors = []

people.forEach((person) => {
  if (person.age >= 18) {
    adults.push(person)
  } else {
    minors.push(person)
  }
})
```

```
people.map((person) => person.firstName)
```

C’est une syntaxe raccourcie très pratique pour les **prédicats** (fonctions qui renvoient vrai ou faux selon leurs arguments, et qui sont par exemple passées aux méthodes `filter`, `every` et `some`) et les **mappers** (fonctions de transformation telles que celles passées à `map`). Dans ce deuxième cas, elles seront par exemple très utiles pour « réduire le bruit » dans les méthodes `render` de nos composants React.

La question du `this`

Historiquement, JavaScript ne proposait qu’une manière de déclarer des fonctions : le mot-clé `function`. Les fonctions ainsi déclarées avaient un comportement spécifique à JavaScript, fondé non pas sur l’endroit de leur déclaration dans le code, mais sur **la syntaxe utilisée pour les appeler.** Cette syntaxe déterminait notamment, au sein de la fonction, la valeur de `this`.

Le fait de pouvoir appeler une fonction en contrôlant le rôle de `this` à l’intérieur a de nombreux avantages, mais comporte aussi un inconvénient : dans les fonctions de rappel (*callbacks*) notamment, on voulait souvent pouvoir continuer à utiliser le `this` en vigueur « à la ligne du dessus », comme pour n’importe quel identifiant ordinaire.

```
const name = 'Extérieur'


const obj = {
  name: 'Intérieur',
  runGreet () {
    // Ici, this.name est bien "Intérieur"
    setTimeout(function () {
      // Ici, this est soit l’objet global (mode laxiste de JS),
      // soit null (mode strict de JS)
```

```
    }, 0)
  }
}

obj.runGreet()
```

Les fonctions fléchées contournent ce problème en ne redéfinissant aucun identifiant lors de leur invocation (pas même `this`).

```
const name = 'Extérieur'

const obj = {
  name: 'Intérieur',
  runGreet () {
    // Ici, this.name est bien "Intérieur"
    setTimeout(() => {
      // Ici, this n'est pas redéfini par la fonction,
      // car c'est une fonction fléchée : comme n'importe
      // quel identifiant, il est donc recherché dans les
      // portées englobantes, et trouvé au niveau de
      // runGreet, c'est donc aussi "Intérieur".
    }, 0)
  }
}

obj.runGreet()
```

Cette capacité à ne pas interférer avec le `this` en vigueur les rendra inestimables pour les méthodes `render` de nos composants React… vous verrez !

Déstructuration

La déstructuration nous permet d'aller rapidement chercher **plusieurs propriétés** au sein d'un objet, ou **plusieurs cellules** au sein de n'importe quel objet itérable (comme un tableau) sans avoir à multiplier les déclarations ou affectations.

Par exemple, au lieu de faire ceci :

```
// À l'ancienne

const firstName = this.props.firstName

const lastName = this.props.lastName

const onClick = this.props.onClick
```

On peut simplement faire :

```
// Avec une déstructuration basée sur les noms

const { firstName, lastName, onClick } = this.props
```

Imaginons maintenant qu'on découpe un texte « prénom nom » en deux, et qu'on veuille affecter les parties à deux identifiants. Au lieu de faire :

```
// À l'ancienne

const names = fullName.split(' ')

const firstName = names[0]

const lastName = names[1]
```

On peut faire :

```
// Avec une déstructuration basée sur les positions

const [firstName, lastName] = fullName.split(' ')
```

Sympathique, non ? Il est possible d'aller beaucoup plus loin, mais ces

exemples suffiront amplement à nos besoins pour ce cours.

Modules natifs, imports et exports

Nous découperons notre application en **modules**, qui sont autant de **fichiers sources séparés**. Pour cela, nous aurons recours à la syntaxe officielle des **ES Modules**. Nous rendrons visibles les parties de nos modules que nous souhaitons à l'aide d'`export` et nous irons chercher les parties qui nous intéressent dans d'autres modules à l'aide d' `import` .

Voici un exemple de deux modules, le second utilisant le premier (ce sont deux fichiers différents, rassemblés ci-dessous pour plus de concision) :

```
// Au sein du fichier textUtils.js


export function countWords (text) {
  return text.split(/\W+/u).filter(Boolean).length
}

export function normalizeSpacing (text) {
  return text.replace(/\s+/u, ' ').trim()
}

// Au sein d'un fichier main.js, dans le même répertoire :

import { countWords } from './textUtils'

console.log(countWords('Hello world, this is nice!'))
```

Il est également possible de déclarer un export « par défaut ». Dans ce cas nous pouvons l'importer sous le nom que l'on veut, sans avoir à recourir aux accolades :

```
// Dans le module exportateur, SuperComponent.js :


export default class SuperComponent {
  // …
}


// Dans le module importateur, dans le même répertoire :


import GreatComponent from './SuperComponent'
```

Désormais, vous avez tous les éléments pour écrire vos composants React de façon moderne !

Écrivez des fonctions pures

React permet de définir des composants à l'aide d'**une simple fonction**. Ce type de composant, appelé *« composant pur fonctionnel »*, doit idéalement toujours renvoyer la même chose pour les mêmes arguments, et ce sans effet de bord ni variation. On parle alors de *fonctions pures*. Cela le rend particulièrement **facile à tester**.

React encourage le recours à ce type de composant, autant que possible, plutôt qu'à ceux définis par des classes. Pour se faire une idée, on estime que pour la majorité des applications, environ 90% des composants seront réalisés de cette façon.

Avant de créer du code dans notre application à proprement

parler, apprenons à réaliser un composant sous forme de fonction. Celle-ci doit retourner un DOM virtuel React, à l'aide de `React.createElement(…)`, comme le montre l'exemple ci-dessous :

```
function CoolComponent() {
  return React.createElement('p', {}, 'Youpi So Cool !')
}
```

Le premier argument est le nom du composant. Dans notre exemple il s'agit d'une String et non d'une référence à une fonction ou classe JavaScript. React déduira donc qu'il s'agit d'un type d'élément fourni nativement par la plate-forme, à savoir le navigateur. Le second argument est lui une série d'attributs (par exemple `id`, `className` …). Enfin, le dernier est le contenu, ici un simple texte.

Pour afficher un DOM virtuel React dans une page web, on utilise `ReactDOM.render(…)`, comme vous pouvez le voir dans le fichier `src/index.js` du squelette créé dans la partie précédente avec Create React App. Ici, nous aurions donc :

```
ReactDOM.render(
  React.createElement(CoolComponent),
  document.getElementById('root')
)
```

Un aperçu de JSX

Utiliser manuellement `React.createElement(…)` devient toutefois très vite verbeux, aboutissant à du code source qui « noie le poisson ». Il devient alors délicat d'y repérer les données importantes. C'est pourquoi React est généralement utilisé avec la **syntaxe JSX**, une extension à JavaScript qui ressemble un peu à du XML au sein de JavaScript - mais je vous assure que c'est une bonne idée ! Pas comme autrefois !

Notre code deviendrait alors :

```
function CoolComponent() {
  return <p>Youpi So Cool !</p>
}

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

C'est déjà quand même beaucoup plus sympa !

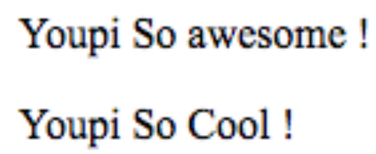
Premières props

On peut fournir à un composant des « attributs », appelés *props* en terminologie React. Ces props sont définies par un ensemble de clés/valeurs, définies dans un objet, qui est passé en argument à la fonction du composant. Cet objet est alors déstructuré automatiquement par la fonction.

```
function CoolComponent({ adjective = 'Cool' }) {
  return <p>Youpi So {adjective} !</p>
}

ReactDOM.render(
  <div>
    <CoolComponent adjective="awesome" />
    <CoolComponent />
  </div>,
  document.getElementById('root')
```

```
)
```

Dans cet exemple, une *expression JSX* apparait ligne 2. C'est une expression JavaScript quelconque, encadrée par des accolades. C’est par ce moyen qu’on met en place des contenus dynamiques dans du JSX. Ce code affiche :



Les deux paragraphes issus de <CoolComponent/>

Vous pouvez jouer interactivement avec ce code sur cette page.

Dans le prochain chapitre, nous verrons JSX en détail.

Décrivez un composant avec JSX

Nous avons vu qu’il est possible de décrire nos DOM virtuels React (« grappes ») avec la fonction `React.createElement(…)`. Nous allons voir dans chapitre qu'il est préférable d'utiliser la syntaxe JSX, créée spécifiquement pour React.

Pourquoi préférer JSX ?

Lorsqu’on se limite à un petit exemple, JSX ne semble pas être d'une énorme utilité…

```
// Avec JSX

<p>Oh le joli paragraphe</p>


// sans JSX

React.createElement(p, {}, 'Oh le joli paragraphe')
```

Ou encore :

```
// Avec JSX

<User first="John" last="Smith" />


// sans JSX

React.createElement(User, { first: 'John', last: 'Smith' })
```

Mais très vite, il devient difficile de se repérer. Jugez plutôt avec cette grappe pourtant assez simple :

```
<form method="post" action="/sessions" onSubmit={this.handleSubmit}>
  <p className="field">
    <label>
      E-mail
      <input
        type="email"
        name="email"
        required
        autoFocus
        value={this.state.email}
        onChange={this.handleFieldChange}
      />
    </label>
```

```
    </p>
    <p className="field">
      <label>
        Mot de passe
        <input
          type="password"
          name="password"
          required
          value={this.state.password}
          onChange={this.handleFieldChange}
        />
      </label>
    </p>
    <p>
      <button type="submit" value="Connexion" />
    </p>
  </form>
```

Remarquez comme il est facile de repérer la connexion aux traitements associés (`handleSubmit` , `handleFieldChange`), la source des données (`this.state.email`, `this.state.password`), ainsi que la structure générale du composant.

Voyons maintenant l'équivalent sans JSX :

```
React.createElement(
  'form',
  { method: 'post', action: '/sessions', onSubmit: this.handleSubmit },
  React.createElement(
    'p',
```

```
    { className: 'field' },
    React.createElement(
      'label',
      null,
      'E-mail',
      React.createElement('input', {
        type: 'email',
        name: 'email',
        required: true,
        autoFocus: true,
        value: this.state.email,
        onChange: this.handleFieldChange,
      })
    )
  ),
  React.createElement(
    'p',
    { className: 'field' },
    React.createElement(
      'label',
      null,
      'Mot de passe',
      React.createElement('input', {
        type: 'password',
        name: 'password',
        required: true,
        value: this.state.password,
        onChange: this.handleFieldChange,
      })
```

```
    )
  ),
  React.createElement(
    'p',
    null,
    React.createElement('button', { type: 'submit', value: 'Connexion' })
  )
)
```

Voyez comme ces mêmes informations sont noyées dans la masse de code, alors même que notre grappe reste modeste. Imaginez ce que ça donne pour un composant un tant soit peu complexe.

JSX, c'est bien plus que du simple confort : c'est une barrière en moins pour la compréhension rapide du fonctionnement de nos composants lorsqu'on lit le code. C'est de la maintenabilité - et de la productivité ! – en plus.

Ni du HTML, ni du XML

Dans la mesure où JSX est un langage à balises, il est tentant pour les développeurs de conserver tous leurs réflexes et modèles mentaux hérités de HTML ou d'XML. En pratique, si des similitudes existent, JSX présente aussi des différences importantes.

Parmi les points communs avec XML, on citera notamment :

- La sensibilité à la casse : les majuscules et minuscules ne sont pas interchangeables, `<CoolComponent />` n'a pas le même sens que `<coolComponent />` .
- L'exigence de fermeture des éléments :
    1. même pour les éléments « seuls », comme par exemple `<input />` (et non pas `<input>`),

2. dans le bon ordre (on ferme dans l'ordre inverse de la fermeture) : `<p><span>Bien</span></p>` mais pas `<p><span>Pas bien</p></span>` .

En revanche, pour le reste, il va falloir ajuster ses habitudes…

Valeurs des props

En JSX, on ne parle pas d'attributs (comme en XML ou HTML), mais de *props*. Ce terme revient dans toute la documentation et l'écosystème React.

String vs. tout le reste

En pratique, une prop peut avoir n'importe quelle valeur possible en JavaScript, mais syntaxiquement, dans JSX, on n'a en gros que deux possibilités : un littéral `String` , matérialisé par les double-quotes `""` ou une expression JSX, définie entre accolades`{}`.

```
<input
  type="email"
  name="email"
  maxlength={42}
  readonly={false}
  onChange={this.handleFieldChange}
  value={this.state.value}
/>
```

Ici, les *props* `type` et `name` ont des valeurs de type `String` . La syntaxe « façon HTML » s'applique donc. En XML ou en HTML il serait possible d'écrire, par exemple, `maxlength=42` (ou `maxlength="42"`). En JSX, il est par contre obligatoire d'utiliser une expression JSX (comme `maxlength={42}` , pour notre exemple) pour toute valeur qui n'est pas de type `String` .

En revanche, le type réel de l'expression JSX est préservé : la *prop* que l'on

récupèrera dans le composant sera bien, par exemple, un nombre, un booléen, un tableau, un autre composant, etc.

Raccourci pour true

JSX garde un raccourci syntaxique classique en HTML, pour les *props* booléennes dont la valeur est `true`. Au lieu d'écrire explicitement la valeur dans une expression JSX, on peut se contenter du nom de la *prop*, comme ceci :

```
<input type="email" name="email" autoFocus required />
```

Cette syntaxe courte est d'ailleurs recommandée.

Mots réservés

Dans la mesure où JSX produit, au final, du code JavaScript, et que celui-ci doit pouvoir s'exécuter dans un environnement ES5 (par exemple Internet Explorer 9+ ou d'anciennes versions de Node.js) et non ES2015+ (navigateurs modernes, versions de Node.js récentes, etc.), il n'est pas possible d'utiliser des mots-clés de JavaScript comme noms de *props*.

On retrouve donc les mêmes ajustements que dans les interfaces du DOM, qui reviennent essentiellement à écrire `className` au lieu de `class` et `htmlFor` au lieu de `for`.

Commentaires

Contrairement à HTML et XML, JSX n'a pas de syntaxe dédiée pour les commentaires (par exemple `<!-- … -->`) : si vous souhaitez mettre des commentaires au sein d'une grappe JSX, il faut le faire au sein d'une expression. Ça peut sembler parfois un peu chargé :

```
<form method="post" action="/sessions" onSubmit={this.handleSubmit}>
  {/* La classe 'field' assure l'espacement vertical convenable */}
  <p className="field">
```

```
    <label>
      E-mail
      <input type="email" name="email" required autoFocus
        value={this.state.email}
        {/*
        Avec les champs contrôlés, il est indispensable de fournir `onChang
        pour éviter que le champ soit fourni en lecture seule au niveau du
        DOM.
        */}
        onChange={this.handleFieldChange}
      />
    </label>
  </p>
  <p><button type="submit" value="Connexion" /></p>
</form>
```

L’importance de la casse

On l’a dit, JSX est sensible à la casse (Majuscules / minuscules), pour ses noms d’éléments comme pour ses noms de *props* (par exemple, `autoFocus` n’est pas la même prop que `autofocus`, qui d'ailleurs n'existe pas).

Pour les noms d’éléments, c’est encore plus important :

- dans une grappe JSX, si un élément **démarre par une minuscule**, le moteur considère qu’il s’agit d’un **élément natif** fourni par la plate-forme (le navigateur, par exemple),
- alors que si l’élément **démarre par une majuscule**, on estime qu’il s’agit d’un **composant React**.

Le code JavaScript obtenu ne sera donc pas le même :

```
[
  <CoolComponent/>,
  <coolComponent/>,
]
// donne :
[
  React.createElement(CoolComponent, null),
  React.createElement('coolComponent', null),
]
```

Notez que dans le second cas, on produit juste une `String` qui donne le nom de la balise supposée native, alors que dans le premier cas, on référence bien le composant React supposé (classe ou fonction).

Mise en application

Pour pratiquer un peu JSX, nous allons ajuster notre application.

Récupérer la base de travail

Nous allons commencer par supprimer certains fichiers inutiles pour le moment dans `src/` : `App.test.js` , `index.css` et `logo.svg` . Ensuite, nous reprendrons des squelettes de travail à partir du dépôt public pour ce projet, disponible sur GitHub, afin de pouvoir nous concentrer sur la partie React.

1. Allez sur le dépôt GitHub, à l'étiquette **debut-jsx** (cliquez sur le lien)
2. Récupérez les feuilles de style `App.css` , `Card.css` et `GuessCount.css` , ainsi que les squelettes de `Card.js` et `GuessCount.js`. Puis posez-les dans votre dossier `src/`.
3. Vous pouvez alors soit apurer votre `App.js` pour être identique à celui de départ sur GitHub, soit y récupérer directement `App.js`

comme base de travail.

4. Vous pouvez aussi en profiter pour changer le `<title>` de `public/index.html` de « React App » vers le plus représentatif « React Memory ».

Écrire le composant Carte

Nous faisons un jeu de *Memory*. Le composant de base qui sera affiché est donc la carte. Elle devra être soit visible, soit présentée « de dos », masquée. Partons du principe que notre composant `<Card />` recevra deux *props* : `card`, qui sera le symbole à afficher ; et `feedback`, qui indiquera l'état visuel de la carte : masquée ou visible. Voici comment ajuster la fonction `Card` :

```
const Card = ({ card, feedback }) => (
  <div className={`card ${feedback}`}>
    <span className="symbol">
      {feedback === 'hidden' ? HIDDEN_SYMBOL : card}
    </span>
  </div>
)
```

Remarquez quelques points :

- On déstructure directement les *props* passées en arguments (un gros objet de *props*).
- Étant donné que la fonction renvoie directement une grappe de DOM virtuel, sans calcul préalable, on se dispense des accolades de bloc et du `return`, pour renvoyer directement l'expression : on trouve donc plutôt des parenthèses autour du JSX.
- La syntaxe de valeur textuelle pour une *prop* ne permet pas l'interpolation (c'est-à-dire l'incrustation de contenu dynamique dans

le texte). On a donc recours pour le `<div>` principal à une expression JSX (entre accolades) qui, elle, peut utiliser la syntaxe des template strings d'ES2015, entre backquotes (`` ` ``), pour incruster dynamiquement la valeur de `feedback` .

Écrire le composant Compteur de tentatives

Le composant `<GuessCount />` est très simple : c'est un compteur de tentatives. Au fil de la partie, on l'affichera avant le plateau de cartes. Il attend juste une *prop* nommée `guesses` . Voici le code final :

```
const GuessCount = ({ guesses }) => <div className="guesses">{guesses}</div
```

Écrire le composant Application

Le composant principal reste `<App />` . Il est défini par une classe et non par une simple fonction, sans raison particulière pour le moment, mais ça viendra : nous en verrons les détails dans la prochaine partie de ce cours.

Pour l'instant, nous allons y poser le compteur et une série de six cartes « en dur » :

```
import Card from './Card'
import GuessCount from './GuessCount'


class App extends Component {
  render() {
    return (
      <div className="memory">
        <GuessCount guesses={0} />
        <Card card="😀" feedback="hidden" />
        <Card card="🎉" feedback="justMatched" />
        <Card card="💖" feedback="justMismatched" />
```

```
        <Card card="🎩" feedback="visible" />
        <Card card="🐶" feedback="hidden" />
        <Card card="🐱" feedback="justMatched" />
      </div>
    )
  }
}
```

Remarquez qu'il y a 4 feedbacks (ou états visuels) possibles et qui sont donc prévus :

- La carte est masquée (`hidden`).
- La carte fait partie de la tentative en cours, qui vient de réussir une paire (`justMatched`).
- La carte fait partie de la tentative en cours, qui vient de rater une paire (`justMismatched`).
- La carte appartient à une paire précédemment réussie (`visible`).

Réagissez aux événements

React facilite énormément la gestion des événements du DOM, notamment en fournissant une syntaxe pratique et concise tout en conservant des performances optimales.

Le beurre et l'argent du beurre

Lorsque nous écrivons le code de notre composant, le plus pratique consiste à pouvoir « connecter » nos gestionnaires d'événements directement depuis le code JSX, **afin d'avoir immédiatement une vision d'ensemble** de la partie interactive de notre composant.

Cependant, si nous adoptions cette approche directement au niveau du

DOM, en attachant nos gestionnaires à même les éléments concernés, nous rencontrerions rapidement des **soucis de performance** d'une part (trop de gestionnaires distincts, notamment sur de longues listes) et de **pérennité** d'autre part (dans le cas où une partie du contenu est rechargé, par exemple via Ajax, nos anciens gestionnaires sont perdus, il faut alors en attacher de nouveaux).

Une bonne pratique déjà ancienne puisque datant d'environ 2005 promeut la **délégation d'événements** : on attache nos gestionnaires le plus haut possible dans notre grappe applicative et on tire parti de la **propagation** des événements à travers le DOM pour reconnecter gestionnaires et éléments ciblés. L'ennui avec cette approche, c'est que les balises qui décrivent notre composant **n'indiquent plus ni quels événements sont gérés**, ni par qui : il faut alors fouiller ailleurs pour le découvrir.

En outre, ce recours à la propagation suppose que tous les événements qui nous intéressent respectent ce principe. Hélas ce n'est pas nativement le cas d'événements critiques tels que `submit` , `focus` ou `change` , par exemple.

React nous permet de déclarer nos gestionnaires à même le code JSX de notre composant, tout en optimisant en interne sa gestion événementielle, avec au plus un gestionnaire réel par type d'événement dans une grappe applicative React. On conserve la **lisibilité immédiate** du côté « déclaration de gestionnaire à la volée » sans sacrifier pour autant la performance. Et pour nous le proposer, et ce quel que soit le type d'événement souhaité, il simule la propagation lorsque celle-ci n'est pas native.

Exemples

Commençons par un composant tout simple :

```
const Greeter = ({ whom }) => (

  <button onClick={() => console.log(`Bonjour ${whom} !`)}>

    Vas-y, clique !
```

```
  </button>
)

ReactDOM.render(<Greeter whom="Roberto" />, document.getElementById('root')
```

Et voilà ! Cliquer sur le bouton obtenu affichera un « Bonjour Roberto ! » dans la console du navigateur.

Imaginons à présent que nous écrivons une gestion de logs (fichiers journaux), avec une liste de lignes à afficher, chaque ligne étant représentée par un composant. Nos lignes peuvent se déplier pour afficher davantage d'informations, ou au contraire se replier.

```
class LogEntry extends Component {
  // …
  render() {
    const className = `log entry ${this.isOpen() ? 'open' : 'closed'}`
    return (
      <li className={className} onClick={this.toggle}>
        …
      </li>
    )
  }
}
```

Notez comme il est pratique de voir, à même le JSX, qu'on va réagir aux clics sur la ligne… et que ce sera avec la méthode `toggle(…)` du composant !

Si nous faisions ça à même le DOM, ou à l'aide d'une technologie moins bien ficelée que React, nous aurions un gestionnaire d'événement DOM enregistré par ligne, ce qui entraînerait rapidement un problème de

réactivité pour *tous* les événements de la page ! Ici il n'en est rien : en pratique, React ne définit qu'un seul gestionnaire d'événement pour tous les clics, à la racine de la grappe applicative (react root).

Mise en application

Il est clair que dans notre jeu, l'application devra réagir lorsque l'on clique sur une carte. Contentons-nous pour le moment de loguer dans la console le symbole de la carte cliquée.

Commençons par une méthode métier sur le composant applicatif :

```
handleCardClick(card) {
  console.log(card, 'clicked')
}
```

Supposons ensuite un événement `onClick` sur les composants `<Card />` , et connectons les cartes présentées au gestionnaire :

```
<Card card="😀" feedback="hidden" onClick={this.handleCardClick} />
<Card card="🎉" feedback="justMatched" onClick={this.handleCardClick} />
<Card card="💖" feedback="justMismatched" onClick={this.handleCardClick} />
<Card card="🎩" feedback="visible" onClick={this.handleCardClick} />
<Card card="🐶" feedback="hidden" onClick={this.handleCardClick} />
<Card card="🐱" feedback="justMatched" onClick={this.handleCardClick} />
```

(Vivement qu'on apprenne à faire des listes !)

Il ne nous reste qu'à implémenter la *prop* `onClick` dans le composant `<Card />` :

```
const Card = ({ card, feedback, onClick }) => (

  <div className={`card ${feedback}`} onClick={() => onClick(card)}>

  // …
```

Tentez à présent de cliquer sur les cartes, tout en observant la console du navigateur !

Des événements garantis conformes W3C

En pratique, les objets événements que nous recevons en argument de nos gestionnaires ne sont pas les événements natifs du navigateur. Ceux-ci ont parfois des divergences de contenu et de comportement d'un navigateur à l'autre, ne respectant pas intégralement les spécifications officielles du W3C sur les événements DOM.

Dans React, comme dans la plupart des bibliothèques (ex. jQuery) et des frameworks (ex. Vue, Ember…), les événements natifs sont enrobés dans un type spécifique, garanti conforme au W3C, pour nous isoler de ces divergences entre navigateurs. Encore un souci de moins pour l'écriture de notre code !

Contextualisez le contenu de vos composants

Jusqu'ici, le contenu de nos composants conservait la même structure d'une utilisation à l'autre. Mais naturellement, dans de véritables applications, certaines parties peuvent changer suivant le contexte. Comment retranscrire cela dans notre JSX ?

Des expressions, pas des instructions !

Il faut tout d'abord garder en mémoire que JSX produit une **expression** JavaScript et non une instruction. Par conséquent, certaines parties du langage qui constituent des instructions, comme un `if` , un `for` ou une

déclaration, n'y sont pas possibles.

L'astuce consiste à remplacer nos instructions classiques de branchement conditionnel par des expressions utilisant les opérateurs logiques (typiquement, `&&` , `||` et l'opérateur ternaire `? :`).

« Si… Alors… »

Pour retranscrire un contenu conditionnel simple (« si la condition X est remplie, alors utiliser la grappe JSX que voici »), on utilisera l'opérateur logique `&&` , avec la condition comme opérande de gauche, et la grappe JSX comme opérande de droite.

Selon la sémantique de base de JavaScript, si la condition échoue, l'opérande de droite ne sera pas évaluée, et vaudra au global `false` . JSX ignorera alors ce `false` et rien n'apparaîtra dans le DOM navigateur à cet endroit.

```
<p>{42 > 43 && document.nonExistingMethod()}</p>
```

Dans le code ci-dessus, le paragraphe sera vide, l'expression s'arrêtant à son premier opérande, qui vaut `false` , et ce dernier étant automatiquement ignoré par JSX.

À l'inverse, si la condition est remplie, l'expression évaluera son opérande de droite, qui constituera alors la valeur finale : notre grappe JSX sera donc utilisée. Prenons pour exemple un objet `user` avec une propriété `admin` à `true` :

```
<p>{user.admin && <a href="/admin">Faire des trucs de VIP</a>}</p>
```

Dans ce cas, le paragraphe contiendra bien le lien. Remarquez que d'un point de vue stylistique, les meilleures pratiques en vigueur dans l'univers React nous recommandent d'enrober et d'indenter les contenus conditionnels non triviaux, comme ceci :

```
<p>{user.admin && (
  <a href="/admin">Faire des trucs de VIP</a>
)}</p>
```

« Si… Alors… Sinon… »

Si nous voulons l'équivalent d'un `if…else…` , nous utiliserons plutôt l'opérateur ternaire `?…:` , qui est l'équivalent naturel sous forme d'expression. Par exemple :

```
<p>{user.loggedIn ? <WelcomeLabel /> : <LoginLink />}</p>
```

```
<p>{user.admin ? (
  <a href="/admin">Faire des trucs de VIP</a>
) : (
  <a href="/request-admin">Demander à devenir VIP</a>
)}</p>
```

Quand découper notre JSX ?

JSX nous permet d'imbriquer nos expressions et grappes sans limite. Cependant, ce n'est pas toujours une bonne idée en termes de lisibilité. Voyons les trois principaux cas dans lesquels on découpera la grappe en plusieurs expressions recombinées ensuite.

Cas 1 : réutilisation au sein du render

Imaginons qu'un même bouton soit présent plusieurs fois dans l'interface :

```
render() {
  return (
    <Card>
      <CardTitle>
```

```
      Oh le joli titre
      <button onClick={this.logOut}>
        <LogoutIcon />
        Déconnexion
      </button>
    </CardTitle>
    …
    <Footer>
      © 2017 Des Gens Bien™ •
      <button onClick={this.logOut}>
        <LogoutIcon />
        Déconnexion
      </button>
    </Footer>
  </Card>
  )
}
```

Cette répétition est fâcheuse, voire nuisible. JSX reste du JavaScript, au final, donc toutes les techniques de code habituelles, et notamment le recours à des variables, restent utilisables. Voici une version plus « propre » :

```
render() {
  const logoutButton = (
    <button onClick={this.logOut}>
      <LogoutIcon />
      Déconnexion
    </button>
  )
```

```
  return (
    <Card>
      <CardTitle>
        Oh le joli titre
        {logoutButton}
      </CardTitle>
      …
      <Footer>
        © 2017 Des Gens Bien™ •
        {logoutButton}
      </Footer>
    </Card>
  )
}
```

Cas 2 : grappe à l'intérieur d'une prop

La valeur d'une *prop* peut être n'importe quelle expression JavaScript… Y compris une autre expression JSX. C'est assez fréquent pour les *props* contenant des icônes, par exemple, lorsque celles-ci sont elles-mêmes fournies en tant que composants React :

```
<ListItem text="Blah blah" rightSideIcon={<MoreVertIcon />} />

<RaisedButton icon={<SettingsIcon />} label="Paramètres" secondary />
```

Si ce type d'utilisation reste lisible, les choses se compliquent quand la *prop* contient une grappe, même petite :

```
<ListItem
  primaryText="Vous êtes connecté·e en tant que"
```

```
  rightSideIcon={
    <IconButton onClick={this.logOut}>
      <LogoutIcon />
    </IconButton>
  }
  secondaryText={currentUser.email}
/>
```

La lisibilité dégringole rapidement… Dans ces cas-là, on préfèrera définir la grappe de *prop* en amont et stocker la référence, comme ceci :

```
const logoutButton = (
  <IconButton onClick={this.logOut}>
    <LogoutIcon />
  </IconButton>
)
…
return (
  …
  <ListItem
    primaryText="Vous êtes connecté·e en tant que"
    rightSideIcon={logoutButton}
    secondaryText={currentUser.email}
  />
  …
)
```

Cas 3 : le JSX est trop massif (mais…)

Si votre grappe JSX commence à être définie par plusieurs dizaines de

lignes, vous aurez sans doute tendance à vouloir la découper en sous-parties nommées, recombinées à la fin lors de votre `return` .

Toutefois, gardez à l'esprit que bien souvent, une grappe JSX démesurée est plutôt un indice clair que votre composant essaie « d'en faire trop », et qu'il serait sans doute pertinent de le découper en sous-composants plus spécifiques.

Mise en application

À ce stade, il nous manque encore quelques savoir-faire pour créer les composants qui seront affichés de façon conditionnelle. Mais voyons cela par la pratique en choisissant d'afficher un texte « GAGNÉ ! » en bas du plateau uniquement quand… le nombre de secondes de l'heure courante est paire 😁. Ajustons le `render()` de `App.js` comme ceci :

```
render() {
  const won = new Date().getSeconds() % 2 === 0
  return (
    …
    {won && <p>GAGNÉ !</p>}
  )
}
```

Manipulez des listes de composants

Jusqu'ici nous n'avons utilisé que des éléments conditionnels ou en quantité fixe. Mais très fréquemment, nous aurons à représenter un ensemble dynamique de données (série, liste) sous forme de composants React. Par exemple des tâches, des utilisateurs, des cases d'un plateau de jeu…

Les boucles, c'est *so 1960*…

Traditionnellement, en programmation impérative, on recourt alors à des boucles, comme la fameuse boucle `for` venue du langage C qui, en dépit

de sa syntaxe parfaitement obscure pour les néophytes, a infecté un grand nombre d'autres langages :

```
// Pas terrible…
for (let index = 0, len = items.length; index < len; ++index) {
  const item = items[index]
  // …
}
```

Depuis ES2015, on a une syntaxe relativement moins moche :

```
// Mieux mais toujours pas top…
for (const item of items) {
  // …
}
```

Mais outre la laideur syntaxique de ce type de boucles, elles constituent des *instructions*, et ne sont donc pas possibles dans une grappe JSX, qui est une *expression*.

Reformuler le besoin

En pratique, le recours aux boucles est ici un réflexe, et non l'expression naturelle de notre besoin. Nous ne cherchons pas à *faire une boucle*, qui est une approche technique. Nous cherchons à transformer une liste de données en une liste de composants : pour chaque donnée, nous voulons produire le composant aux réglages correspondants.

En programmation fonctionnelle, la manière naturelle de répondre à ce besoin est la méthode `map`, disponible en JavaScript sur les tableaux. Elle prend une fonction de transformation en argument, qui sera appelée pour chaque élément du tableau, et chargée de produire la *transformée* de cet élément. On obtient un nouveau tableau, qui contient toutes les transformées.

Par exemple, pour prendre un tableau de nombres et produire un tableau de leur double, on pourrait écrire ceci :

```
const numbers = [1, 2, 3, 4]

const doubles = numbers.map(x => x * 2) // [2, 4, 6, 8]
```

(Remarquez la façon dont le recours aux fonctions fléchées simplifie l'écriture)

De la même façon, on peut utiliser `map` pour produire une grappe JSX associée à chaque élément d'une liste. Imaginons une liste de personnes :

```
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Claire' },
  { id: 4, name: 'David' },
]
```

Pour produire une liste de liens avec les noms des utilisateurs, on écrirait sans doute ceci :

```
render () {
  return (
    <div className="userList">
      {this.props.users.map((user) => (
        <a href={`/users/${user.id}`}>{user.name}</a>
      ))}
    </div>
  )
}
```

Ou en déstructurant l'élément reçu dans la fonction de rappel :

```
render () {
  return (
    <div className="userList">
      {this.props.users.map(({ id, name }) => (
        <a href={`/users/${id}`}>{name}</a>
      ))}
    </div>
  )
}
```

Pour plus d'informations sur la méthode `map` et ses amies (`filter` , `every` , `some` , `reduce` ...), consultez l'excellente documentation en français sur MDN.

La clé du succès : `key`

Manifestement, JSX accepte des tableaux au sein de ses grappes. Après tout, le résultat d'un appel à `map` est lui-même un tableau... Toutefois, si on s'arrête au code ci-dessus, des soucis peuvent apparaître lorsque les données sous-jacentes évoluent. Par exemple :

- Un tri change leur ordonnancement
- Un élément est inséré quelque part dans la liste
- Un élément est retiré de la liste

Le problème que rencontre React dans ces situations est de pouvoir préserver une association correcte entre une donnée d'origine, dans la liste, et le composant correspondant au sein du DOM virtuel. Il n'existe pas d'heuristique automatique fiable pour réaliser cette association, et faute de mieux, React doit se contenter d'utiliser la position dans la liste.

Imaginons que nous insérons, par exemple, un élément au début d'une liste existante de 1000 éléments. Dans le meilleur des cas, React va considérer que tous les éléments ont changé (puisque tous les éléments existants vont voir leur indice incrémenté) et générer un recalcul de toute la liste. Au pire, React va simplement ajouter le nouvel élément dans le DOM virtuel, ce qui fera doublon avec le 1000e, non rafraîchi. Une opération de suppression produirait également des décalages inattendus, et un réordonnancement pourrait passer inaperçu.

Comme solution, afin de conserver une association correcte entre données d'origine et DOM virtuel, mais aussi afin d'optimiser la retranscription dans le DOM des changements aux données d'origine, React exige que tout élément présent au sein d'un tableau dans une grappe JSX soit doté d'une *prop* technique nommée `key` . Cette *prop* doit impérativement être :

1. Unique au sein du tableau
2. Stable dans le temps (pour la même donnée source, on aura toujours la même valeur de `key=`)

La position dans le tableau ne satisfait pas au second critère, pas plus qu'une valeur aléatoire. Mais en pratique, dans la plupart des cas, nos données sources sont dotées d'une propriété unique (type `id` ou `stamp` , par exemple), que l'on pourra réutiliser.

Ainsi, la version 100% correcte de la liste vue précédemment est la suivante :

```
render () {
  return (
    <div className="userList">
      {this.props.users.map(({ id, name }) => (
        <a href={`/users/${id}`} key={id}>{name}</a>
      ))}
    </div>
```

```
  )
}
```

C’est suffisamment critique pour que, pendant le développement, React nous avertisse systématiquement dans la console du navigateur lorsqu’il rencontre un tableau dont tout ou partie des composants ne sont pas dotés de la *prop* `key` . Il s'agit donc d'une bonne habitude à prendre…

Mise en application

Pour pratiquer les listes, nous allons ajuster notre application.

Récupérer la base de travail

Reprenons d’abord des squelettes de travail depuis le dépôt public pour ce projet, disponible sur GitHub, afin de pouvoir nous concentrer sur la partie React.

1. Allez sur le dépôt GitHub, à l’étiquette **debut-listes** (suivez le lien)
2. Récupérez la feuille de style `HallOfFame.css` et le squelette `HallOfFame.js` . Posez-les dans votre dossier `src/` . Nous avons égaleme,y ajouté un import et une méthode métier de génération de liste de cartes dans `App.js` : récupérez également ce fichier.
3. Ajoutez `lodash.shuffle` aux dépendances :

```
npm install --save lodash.shuffle
```

Afficher le plateau de cartes

Le fichier `App.js` que nous venons de récupérer initialise un champ `cards` dans le composant avec une liste mélangée de paires de cartes. Nous allons ajuster notre `render()` pour afficher cette liste, toutes cartes visibles pour le moment, comme ceci :

```
return (
  <div className="memory">
    <GuessCount guesses={0} />
    {this.cards.map((card, index) => (
      <Card
        card={card}
        feedback="visible"
        key={index}
        onClick={this.handleCardClick}
      />
    ))}
    {won && <p>GAGNÉ !</p>}
  </div>
)
```

Ici, les cartes ne changeront jamais de position dans la liste, et la liste elle-même est figée pour une partie : recourir à l'index comme `key=` est donc acceptable.

Afficher le Tableau d'Honneur

Nous avons récupéré le squelette du composant `<HallOfFame />`, qui attend une liste de parties. Nous ne disposons pas pour le moment de cette information, mais le squelette exporte une constante qui fournit une fausse liste de parties, pour nous permettre d'avancer.

Voici comment nous ajustons le code de `HallOfFame.js` pour produire la liste, en remplaçant le `<tr>FIXME</tr>` par ceci :

```
{
  entries.map(({ date, guesses, id, player }) => (
    <tr key={id}>
```

```
      <td className="date">{date}</td>
      <td className="guesses">{guesses}</td>
      <td className="player">{player}</td>
    </tr>
  ))
}
```

Pour vérifier, il ne nous reste plus qu'à afficher ce tableau à la place de « GAGNÉ ! » dans l'application :

```
import HallOfFame, { FAKE_HOF } from './HallOfFame'

// …

render() {
  // …
  return (
    // …
    {won && <HallOfFame entries={FAKE_HOF} />}
  )
}
```

Un mot sur le « filtrage automatique » des tableaux

Afin de faciliter le recours aux tableaux, JSX va automatiquement ignorer certaines valeurs y figurant, notamment `true` , `false` , `null` et `undefined` . L'idée est de permettre aux fonctions de rappel, passées à `map` , de simplement faire un `return` vide lorsqu'elles ne souhaitent pas produire un composant pour une donnée particulière. En fait, ces valeurs sont ignorées partout, et non uniquement au sein des tableaux.

En pratique, ce genre de filtrage devrait être fait en amont du `render()` , ce

dernier se concentrant sur le rendu pur et dur, plutôt que sur de la logique applicative de filtrage. Mais il est toujours pratique d'avoir la possibilité de filtrer à la volée dans le `map` comme solution de dernière minute.

Une des clés de React est la notion que les données « descendent » à travers l'arborescence de composants. Le mécanisme-clé pour cela, que nous avons déjà vu en partie, ce sont les *props*.

Composants parents et enfants

Cette notion n'est pas tout à fait identique à son homonyme du DOM, car avec React on qualifie de composant *enfant* tout composant défini dans le `render()` du composant *parent*, quel que soit son niveau de profondeur.

On dit donc que le composant qui fournit le `render()` est le *parent*, et que tout composant figurant dans ce `render()` est un *enfant*.

Deux règles sont importantes :

- Une *prop* est toujours passée par un composant parent à son composants enfants : c'est le seul moyen normal de transmission
- Une *prop* est considérée en lecture seule dans le composant qui la reçoit

Props « techniques »

React définit trois *props* « techniques », dont le comportement sort de l'ordinaire :

`key`

Nous avons vu dans la partie précédente que lorsqu'on manipule des tableaux dans une grappe React, il est impératif d'équiper chaque élément du tableau d'une *prop*spéciale `key` . Cela permettra à React de gérer au mieux l'évolution du tableau d'un `render()` à l'autre.

L'absence de cette *prop* entraîne un avertissement en mode développement. Par ailleurs, elle n'est pas consultable par le composant enfant qui la reçoit : elle ne figure pas dans sa liste de *props*.

`children`

Cette *prop* est particulière : elle n'est pas fournie à l'aide d'un attribut, mais en imbriquant des composants à l'intérieur du composant concerné. La liste des composants imbriqués constitue alors la *prop* `children` du composant qui les « entoure ». Elle est donc automatiquement renseignée. Par exemple, dans le JSX suivant :

```
return (

  <FileList>

    <UploadCreator />

    <StatusBar />

  </FileList>

)
```

La *prop* `children` du composant `<FileList />` est un tableau contenant `<UploadCreator />` et `<StatusBar />` .

Notez que `children` peut également être une fonction, ce qui permet l'injection localisée de dépendance par le composant concerné, comme par exemple dans l'API de React Motion.

`dangerouslySetInnerHTML`

Comme son nom l'indique, cette *prop* ne doit être utilisée qu'en tout dernier recours, lorsque vous souhaitez injecter du balisage manuel au sein d'une grappe React. En sécurité supplémentaire, elle requiert comme valeur non pas une `String` mais un objet doté d'une propriété `__html` dont la valeur est le balisage voulu. Voici ce que cela donnerait :

```
function createMarkup() {

  return { __html: 'First &middot; Second' }

}



function MyComponent() {

  return <div dangerouslySetInnerHTML={createMarkup()} />

}
```
Pour rappel, cette technique ne doit être utilisée qu'en dernier recours.

Valeurs par défaut

Un composant peut définir des valeurs par défaut pour ses *props*. Nous avons jusqu'à présent vu la définition de composants à l'aide de fonctions. Il peut donc être tentant d'utiliser la syntaxe ES2015 de valeurs par défaut dans les déstructurations, comme le montre l'exemple suivant :

```
function MyCoolComponent({ l10n = true, name, required = false, value }) {

  // …

}
```
Cependant, cette approche pose deux problèmes.

Tout d'abord, elle ne sera pas possible pour des composants définis à l'aide de classes.

Ensuite, lorsque, dans le prochain chapitre, nous allons définir formellement nos exigences sur les *props*, ces exigences seront vérifiées avant d'appeler la fonction. Elles ne verront donc pas les éventuelles valeurs par défaut.

Pour ces deux raisons, on aura uniquement recours à la propriété statique `defaultProps`. Par « statique », on entend une propriété qui ne diffère pas d'une instance du composant à l'autre, ou d'un appel de la fonction composant à l'autre, mais qui reste la même en étant définie au niveau du « type » du composant lui-même.

Pour un composant défini par une fonction, il suffit de définir la propriété sur la fonction elle-même, comme ceci :

```
function MyCoolComponent({ l10n, name, required, value }) {

  // …

}



MyCoolComponent.defaultProps = {

  l10n: true,

  required: false
```

```
}
```

Ainsi, on garde le même mécanisme technique quel que soit le type de définition du composant (fonction ou classe) d'une part, et on est assuré que les valeurs par défaut sont bien prises en compte par les mécanismes de validation des props que fournit React d'autre part.

Dans notre application, aucun composant n’a de valeur par défaut pour ses *props*, et le reste des notions a déjà été mis en œuvre. Passons donc au chapitre suivant pour acquérir de nouveaux savoir-faire que nous pourrons appliquer à notre jeu de Memory.

Définissez formellement vos props avec PropTypes

Il arrive hélas trop souvent qu'on oublie une *prop*, qu'on fasse une faute de frappe, ou qu'on y mette une valeur inappropriée.

Le problème

Sans mécanisme formel de définition des *props*, le bug est parfois difficile à repérer. Imaginez le composant suivant :

```
function Gauge({ value, max }) {

  // …
```

```
}


Gauge.defaultProps = {

  max: 100,

}
```

Imaginons à présent qu'on l'appelle en oubliant la *prop*, pourtant manifestement obligatoire, `value` ; ou tout simplement en faisant une faute de frappe :

```
<Gauge />


<Gauge Value={50} />


<Gauge value="50" />
```

Assez probablement, nous n'aurons aucune erreur, aucun avertissement dans la console…

Les props sont l'API de votre composant

Le seul moyen pour un composant parent de « configurer » notre composant, de lui indiquer quoi faire, c'est de lui passer les bonnes *props*. Les props sont, véritablement, **l'API de notre composant**. À ce titre, il semble fondamental de définir formellement cette API, et donc la liste de nos *props*.

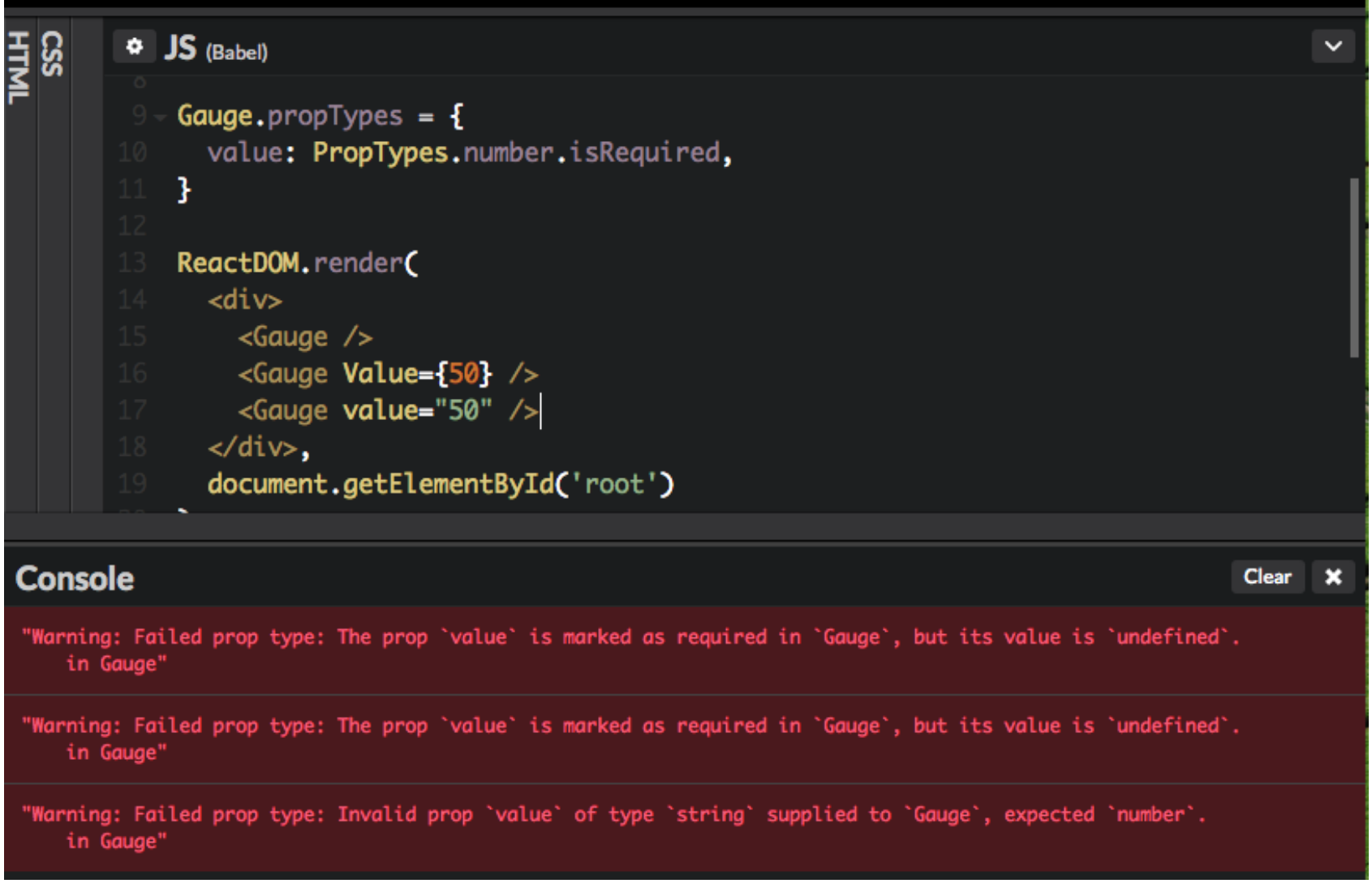
`propTypes`

Pour cela, React examine sur tout composant une propriété statique, cousine en quelque sorte de `defaultProps` que nous avons vue dans le chapitre précédent, nommée `propTypes` . C'est un objet dont les clés sont les noms des *props* attendues, et les valeurs des validateurs de *props*.

Le module standard `prop-types` fournit une série de validateurs basés essentiellement sur le type de valeur de base (nombre, texte, booléen, fonction de rappel…) et des agencements plus complexes (tableaux, objets, énumérations de valeurs ou types possibles…). Pour notre composant `<Gauge/>`, ça pourrait donner ceci :

```
import PropTypes from 'prop-types'


// …


Gauge.propTypes = {
  max: PropTypes.number,
  value: PropTypes.number.isRequired,
}
```

Ainsi, en mode développement en tout cas, si nous oublions une *prop* requise, ou transmettons le mauvais type de valeur, nous aurons un avertissement détaillé dans la console. Par exemple :



Messages d'avertissements dus aux validateurs de props

Si vous avez besoin de validateurs un peu plus travaillés, par exemple pour exiger des nombres entiers dans un intervalle donné, Airbnb a publié le module airbnb-prop-types. Et pour le reste, les validateurs sont juste des fonctions, vous pouvez donc facilement écrire des validations très personnalisées !

Notez bien que le mécanisme de `propTypes` n'est utilisé qu'en développement : dans un build de production (tel que celui obtenu par `npm run build` dans une application générée avec Create React App), ces vérifications sont automatiquement élaguées.

Mise en application

Il est temps d'équiper nos composants de définitions formelles pour leurs *props*. Commençons par ajouter le module officiel `prop-types` à nos dépendances :

```
npm install --save prop-types
```

Le composant Carte

C'est une fonction : on ajoute donc la propriété `propTypes` *a posteriori* :

```
import PropTypes from 'prop-types'


// La fonction Card ici…


Card.propTypes = {
  card: PropTypes.string.isRequired,
  feedback: PropTypes.oneOf([
    'hidden',
    'justMatched',
    'justMismatched',
    'visible',
  ]).isRequired,
  onClick: PropTypes.func.isRequired,
}
```

Remarquez ici le combinateur `oneOf` , qui fonctionne comme une énumération, en limitant les valeurs à une série précise.

Le composant Compteur de tentatives

On suit le même principe, pour l'unique *prop* `guesses` , numérique et requise :

```
import PropTypes from 'prop-types'


// La fonction GuessCount ici…


GuessCount.propTypes = {
```

```
  guesses: PropTypes.number.isRequired,
}
```

Le composant Tableau d'honneur

Cette fois-ci la définition est plus fine, la *prop* `entries` étant un tableau, requis, d'objets ayant une structure bien précise. Voici ce que ça donne :

```
import PropTypes from 'prop-types'

// La fonction HallOfFame ici…

HallOfFame.propTypes = {
  entries: PropTypes.arrayOf(
    PropTypes.shape({
      date: PropTypes.string.isRequired,
      guesses: PropTypes.number.isRequired,
      id: PropTypes.number.isRequired,
      player: PropTypes.string.isRequired,
    })
  ).isRequired,
}
```

On retrouve ici deux combinateurs classiques :

- `arrayOf` , qui indique que la *prop* sera un tableau de valeurs dont la définition est fournie en argument.
- `shape` , qui décrit un objet dont les clés sont connues, en précisant les types de leurs valeurs.

Le composant applicatif ne reçoit pas de *props* particulières, aussi il est inutile d'y définir `propTypes`.

Gérez la complexité avec les classes ES2015

Jusqu'ici, nous avons utilisé uniquement des fonctions pour créer des classes : cela suffisait à nos besoins. Néanmoins, nous devrons parfois recourir à des classes pour des cas plus complexes.

Fonction ou classe, comment choisir ?

La principale limite des fonctions, c'est qu'elles se restreignent en fait au rendu, c'est-à-dire à la constitution du DOM virtuel du composant, avec pour seule information entrante la liste des *props* issue du parent.

En conséquence, si nous avons besoin de préserver des informations d'un

rendu à l'autre, pour faire une horloge par exemple, ou bien un compteur, ou encore persister des données, cette approche ne suffit plus. Il est en effet impossible d'utiliser un état local sur un composant défini par une simple fonction.

Autre type de besoin, dont voici un exemple : si notre composant doit interagir de façon spécifique avec la couche de rendu, notamment le DOM du navigateur, pour s'interfacer avec des API web ou des éléments spécifiques du DOM, et ce afin d'enrober une bibliothèque tierce par un « emballage React », nous aurons besoin de *méthodes de cycle de vie*, mais qui nécessitent une instance qui dure dans le temps, et donc une définition de composant par classe. Nous verrons cela dans un chapitre ultérieur.

De façon plus subjective, si nous souhaitons ajouter au rendu proprement dit des méthodes métier supplémentaires, les regrouper avec le `render()` dans une entité cohérente, comme une classe, sera plus élégant que le fait de simplement lister une série de fonctions simples au sein d'un module.

Cependant, les fonctions simples restent largement suffisantes pour la grande majorité des composants, ceux appelés « présentationnels » ou « bêtes ». Dès lors qu'une simple fonction suffit, cette approche est à privilégier. Elle est plus légère et offre des avenues futures d'optimisation automatique intéressantes.

Rappels sur une classe

Avec ES2015, une classe est définie comme suit :

```
class Person {

  constructor(first, last) {

    this.first = first

    this.last = last

  }


  fullName() {
```

```
    return `${this.first} ${this.last}`
  }
}
```

Lorsque la classe en spécialise une autre, on emploie alors `extends` pour le signaler. On peut ensuite peut utiliser `super` pour recourir au constructeur hérité, ou aux méthodes héritées dans leurs versions originales :

```
class CoolComponent extends Component {
  constructor(props) {
    super(props)
    this.state = { collapsed: props.initialCollapsed }
  }

  render() {
    // …
  }
}
```

Intégrer `defaultProps` et `propTypes`

Il reste bien sûr possible de déclarer ces propriétés statiques comme on le faisait pour les composants définis par fonctions : après coup, sur l'identifiant de la fonction. Mais avec les initialiseurs de champs prévus dans ES2018 et déjà implémentés un peu partout, on préfèrera généralement les lister en haut du corps de classe, avec le mot-clé `static` :

```
class CoolComponent extends Component {
  static defaultProps = {
    initialCollapsed: false,
  }
```

```
  static propTypes = {
    initialCollapsed: PropTypes.bool.isRequired,
    items: PropTypes.arrayOf(CoolItemPropType).isRequired,
  }

  constructor(props) {
    super(props)
    this.state = { collapsed: props.initialCollapsed }
  }

  render() {
    // …
  }
}
```

Un mot sur `createClass` …

Dans des didacticiels plus anciens, vous trouverez encore de nombreuses références à l'API d'origine de création de composant avec React : `React.createClass` . Cette API était là car React n'avait pas encore adopté ES2015, et avait donc besoin d'un mécanisme à lui pour « assembler » rapidement une classe, avec ses méthodes, validateurs et valeurs par défaut de *props*, état par défaut, etc.

Cette API est considérée comme dépréciée aujourd'hui, mais le plus important lorsqu'on lit du code écrit avec elle, ou lorsqu'on migre de ce type de code vers des classes ES2015, est de garder en tête une différence fondamentale : avec `React.createClass` , toutes les méthodes définies étaient *auto-bound*, c'est-à-dire que `this` y désignait toujours, par défaut, l'instance du composant.

Ce confort avait toutefois un coût non négligeable en termes de performances, surtout lorsqu'il n'était pas nécessaire. Avec les classes ES2015, il nous appartient de décider au cas par cas, pour nos méthodes, celles qui nécessitent un *binding* garanti, et de prendre les mesures nécessaires pour cela. Nous y reviendrons en détail dans un prochain chapitre.

Le composant `<App />` utilise déjà cette forme de déclaration. Quant à l'autre composant que nous écrirons sous cette forme, pour la saisie du nom une fois le plateau terminé, nous devons d'abord explorer les notions d'état local et de gestion de formulaires. C'est pour bientôt !

Types de méthodes métier

Dans nos composants définis par des classes, les méthodes sont de différentes natures :

- Calculs et construction de données, qu'on souhaite par exemple sortir du `render()` pour que celui-ci se concentre sur sa tâche principale : structurer le DOM virtuel.
- Méthodes de cycle de vie (nous les verrons plus en détail dans le dernier chapitre de cette partie)
- Gestionnaires d'événements ; il est possible de les définir à la volée dans `render()`, mais lorsqu'ils ne dépendent que de l'événement reçu, on préfèrera alors les extraire pour des raisons de performance.

Exemple de méthode métier de calcul

Dans le code suivant, un calcul quelque peu complexe est sorti du `render()` pour découper les responsabilités :

```
class TrackerScreen extends Component {

  // …

  overallProgress() {

    const { goals, todaysProgress } = this.props

    const [totalProgress, totalTarget] = getDayCounts(goals, todaysProgress)

    return totalTarget === 0 ? 0 : Math.floor(totalProgress * 100 / totalTarget)

  }



  render() {

    return (

      <Card>

        <CardTitle

          title={formatDate(new Date(), 'LL')}

          subtitle={<Gauge value={this.overallProgress()} />}

        />

        {/* … */}

      </Card>

    )

  }
```

```
}
```

De cette façon, `render()` se concentre sur son boulot : produire la « grappe » React du composant, sans être « pollué » par du code de calcul.

Le souci du `this`

Dans l'exemple précédent, la méthode métier était immédiatement appelée (`this.overallProgress()`), du coup au sein de `overallProgress()` le `this` était bien le composant courant, de sorte que `this.props` était correct.

En revanche, très fréquemment, nous aurons des méthodes métier qui seront passées par référence dans des *props*, notamment pour les gestionnaires d'événements et fonctions de rappel (*callbacks*). En JavaScript, le passage par référence d'une fonction « classique » (définie avec `function` , ou la syntaxe courte de méthode) ne définit pas le this associé. Voyez plutôt :

```
class LoginScreen extends Component {

  logIn(event) {

    event.preventDefault()

    console.log('LOGIN', this)

  }



  render() {

    return (

      <form onSubmit={this.logIn}>

        {/* … */}

        <p>

          <button type="submit">Connexion !</button>

        </p>

      </form>

    )

  }

}
```

Si on clique sur le bouton pour envoyer le formulaire, la console affiche alors :

```
"LOGIN" undefined
```

Une rustine en attendant la suite

Il existe plusieurs manières de remédier à ce problème, que nous explorerons dans le chapitre suivant. Nous allons pour le moment contourner l'obstacle en évitant de passer une référence sur fonction classique. Nous allons plutôt recourir à une fonction fléchée créée à la volée :

```
class LoginScreen extends Component {

  logIn(event) {

    event.preventDefault()

    console.log('LOGIN', this)

  }



  render() {

    return (

      <form onSubmit={event => this.logIn(event)}>

        {/* … */}

        <p>

          <button type="submit">Connexion !</button>

        </p>

      </form>

    )

  }

}
```

Cette variation fonctionne, certes, mais n'est pas particulièrement optimisée en termes de performances, comme nous le détaillerons au prochain chapitre.

Les méthodes `generateCards()` et `handleCardClick()` de notre composant `<App />` sont des méthodes métier, même si elles n'ont pas (du moins pour l'instant) besoin d'utiliser `this` . Avant de les enrichir et d'en ajouter, il est utile de découvrir comment garantir le `this` lorsqu'on en a besoin : c'est le but du prochain chapitre.

Faites référence au bon « this » dans vos fonctions

Pourquoi faire attention aux fonctions fléchées ?

On voit très fréquemment des appels à des méthodes métier depuis le JSX à l'aide de fonctions fléchées :

```
<AwesomeConfirm onConfirm={() => this.triggerAwesome()} />

{

  items.map(item => (

    <CoolItem key={item.id} item={item} onEdit={() => this.editItem(item)}
```

```
  ))
}
```

Le problème ici est qu'à chaque `render()` produisant cette grappe, on crée de nouvelles fonctions pour les passer aux *props* : outre le coût de création et d'occupation mémoire associé, les composants ainsi paramétrés ont l'impression de recevoir des *props* différentes à chaque fois. Elles considéreront qu'ils devront re-renderer eux-mêmes, y compris lorsque cela est en fait superflu.

Dans le second cas ci-dessus, la fonction fléchée à la volée est compréhensible, puisque l'appel de la méthode métier requiert une donnée fournie par la *closure* (ici le contexte en cours dans la fonction de rappel passée à `map` , pour aller y chercher `item`).

Or, dans le premier cas, la fonction fléchée n'est qu'un passe-plat : elle n'apporte aucun élément de contexte supplémentaire à la méthode métier invoquée. On voudrait pouvoir juste dire :

```
<AwesomeConfirm onConfirm={this.triggerAwesome} />
```

Ainsi, on passe la même valeur au composant à chaque `render()` , ce qui est plus rapide, moins gourmand en mémoire, et peut éviter au composant des re-renderings superflus.

Il y a un cependant un souci : en passant une référence à une fonction classique (déclarée avec `function` ou avec la syntaxe courte de méthode au sein d'un objet ou d'une classe), le `this` en vigueur disparaît : il ne fait pas « partie » de la référence passée.

Alors comment faire ? Il existe trois approches, chacune ayant ses avantages et ses inconvénients.

Première approche : `bind` dans le constructeur

On peut choisir cette approche, finalement très traditionnelle, qui consiste

à faire en sorte que chaque fois que notre propre composant est instancié, il se dote de ses propres « variantes » de la méthode. Ces variantes lui sont alors explicitement associées, c'est-à-dire qu'elles l'utiliseront toujours en tant que `this`. Voici un exemple pour illustrer :

```
class LoginScreen extends Component {

  constructor(props) {

    super(props)

    this.logIn = this.logIn.bind(this)

  }



  // …

}
```

Deuxième approche : initialiseur de champ

Une seconde approche, particulièrement populaire depuis mi-2017 dans les didacticiels, présentations et autres supports, consiste à utiliser la syntaxe d'initialiseur de champs qui devrait faire partie d'ES2018 (et est, entre-temps, déjà pas mal prise en charge nativement, et à défaut simulée par Babel) :

```
class LoginScreen extends Component {

  logIn = (event) => {

    // …

  }



  // …

}
```

En fait, cette syntaxe revient très précisément à faire ceci, en plus concis :

```
class LoginScreen extends Component {

  constructor(props) {

    super(props)

    this.logIn = (event) => {

      // …

    }

  }


  // …

}
```

Vu que les fonctions fléchées n'ont pas de contexte d'invocation, et notamment pas leur propre `this` mais celui le plus proche défini dans les portées englobantes (les *closures*), une telle méthode utilise bien le `this` en vigueur au niveau du constructeur.

Troisième approche : `@autobind`

Mon approche préférée, pour les raisons que je vais détailler juste après, repose sur les *décorateurs*, une syntaxe qui ne devrait devenir officielle qu'en 2019, voire 2020, simulable entre-temps avec Babel :

```
import autobind from 'autobind-decorator'


// …


class LoginScreen extends Component {

  @autobind

  logIn(event) {

    // …

  }
```

```
  // …
}
```

Précis, pointu.

Quelle approche choisir ?

L'approche 1, le `bind` explicite dans le constructeur, a l'avantage d'être… explicite. L'intention est claire : on a besoin que cette méthode soit *bound*, c'est-à-dire qu'on veut garantir son `this`, ce qui dans un composant React revient à dire qu'on va passer cette méthode par référence ici et là sans avoir à se préoccuper de cette garantie. En revanche, le code qui fournit cette garantie - et exprime ce besoin - est distant de la déclaration de la méthode elle-même, qui peut se trouver des dizaines de lignes plus bas. En effet, si je regarde la déclaration de `logIn`, rien ne m'indique qu'elle est *bound*…

L'approche 2, l'initialisation de champ par fonction fléchée, est par définition positionnée au niveau de la déclaration de la méthode. Ce qui me gêne ici, c'est que l'intention n'a rien d'évident.

De la même manière qu'on a vu fleurir un peu partout des créations de fonctions du type `var doSomething = function(x) { … }` par simple copier-coller irréfléchi au lieu de recourir à une véritable déclaration de fonction (`function doSomething(x)`), avantageuse dans la grande majorité des cas, on risque de voir tout le monde se mettre à déclarer l'ensemble de ses méthodes par fonction fléchée au lieu de la syntaxe concise.

Ce type de culte du cargo a un coût : de nombreuses méthodes seront inutilement dupliquées en mémoire, à chaque instance de notre composant, ralentissant au passage la construction de celles-ci.

L'approche 3, le recours à un décorateur, est à mon sens la meilleure de toutes. **Non seulement elle est positionnée sur la déclaration,**

mais de plus son intention est claire : il est alors moins probable que les développeurs copient-collent le décorateur sans réfléchir pour toutes leurs déclarations de méthodes.

L'inconvénient pour le moment (automne 2017) est qu'en raison de l'historique de spécification des décorateurs, il n'existe pas de plugin Babel à jour sur la nouvelle spécification, et que Create React App, notamment, préfère ne pas utiliser automatiquement l'ancienne implémentation. Dans une application gérée par Create React App, comme c'est le cas pour ce cours, nous n'avons donc pas accès à cette syntaxe.

Du coup, dans le cadre de notre cours, je recommande d'utiliser l'approche 2, mais en ajoutant un commentaire explicitant l'intention au-dessus de l'initialisation, comme ceci :

```
class LoginScreen extends Component {

  // This method is declared using an arrow function initializer solely

  // to guarantee its binding, as we cannot use decorators just yet.

  logIn = (event) => {

    // …

  }


  // …

}
```

Mise en application

La méthode `handleCardClick()` de notre composant `<App />` va avoir besoin de `this` , et pour le moment cela pose problème. Pour le vérifier, modifions son code :

```
handleCardClick(card) {

  console.log(card, this)
```

```
}
```

Cliquez à présent sur une carte, par exemple une avec une pomme : que dit la console ?

```
🍎 undefined
```

Notre code est transpilé (converti) par Babel, qui par défaut garantit le mode strict partout (ce qui est une excellente chose). Résultat, dans les fonctions passées par référence, `this` est `undefined` . Puisque Create React App ne permet pas pour le moment (automne 2017) de recourir aux décorateurs, nous utiliserons la syntaxe à base d'initialiseur à la volée, avec un petit commentaire explicitant l'intention :

```
// Arrow fx for binding

handleCardClick = (card) => {

  console.log(card, this)

}
```

Retestons le clic :

```
🍎 App {props: {…}, context: {…}, refs: {…}, updater: {…}, cards: Array(36)
```

Voilà qui est mieux.

Mettez en place un état local

Nos composants ont pour le moment uniquement utilisé les *props* pour gouverner leur apparence et leur fonctionnement. Mais une fois nos composants renderés, ils commencent à « vivre » en interaction avec l'utilisateur ou le système. Il faut alors qu'ils puissent faire évoluer leurs données sous-jacentes, et pas forcément que les *props* bougent d'elles-mêmes.

Valeurs des champs d'un formulaire, état déplié/replié d'un élément, heure courante… ce sont ici juste quelques exemples d'informations qui évoluent dans le temps une fois que le composant est présent dans le DOM.

Pour représenter ces données, qui persistent d'un `render()` à l'autre ou

simplement re-déclenchent un rendu, React utilise l'état local des composants.

Où est situé l'état local ?

L'état local est présent à l'intérieur d'un composant, dans sa propriété `state`. On doit obligatoirement utiliser une définition de composant par classe, et non une simple fonction. L'état par défaut est `null`, il faudra donc le plus souvent l'initialiser manuellement dans le constructeur, soit de manière explicite :

```
class Accordion extends Component {

  constructor(props) {

    super(props)

    this.state = { expandedPanels: new Set() }

  }


  // …

}
```

Ou, implicitement, par un initialiseur de champ :

```
class Accordion extends Component {

  state = { expandedPanels: new Set() }


  // …

}
```

Personnellement, je préfère la première solution.

Qui a accès à l'état local ?

Uniquement votre composant : les parents, tout comme les enfants, ne

peuvent pas manipuler votre état local (ils n'en ont pas la possibilité technique). C'est véritablement un état **local**, il ne regarde que vous.

En conséquence, si vous avez des éléments d'état à « partager » entre plusieurs composants, React vous demandera de *« lift state »*, c'est-à-dire de faire remonter ces données vers l'état local du plus proche composant ancêtre commun dans la grappe, celui qui contient dans sa grappe l'ensemble des composants souhaitant « partager » l'info. Il vous appartiendra alors de :

1. faire « redescendre » ces infos à coup de *props* jusqu'aux composants qui en ont besoin

2. faire « remonter » les demandes d'évolution à coup de *props* de type fonction, fournies par le composant doté de l'état local, et utilisées par les composants qui en ont besoin.

Imaginons un accordéon avec des composants « panneau d'accordéon », qui gouvernent leur ouverture-fermeture, à ceci près que l'accordéon doit permettre de tout ouvrir ou de tout refermer. Il doit donc être à l'origine de cette information sur les états ouverts ou fermés, sans quoi il n'aurait aucun moyen d'action dessus, à moins de découper les choses de façon bancale. On pourrait imaginer le composant `<AccordionPanel />` , qui n'aurait donc pas d'état propre, comme ceci :

```
function AccordionPanel({ item: { title, content }, open, onToggle }) {
  return (
    <section className="accordion-panel">
      <h1 className="accordion-header" onClick={() => onToggle(item)} />
      <div className="accordion-content">{content}</div>
    </section>
  )
}
AccordionPanel.defaultProps = {
```

```
  open: false,
}
AccordionPanel.propTypes = {
  item: ItemPropType.isRequired,
  open: PropTypes.bool,
  onToggle: PropTypes.func.isRequired,
}
```

Ici, le panneau tire son contenu et son état (ouvert/fermé) de ses *props*, il n'a aucun état propre. Du coup, le traitement de la demande de bascule ouvert/fermé est délégué à une fonction de rappel passée en *prop*, `onToggle` , puisqu'il n'a rien « localement » qu'il puisse modifier pour changer son statut ouvert.

Au-dessus, on aurait un composant Accordéon qui ressemblerait sans doute à ceci :

```
class Accordion extends Component {
  static propTypes = {
    items: PropTypes.arrayOf(ItemPropType).isRequired,
  }

  constructor(props) {
    super(props)
    this.state = { expandedPanels: new Set() }
  }

  render() {
    const { items } = this.props
    return (
      <div className="accordion">
```

```
      {items.map((item) => (
        <AccordionPanel
          key={item.id}
          item={item}
          onToggle={this.togglePanel}
          open={this.state.expandedPanels.has(item)}
        />
      ))}
    </div>
  )
}

// This method is declared using an arrow function initializer solely
// to guarantee its binding, as we cannot use decorators just yet.
togglePanel = (panel) => {
    const expandedPanels = this.state.expandedPanels
    if (expandedPanels.has(panel)) {
      expandedPanels.remove(panel)
    } else {
      expandedPanels.add(panel)
    }
    this.setState({ expandedPanels })
  }
}
```

Remarquez le fait que l’accordéon gère lui-même les états ouvert/fermé de ses panneaux via sa variable d’état local `expandedPanels` , et qu'il la met à jour non pas directement, mais à l’aide de `setState()` .

Appeler cette méthode est indispensable à la prise en compte de la

modification par React, mais son véritable fonctionnement est souvent mal compris. Nous explorerons ceci plus en détail dans le prochain chapitre.

Mise en application

Maintenant que nous savons gérer un état local, nous avons pas mal de travail à faire dans `<App/>`. En effet, tout l'état courant de notre jeu est stocké comme état local dans ce composant racine.

Commençons par remplacer le champ temporaire `cards` par une initialisation du champ « officiel » `state`, en bonne et due forme :

```
class App extends Component {

  state = {

    cards: this.generateCards(),

    currentPair: [],

    guesses: 0,

    matchedCardIndices: [],

  }


  // …
```

La liste des cartes fait désormais partie de notre état local. Quelles autres données y retrouve-t-on ?

- `currentPair` est un tableau représentant la paire en cours de sélection par la joueuse. À vide, aucune sélection en cours. Un élément signifie qu'une première carte a été retournée. Deux éléments signifient qu'on a retourné une seconde carte, ce qui déclenchera une analyse de la paire et l'avancée éventuelle de la partie.

- `guesses` est le nombre de tentatives de la partie en cours (nombre de paires tentées, pas nombre de clics)

- `matchedCardIndices` liste les positions des cartes appartenant aux paires déjà réussies, et donc visibles de façon permanente.

Le début de notre méthode `render()` change également. Elle va aller chercher les infos utiles dans l'état local courant, et les utiliser pour nos *props* et pour la source de notre liste de cartes :

```
render() {
  const { cards, guesses, matchedCardIndices } = this.state
  const won = matchedCardIndices.length === cards.length
  return (
    <div className="memory">
      <GuessCount guesses={guesses} />
      {cards.map((card, index) => (
  // …
```

La définition de `won` n'est donc plus un simulacre fondé sur le moment présent, mais vient bien du fait que toutes les cartes ont été retournées de façon permanente.

Nous disposons maintenant d'un état qui décrit l'avancement du jeu. Il faut donc arrêter d'afficher de base toutes les cartes (ce qui tuerait évidemment la partie dans l'œuf). Ajoutez la méthode métier `getFeedbackForCard()` que voici à la classe :

```
getFeedbackForCard(index) {
  const { currentPair, matchedCardIndices } = this.state
  const indexMatched = matchedCardIndices.includes(index)

  if (currentPair.length < 2) {
    return indexMatched || index === currentPair[0] ? 'visible' : 'hidden'
  }
```

```
  if (currentPair.includes(index)) {
    return indexMatched ? 'justMatched' : 'justMismatched'
  }

  return indexMatched ? 'visible' : 'hidden'
}
```

Vous vous apercevez qu'on y utilise `this` (pour aller consulter l'état), mais comme nous appellerons cette méthode directement (et non par référence), nous n'avons pas besoin de garantir le `this` avec une syntaxe à base d'initialiseur. Appelons-la justement depuis le `render()` :

```
{
  cards.map((card, index) => (
    <Card
      card={card}
      feedback={this.getFeedbackForCard(index)}
      key={index}
      onClick={this.handleCardClick}
    />
  ))
}
```

Vous devriez à présent voir toutes les cartes cachées, l'état initial n'ayant aucune position dans `matchedCardIndices` .

Il nous faut à présent faire évoluer l'état au fil des clics, en commençant par le champ d'état `currentPair` , qui permet de constituer la paire actuellement tentée. Modifions `handleCardClick()` comme suit :

```
// Arrow fx for binding

handleCardClick = index => {

  const { currentPair } = this.state


  if (currentPair.length === 2) {

    return

  }


  if (currentPair.length === 0) {

    this.setState({ currentPair: [index] })

    return

  }


  this.handleNewPairClosedBy(index)

}
```

C'est désormais l'index de la carte, et non son symbole (ambigu car présent deux fois), qui nous intéresse. Il faut donc commencer par fournir cette information au composant `<Card/>` :

```
{

  cards.map((card, index) => (

    <Card

      card={card}

      feedback={this.getFeedbackForCard(index)}

      index={index}

      key={index}

      onClick={this.handleCardClick}

    />
```

```
  ))
}
```

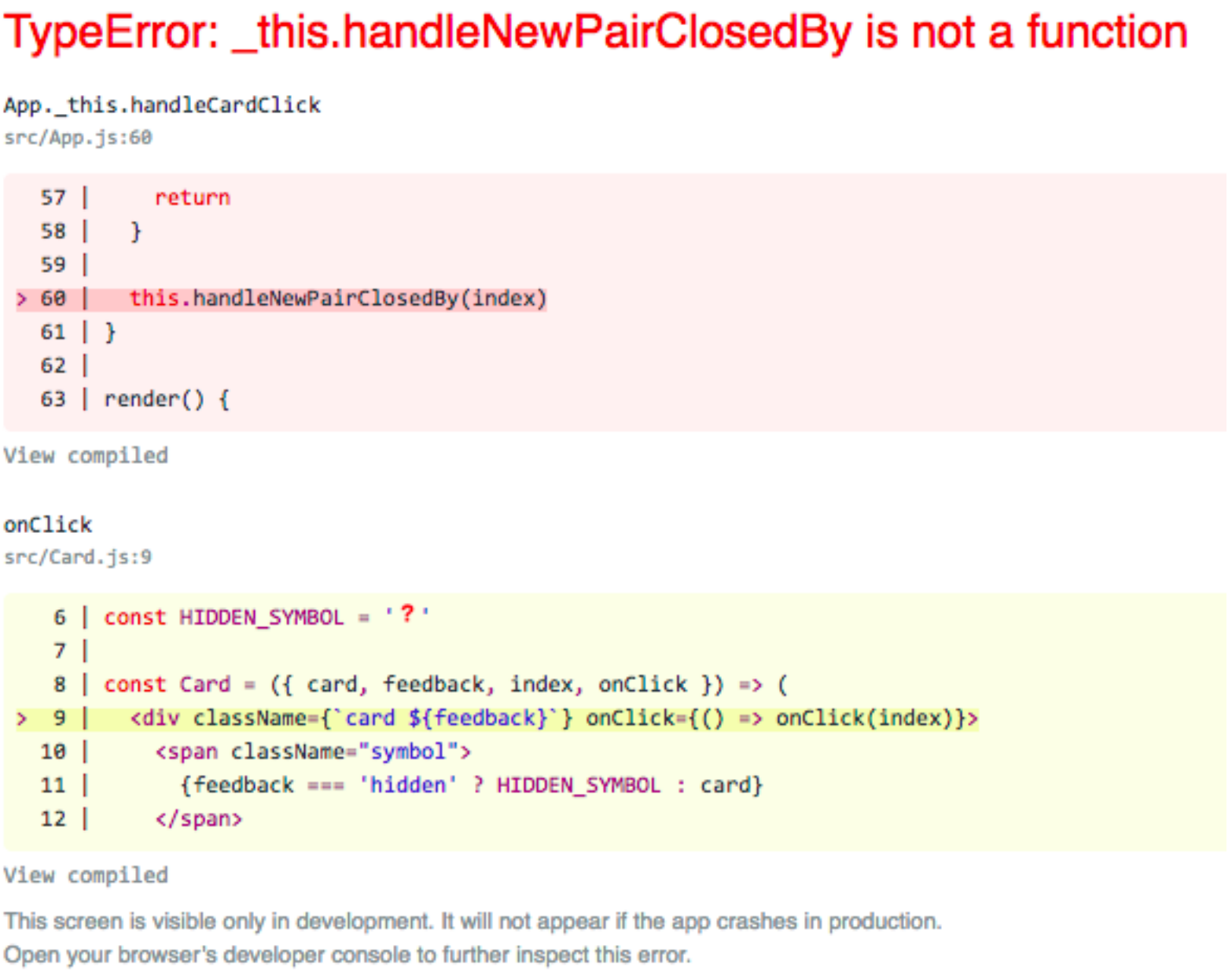
Ensuite, ce composant peut la renvoyer dans la gestion de clic… sans oublier de la déstructurer depuis ses *props* et de déclarer la *prop* en question dans ses `propTypes` :

```
const Card = ({ card, feedback, index, onClick }) => (
  <div className={`card ${feedback}`} onClick={() => onClick(index)}>
    <span className="symbol">
      {feedback === 'hidden' ? HIDDEN_SYMBOL : card}
    </span>
  </div>
)

Card.propTypes = {
  card: PropTypes.string.isRequired,
  feedback: PropTypes.oneOf([
    'hidden',
    'justMatched',
    'justMismatched',
    'visible',
  ]).isRequired,
  index: PropTypes.number.isRequired,
  onClick: PropTypes.func.isRequired,
}
```

À présent, si vous cliquez sur une première carte, celle-ci doit s’afficher sans effet particulier. En revanche, en cliquant sur une deuxième carte, la méthode `handleNewPairClosedBy()` n’étant pas encore écrite, une erreur

sera générée.

Create React App se met en quatre pour nous fournir un affichage d'erreur de très grande qualité dans le navigateur au moyen d'un *overlay* refermable. Constatez par vous-même :



Magnifique pile d'appels cliquable sur erreur d'exécution grâce à Create React App

Avant d'écrire la méthode manquante, nous allons prendre le temps, dans le prochain chapitre, d'apprendre les subtilités de `setState()` .

Mettez à jour l'état local avec « setState »

Appeler setState avec un objet, ça fait quoi ?

Ça envoie une *série de modifications* à l’état local du composant. Il faut bien comprendre que ce n’est *pas le nouvel état* dans son intégralité, mais juste des *différences*. C’est pratique, car ça évite de toujours devoir envoyer un état complet ; si on veut juste changer une propriété de l’état, on se contente de cette propriété :

```
this.setState({ open: true }) // modifie uniquement cette propriété
```

Les soucis commencent lorsqu'on oublie cet aspect « différentiel » et que l'on omet donc de réinitialiser certains champs. Imaginons un état représentant les champs d'un formulaire que l'on souhaiterait remettre à vide :

```
class Form extends Component {
  constructor(props) {
    super(props)
    this.state = { name: '', target: 5, units: '' }
  }

  // …

  resetForm() {
    this.setState({}) // Ooops !
  }
}
```

Ici, `resetForm()` ne réinitialisera en fait rien du tout, car ce que l'on envoie n'est autre qu'une
une *liste vide de modifications*. Voici une implémentation correcte :

```
const DEFAULT_STATE = { name: '', target: 5, units: '' }

class Form extends Component {
  constructor(props) {
    super(props)
    this.state = { ...DEFAULT_STATE }
  }
```

```
  // …


  resetForm() {
    this.setState(DEFAULT_STATE) // Mieux !
  }
}
```

Attention, c’est asynchrone !

Un autre aspect fondamental de `setState()` : il est asynchrone. Il traitera donc les mises à jour plus tard, au moment le plus opportun, par lots. Voici un exemple d’utilisation naïve qui ne fonctionnera pas du tout :

```
doSomethingWrong() {
  // this.state.open est `false`
  this.setState({ open: true })
  console.log(this.state.open === true) // `false` : pas encore…
}


doSomethingSuperWrong() {
  // this.state.count == 0
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  console.log(this.state.count) // 0
  // Et même une fois pris en compte, ce sera 1, pas 3, vu que
  // tout le long de cette méthode, `this.state.count` valait 0.
}
```

Mais… pourquoi ?!

Elle ne fonctionnera pas pour des raisons de performance. Cela nous permet, dans nos codes, d'appeler `this.setState()` à de multiple occasions, avec des modifications diverses et variées, sans ralentir le système. React fusionne les demandes et applique le résultat au meilleur moment, en garantissant simplement que les modifications seront appliquées avant la prochaine étape de cycle de vie (concept que nous explorerons plus en détail au prochain chapitre).

Et ce 2e argument de rappel ?

La méthode `setState()` accepte un callback en 2e argument : est-il là pour nous dire que l'état est à jour ? Pas tout à fait : il est là pour nous dire que l'ensemble des appels à `setState()` ont été traités et que le composant a de nouveau fait un rendu. En pratique, on préfère utiliser la méthode de cycle de vie `componentDidUpdate()` .

En pratique, si on veut se garantir le résultat de modifications incrémentales ou se prémunir d'autres soucis liés à la nature asynchrone de `setState()` , on utilisera plutôt un premier argument de type fonction.

Appeler `setState()` avec une fonction

L'approche la plus fiable, générique et pérenne de `setState()` consiste à l'appeler avec un seul argument qui est une fonction de rappel, ce qui constitue sa signature canonique. La fonction a deux arguments : l'état d'avant (qui tient compte de toute tentative de `setState()` exprimée auparavant) et les *props* en vigueur du composant. Elle renvoie un objet qui sera traité comme celui passé jusqu'ici : une série de modifications à apporter à l'état.

Ainsi, on peut par exemple faire de l'incrémental :

```
doSomethingRight() {
  // this.state.count vaut 0
  handleClick() {
```

```
    this.setState(
      (prevState, props) => ({ count: prevState.count + props.inc })
    )
    this.setState(
      (prevState, props) => ({ count: prevState.count + props.inc })
    )
    this.setState(
      (prevState, props) => ({ count: prevState.count + props.inc })
    )
  }
}
```

Mise en application

À présent que nous comprenons bien `setState()`, écrivons la méthode manquante, `handleNewPairClosedBy()`. Celle-ci permet d'arbitrer la paire fraîchement constituée et de faire effectivement avancer la partie :

```
const VISUAL_PAUSE_MSECS = 750

class App extends Component {
  …
  handleNewPairClosedBy(index) {
    const { cards, currentPair, guesses, matchedCardIndices } = this.state

    const newPair = [currentPair[0], index]
    const newGuesses = guesses + 1
    const matched = cards[newPair[0]] === cards[newPair[1]]
    this.setState({ currentPair: newPair, guesses: newGuesses })
    if (matched) {
```

```
      this.setState({ matchedCardIndices: [...matchedCardIndices, ...newPai
    }
    setTimeout(() => this.setState({ currentPair: [] }), VISUAL_PAUSE_MSECS
  }
  …
}
```

Notez les multiples appels à `this.setState()`, qui se préoccupent de divers champs, et seront tous en réalité exécutés d'un bloc avant le prochain `render()`, au moment le plus opportun. Exception toutefois : le dernier appel, placé dans un `setTimeout(…, VISUAL_PAUSE_MSECS)` qui le retarde de 750ms, afin de laisser le temps de visualiser son résultat, surtout en cas de paire incorrecte qui sera alors masquée (`currentPair` repassant à vide).

Dans le cas où la syntaxe `[...matchedCardIndices, ...newPair]` ne vous serait pas familière, il s'agit d'un *spread*, qui « étale » le contenu des itérables (en l'occurrence des tableaux) à la volée. Dans ce cas, le résultat est injecté au sein d'un nouveau tableau, délimité par les crochets de part et d'autre. Puisqu'il n'y a ici que deux tableaux, et que ce sont effectivement des `Array` JavaScript, une version plus traditionnelle de ce code aurait été `matchedCardIndices.concat(newPair)`.

À ce stade, vous pouvez jouer jusqu'à arriver en fin de partie, et voir alors s'afficher le « faux » tableau d'honneur. Autant dire que je vous ai perdu·e·s pour une bonne vingtaine de minutes… 😁

Apprivoisez le cycle de vie des composants

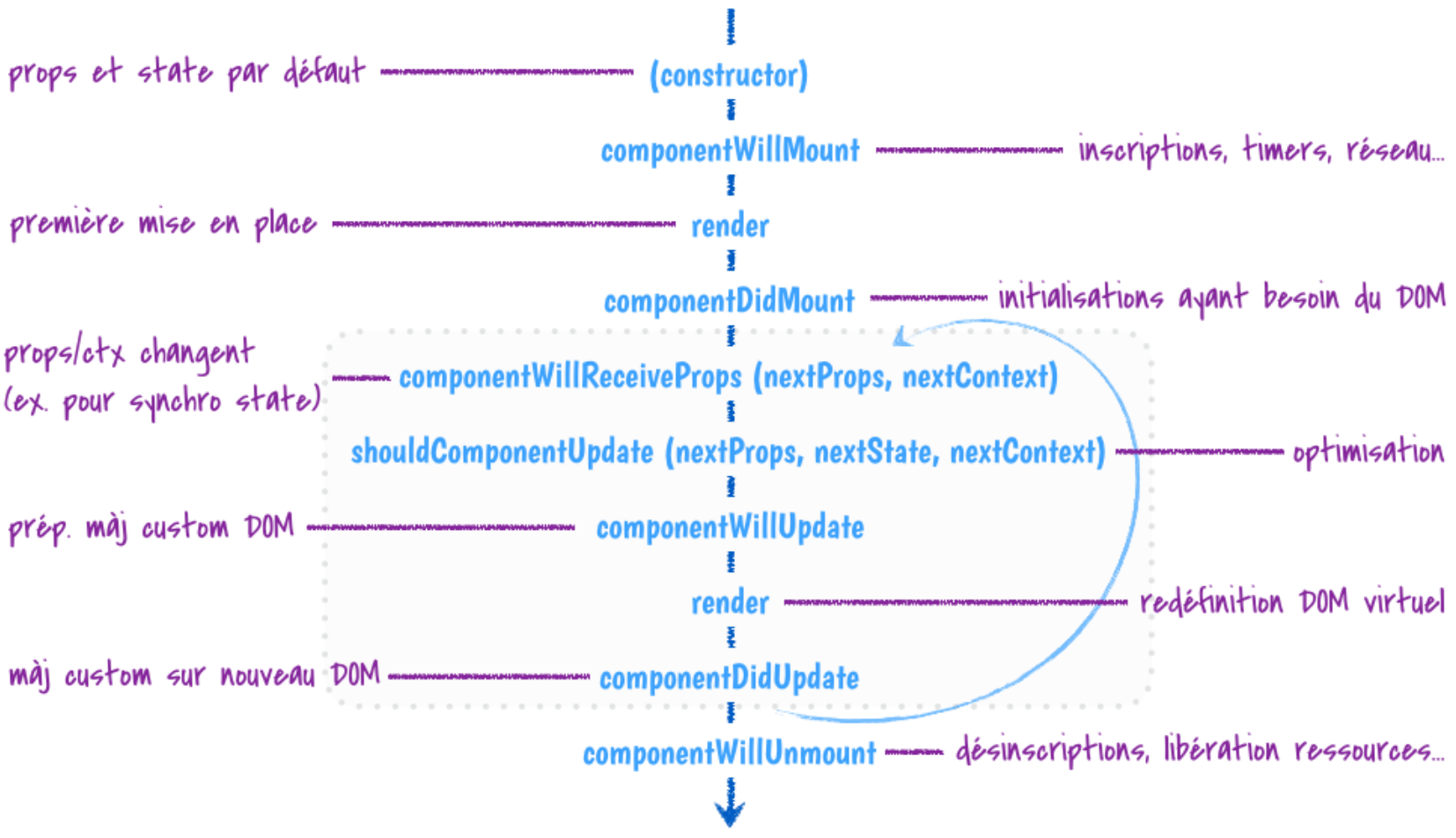
Les composants basés sur des classes passent par toute une série d'étapes au cours de leur vie. React nous permet de réagir à ces étapes en implémentant dans nos classes des méthodes aux noms spécifiques, appelées méthodes de cycle de vie.

En pratique, la majorité de nos composants n'aura besoin que de `render()` et, souvent, du constructeur. Une autre méthode de cycle de vie souvent utile est `componentWillReceiveProps()`. Toutefois, certains composants auront des besoins plus précis, en particulier ceux qui devront interagir avec le DOM brut du navigateur (par exemple pour enrober des

bibliothèques qui travaillent avec le DOM, comme D3, OpenStreetMaps, HighCharts…) ou avec les API Web (timers, réseau, bases de données…).

Les étapes du cycle de vie

Une partie des étapes n'a lieu qu'une fois par composant (tout ce qui va jusqu'au `render()` initial, puis lors du retrait du composant du DOM virtuel). Dans le diagramme ci-dessous, la boîte grise en tirets contient la partie qui, elle, se déclenche chaque fois que le composant parent refait un `render()`, ce qui sollicite à nouveau notre composant existant pour qu'il se mette à jour.



Les étapes du cycle de vie d'un composant défini par classe

La construction

Le constructeur est invoqué en premier, comme pour n'importe quel objet, lorsque le composant apparaît pour la première fois dans un DOM virtuel… Il reçoit les *props* initiales en argument. C'est le bon endroit pour initialiser l'état local et, éventuellement, garantir le `this` de certaines méthodes à l'aide d'un `.bind()` manuel.

Arrivée dans le DOM, départ du DOM

Juste avant que le composant fasse son premier `render()`, `componentWillMount()` se déclenche. C'est l'endroit adapté pour inscrire des timers (ex. `setInterval()`), ouvrir des connexion réseau ou bases de données, etc.

Dès que le composant a été retranscrit pour la première fois dans le DOM réel, `componentDidMount()` survient, et constitue le bon endroit pour aller chercher des références directes à certains éléments du DOM que l'on peut alors manipuler via des bibliothèques tierce-partie.

À l'inverse, juste avant que le composant ne quitte complètement le DOM, `componentWillUnmount()` se déclenche. C'est le moment idéal pour annuler / inverser les initialisations faites dans `componentWillMount()` ou `componentDidMount()` : annuler des timers, fermer des connexions, désactiver des enrobages DOM par des bibliothèques tierces, etc.

Cycle de mise à jour

Chaque fois que le composant parent refait son `render()`, dans le cas où nous y figurions déjà et y figurons toujours, notre composant passe par une série d'étapes de mise à jour.

La première étape est `componentWillReceiveProps()`. Elle prend *a minima* les futures *props* en argument (celles fournies par le dernier `render()` en date de notre composant parent). À ce stade, `this.props` contient encore nos *props* précédentes. Cette méthode est, après `render()`, la plus fréquemment utilisée car elle constitue l'endroit idéal pour synchroniser notre état local avec les *props* ayant pu servir à l'initialiser dans le constructeur, en utilisant `this.setState()`. Par exemple :

```
class UpdateGoalForm extends Component {

  constructor(props) {

    super(props)

    this.state = { goal: props.goal }

  }
```

```
  componentWillReceiveProps({ goal }) {
    this.setState({ goal })
  }

  // …
}
```

On arrive ensuite sur une optimisation fort utile : `shouldComponentUpdate()`. C'est une méthode censurante : si elle renvoie `false`, tout le reste de la mise à jour est laissé de côté. Par défaut, elle renvoie `true`, mais on peut l'implémenter pour décider quand refaire notre propre `render()` ou non.

Imaginons par exemple que notre `render()` lui-même ne « bouge pas » dès lors que `this.props.goal.name` et `this.state.isExpanded` ne changent pas ; on pourrait optimiser pour ça de la façon suivante :

```
// 1er argument : futures props.  2e : futur état local.
shouldComponentUpdate({ goal: { name } }, { isExpanded }) {
  return name !== this.props.goal.name || isExpanded !== this.state.isExpan
}
```

Enfin, si `shouldComponentUpdate()` n'a pas censuré la suite en renvoyant `false`, on s'apprête à faire un nouveau `render()`. Si notre composant a récupéré des références DOM directes pour faire son travail, celles-ci vont peut-être devenir obsolètes. C'est la raison pour laquelle les `render()` ultérieurs à l'initial sont encadrés par `componentWillUpdate()` et `componentDidUpdate()` : elles nous permettent de « faire le ménage » dans nos traitements DOM directs en amont du changement, et de « raccrocher les wagons » en aval, une fois le nouveau DOM à jour.

Notre jeu de Memory n'a pas besoin de ces méthodes de cycle de vie, il est

suffisamment simple pour s'en passer. En revanche, pour aller au bout de notre application, il va nous falloir apprendre à utiliser les formulaires ! C'est le sujet développé dans la prochaine partie.

Simplifiez votre gestion des valeurs et événements

Lorsque l’on travaille avec des champs de formulaires, on repose en pratique sur les définitions HTML et DOM de ces champs ; et c’est bien là le problème, car cet aspect des choses a évolué de façon si désorganisée au fil des ans, qu’on se trouve confronté aujourd’hui à un sacré sac de nœuds lorsque l’on manipule des champs de formulaires « en direct ». Heureusement, React ramène un peu de rationalité à tout cela.

Valeur(s) d’un champ

En HTML, la valeur réelle d’un champ dépend largement du type de balise

utilisée. Elle peut par exemple être définie par l'attribut `value=` , les attributs `checked=` , `multiple=` et `selected=` , ou encore par le contenu de l'élément (pour `<textarea>`). Côté DOM, c'est la même pagaille, avec les propriétés `value` , `selectedIndex` et `options[…].selected` , `checked` et j'en passe ! Il n'est pas étonnant que très vite, on préfère passer par des enrobages masquant toutes ces incohérences, comme par exemple la méthode `.val()` des objets jQuery.

Avec React, on utilise simplement la *prop* `value=` , quel que soit le type de composant. React procède aux ajustements nécessaires selon la balise exploitée. S'il s'agit de valeurs multiples, comme par exemple pour un `<select multiple>` et ses enfants `<option>` , vous passerez tout simplement un tableau de valeurs à la *prop*.

Notez toutefois que si vous souhaitez simplement utiliser une valeur par défaut, sans contraindre ensuite la saisie de l'utilisateur, vous utiliserez plutôt la prop `defaultValue=` . Nous y reviendrons dans les prochains chapitres avec la notion de champs contrôlés ou non.

Détection de changement d'un champ

C'est encore plus difficile de détecter correctement les changements de valeur d'un champ. L'événement « natif » `change` n'est pas disponible partout (on préfèrerait `click` sur les cases à cocher et boutons radios, par exemple), et ne se déclenche qu'à la perte de focus (`focusout` ou `blur`), sauf pour les `<select>` …

Pour détecter correctement, et immédiatement, toute forme de changement possible d'une valeur de champ, il faudrait gérer intelligemment toute une série d'événements natifs dont `change` , `keydown` , `keyup` , `click` , `input` , `focus` , `blur` et `selectionchange` …

React nous simplifie là aussi considérablement la vie, en normalisant le comportement de changement *via* la *prop* `onChange` , qui est certifiée comme étant immédiate (*« live »*), quel que soit le type de champ.

Avant de pouvoir utiliser un formulaire dans notre jeu de Memory, pour saisir le nom du joueur ou de la joueuse en cas de victoire, il nous faut découvrir la notion de champ contrôlé : c’est le sujet du prochain chapitre.

Validez et formatez à la volée avec les champs contrôlés

React propose deux façons de manipuler des champs de formulaires : l'approche **contrôlée**, utilisée la plus souvent, et l'approche non-contrôlée, moins fréquente.

Dans ce chapitre, nous verrons en détail la version contrôlée. Le chapitre suivant présentera les champs non-contrôlés.

Pourquoi les champs contrôlés ?

Il y a plusieurs raisons de vouloir recourir à des champs contrôlés.

Lorsque nous proposons un formulaire, il est fréquent d'avoir besoin, au moment de sa validation, d'accéder aux valeurs des champs, ne serait-ce que pour vérifier leur contenu ou effectuer une requête API en lieu et place de l'envoi classique du formulaire, qui rafraîchirait toute la page.

De plus, nous voulons souvent contrôler le plus tôt possible la validité de la valeur saisie dans les champs : empêcher des adresses e-mail invalides, des montants trop bas, des dates hors d'un intervalle défini, des formats incorrects, etc. Il faut donc pouvoir intervenir au fil de la frappe.

Au-delà du simple contrôle, il peut être avantageux, pour l'expérience utilisateur (UX), de formater à la volée les données saisies lorsque la nature du champ s'y prête (numéro de téléphone, de carte bancaire, de SIREN / SIRET / TVA, de sécurité sociale, IBAN / RIB, etc.). Ceci permet à l'utilisateur de mieux vérifier ses saisies par contrôle visuel.

Le recours aux champs contrôlés nous permet d'intervenir au fil de la frappe pour valider et retravailler la saisie, ce qui répond à toutes les problématiques courantes.

En pratique, les valeurs des champs sont stockées dans l'état local de notre composant, ce qui présente deux avantages :

- Il est facile d'y accéder au moment de l'interception de l'envoi pour les exploiter comme bon nous semble : inutile alors d'aller les « repêcher » dans le DOM.

- Le stockage étant le résultat de nos contrôles et formatages, il est facile de connecter les valeurs exploitées par les champs sur l'état local, empêchant ainsi toute saisie incorrecte, ne serait-ce qu'un court instant.

Comment savoir si un champ est contrôlé ?

Un champ contrôlé est doté d'une *prop* `value=` . De plus, il est généralement doté d'une *prop* `onChange=` qui amène vers un gestionnaire assurant le contrôle et le formatage de la saisie, puis son stockage dans

l'état local. Sans ce gestionnaire, le champ serait, *de facto*, en lecture seule pour l'utilisateur.

Pour les cases à cocher et boutons radio, le fait de définir `checked=` place également le champ en mode contrôlé, même en l'absence de `value=` .

Deux exemples classiques

Voici deux petits exemples de composants dédiés à des saisies spécifiques. Le code présenté ici se concentre sur le contrôle interne, sans évoquer les props nécessaires à l'utilisation de ces composants par des formulaires.

Numéro de téléphone français (5 × 2 chiffres)

```
class FrenchPhoneField extends Component {

  static defaultProps = {

    name: 'tel',

    placeholder: '0x xx xx xx xx',

    required: false,

}


static propTypes = {

  name: PropTypes.string.isRequired,

  placeholder: PropTypes.string,

  required: PropTypes.bool,

}


constructor(props) {

  super(props)

  this.state = { value: '' }

}
```

```
// This method is declared using an arrow function initializer solely
// to guarantee its binding, as we cannot use decorators just yet.
handleChange = ({ target: { value } }) => {
  value = value
    // Remove all non-digits, turn initial 33 into nothing
    .replace(/\D+/, '')
    .replace(/^330?/, '0')
    // Stick to first 10, ignore later digits
    .slice(0, 10)
    // Add a space after any 2-digit group followed by more digits
    .replace(/(\d{2})(?=\d)/g, '$1 ')
  this.setState({ value })
}

render() {
    const { name, placeholder, required } = this.props
    return (
      <input
        autocomplete="tel"
        name={name}
        onChange={this.handleChange}
        placeholder={placeholder}
        required={required}
        type="tel"
        value={this.state.value}
      />
    )
  }
}
```

Numéro de carte bancaire (4 × 4 chiffres)

Cet exemple est proche du précédent :

```
class CreditCardField extends Component {
  static defaultProps = {
    name: 'cc-number',
    placeholder: 'xxxx xxxx xxxx xxxx',
    required: false,
  }

  static propTypes = {
    name: PropTypes.string.isRequired,
    placeholder: PropTypes.string,
    required: PropTypes.bool,
  }

constructor(props) {
  super(props)
  this.state = { value: '' }
}

// This method is declared using an arrow function initializer solely
// to guarantee its binding, as we cannot use decorators just yet.
handleChange = ({ target: { value } }) => {
  value = value
    // Remove all non-digits
    .replace(/\D+/, '')
    // Stick to first 16, ignore later digits
```

```
    .slice(0, 16)
    // Add a space after any 4-digit group followed by more digits
    .replace(/(\d{4})(?=\d)/g, '$1 ')
  this.setState({ value })
}

render() {
  const { name, placeholder, required } = this.props
  return (
    <input
      autocomplete="cc-number"
      name={name}
      onChange={this.handleChange}
      placeholder={placeholder}
      required={required}
      type="text"
      value={this.state.value}
    />
  )
  }
}
```

Mise en application

Commençons par récupérer quelques styles et squelettes pour le nouveau et dernier composant de notre jeu : la saisie du nom en cas de victoire.

1. Allez sur le dépôt GitHub, à l'étiquette **debut-champs-controles** (suivez le lien)

2. Récupérez la feuille de style `HighScoreInput.css` et le squelette de

`HighScoreInput.js` , ainsi que la version à jour de `HallOfFame.js` (qui fournit une méthode de sauvegarde du tableau d'honneur dans le stockage local de votre profil navigateur) et posez-les dans votre dossier `src/`

Mettre au point la saisie du nom

Afin de ne pas avoir à gagner une partie à chaque fois qu'on recharge la page pour mettre au point le composant, modifions le `render()` de `App.js` comme suit :

```
<HighScoreInput guesses={guesses} />

{won && <HallOfFame entries={FAKE_HOF} />}
```

Pour le moment, le formulaire s'affiche en permanence.

Contrôler le champ

Commençons par doter `<HighScoreInput />` d'un état local pour la valeur de nom saisie :

```
class HighScoreInput extends Component {
  state = { winner: '' }

  render() {
    return (
      // …
      <input autoComplete="given-name" type="text" value={this.state.winner
      // …
    )
  }
}
```

À ce stade, le champ est, *de facto*, en lecture seule. Ajoutons un gestionnaire de saisie qui va le rendre éditable. L'idée ici est de forcer une saisie majuscule, comme sur les vieilles bornes d'arcade :

```
// Arrow fx for binding

handleWinnerUpdate = (event) => {

  this.setState({ winner: event.target.value.toUpperCase() })

}


render() {

  return (

    // …

    <input

      autoComplete="given-name"

      type="text"

      onChange={this.handleWinnerUpdate}

      value={this.state.winner}

    />

    // …

  )

}
```

Quelque soit la façon dont vous saisissez du texte dans le champ (copier-coller, complétion, frappe, etc.), il doit désormais automatiquement être en majuscules.

Intercepter l'envoi du formulaire

Commençons par traiter l'événement `submit` du formulaire, qui sera déclenché quel que soit la manipulation utilisée (touche `Entrée` dans le champ, clic sur le bouton, touche `Espace` sur le bouton, etc.) :

```
render() {
  return (
    <form className="highScoreInput" onSubmit={this.persistWinner}>
    …
  )
}
```

Définissons la méthode métier. Fournie par référence, elle doit donc garantir son `this` pour pouvoir manipuler`state` :

```
// Arrow fx for binding
persistWinner = (event) => {
  event.preventDefault()
  const newEntry = { guesses: this.props.guesses, player: this.state.winner
  saveHOFEntry(newEntry, this.props.onStored)
}
```

La fonction `saveHOFEntry()` est fournie par le `HallOfFame.js` que vous avez récupéré plus tôt. Elle attend une fonction de rappel en second argument, qu'elle appellera avec le tableau d'honneur à jour une fois celui-ci ajusté et persisté dans le navigateur. Ce n'est pas le rôle de la saisie de score de réagir à ça. Nous allons donc déclarer une *prop* `onStored` de type fonction, que `<App />` nous fournira :

```
HighScoreInput.propTypes = {
  guesses: PropTypes.number.isRequired,
  onStored: PropTypes.func.isRequired,
}
```

Afficher intelligemment la saisie et le tableau d'honneur

Justement, jusqu'ici `<App />` a la main lourde sur l'affichage de ces

éléments : non seulement il les affiche tout le temps, ou presque, et avec les mauvaises données. On va donc modifier la partie conditionnelle du JSX en bas de son `render()`, en allant chercher une nouvelle information dans l'état local : `hallOfFame`, qui serait le tableau d'honneur à jour.

Commençons par ajuster l'initialisation de l'état en haut de classe :

```
state = {
  cards: this.generateCards(),
  currentPair: [],
  guesses: 0,
  hallOfFame: null,
  matchedCardIndices: [],
}
```

Ajoutons ensuite une méthode qui va recevoir un tableau d'honneur et qui en ajustera l'état avec :

```
// Arrow fx for binding
displayHallOfFame = (hallOfFame) => {
  this.setState({ hallOfFame })
}
```

Au début du `render()`, allons chercher `hallOfFame`, également présent dans l'état :

```
render() {
  const { cards, guesses, hallOfFame, matchedCardIndices } = this.state
  // …
```

Enfin, revoyons complètement la fin de grappe JSX, en remplaçant le champ de saisie et la condition simple basée sur `won` par ceci :

```
{
  won &&
    (hallOfFame ? (
      <HallOfFame entries={hallOfFame} />
    ) : (
      <HighScoreInput guesses={guesses} onStored={this.displayHallOfFame} /
    ))
}
```

Notez que tout le bloc est assujetti à `won` :

- Sans fin de partie, aucun affichage possible.
- Mais si `won` est à `true`, on regarde si on dispose d'un tableau d'honneur, ce qui signifierait qu'on a déjà procédé à la saisie du nom, auquel cas on affiche le tableau en se basant sur ces données véritables.
- Faute de tableau, on n'a manifestement pas encore saisi le nom : on utilise donc le composant de saisie, en passant notre `displayHallOfFame()` comme fonction de rappel une fois le tableau à jour et persisté.

Pour tester ça rapidement, vous pouvez ajuster temporairement la définition de `won`, pour limiter à quelques paires réussies par exemple :

```
// TEMPORAIRE
const won = matchedCardIndices.length === 4 // cards.length
```

Et voilà notre jeu de Memory terminé ! *(pensez quand même à remettre la définition de `won` normale.)*

Il lui manque toutefois des tests, que nous verrons dans la prochaine partie. Et il nous reste à parler brièvement des champs non contrôlés, ce

que nous ferons dans le prochain chapitre.

Déléguez des traitements avec les champs non contrôlés

Les champs non contrôlés sont pratiques lorsque l’on souhaite récupérer une valeur de champ sans contraindre sa saisie ni son formatage. Cela peut notamment se produire parce que ces champs sont déjà « contrôlés » par un mécanisme externe à React (utilisation d'une bibliothèque tierce partie comme un plugin jQuery, validation par HTML5 et DOM pur, etc.).

Valeur par défaut

Ce n’est pas parce qu’on ne contrôle pas le champ qu’on ne souhaite pas définir sa valeur initiale. Comme on l’a évoqué au premier chapitre de cette

partie, on peut pour cela recourir à la prop `defaultValue=` plutôt que `value=` .

Pour les cases à cocher et les boutons radio, on a dans le même esprit `defaultChecked=` pour l'état coché ou non.

Utiliser une *ref*

Le moyen le plus fiable pour récupérer, dans notre composant, une référence sur le champ dont on veut obtenir la valeur le moment venu, consiste à utiliser une *prop* `ref=` sur le champ, par exemple comme ceci :

```
render() {
  const { defaultNick, name, placeholder, required } = this.props
  return (
    <p>
      <label>
        Surnom :
        <input
          defaultValue={defaultNick}
          name={name}
          placeholder={placeholder}
          ref={(field) => { this.nicknameField = field }}
          required={required}
          type="text"
        />
      </label>
    </p>
  )
}
```

Approfondissez vos connaissances avec les documentations

La documentation officielle de React revient en détail sur les points abordés dans cette partie :

- Forms détaille la gestion des formulaires, notamment pour le mode contrôlé et la simplification des événements et valeurs.
- Uncontrolled Components précise la partie non contrôlée.

Il existe par ailleurs de nombreux éléments dans l'écosystème pour faciliter

la gestion des formulaires et autres champs personnalisés.

- Le fameux référentiel **Awesome React** a naturellement une section entière dédiée aux formulaires, et même une sous-partie spécifique aux saisies auto-complétées.

- De son côté, **Awesome React Components** a une liste complète de composants personnalisés dédiés à la saisie en formulaires.