ORSYS formation

Learning for success

VOTRE SUPPORT DE COURS

FORMATIONS ORSYS

SÉMINAIRES - COURS DE SYNTHÈSE - STAGES PRATIQUES - CERTIFICATIONS

PARCOURS CERTIFIANTS - FORMATIONS À DISTANCE

E-LEARNING - COACHING

www.orsys.com - +33 (0)1 49 07 73 73 - info@orsys.fr

ORSYS formation

Formation ReactJS.

Maîtriser le Framework JavaScript de Facebook.

Prenant à contrepied les modèles traditionnels, le Framework maintenu par Facebook favorise la simplicité et la performance des composants de RIA.

Vous apprendrez dans ce cours à développer des applications avec ReactJS, JSX et Flux/Redux et découvrirez le principe et les bénéfices du développement isomorphique.

Objectifs pédagogiques

- Développer avec ReactJS.
- Concevoir une SPA avec ReactJS et Flux.
- Comprendre le subset JavaScript JSX.
- Optimiser les performances des RIA.

Présentation

Participants. (Tour de table)

Développeurs JavaScript, architectes et chefs de projets web.

Prérequis.

Bonne connaissance de JavaScript, pratique du développement web.

Récapitulatif matinal.

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

Concertation personnelle.

Votre formateur passera vous assister individuellement aussi souvent que possible.
N'hésitez pas à le solliciter pendant la journée de formation ou pour revenir sur un point particulier en fin de journée.

Votre Formateur.

desorbaix.formation@free.fr

+33 6.64.27.63.60

Version numérique

Pour votre confort de lecture et de manipulation ce support vous est également distribué en version numérique. (**voir partage**)

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

et j'en profite pour remercier celui qui m'a fort aidé pour ce support.Mr Renaud DUIBUIS formateur en technologies web chez Orsys.

Autres références

- reactjs officiel
- codementor.io
- Playground
- Playground JSX
- Playground No JSX
- node.js
- npmjs
- Webpack

Introduction

ReactJS

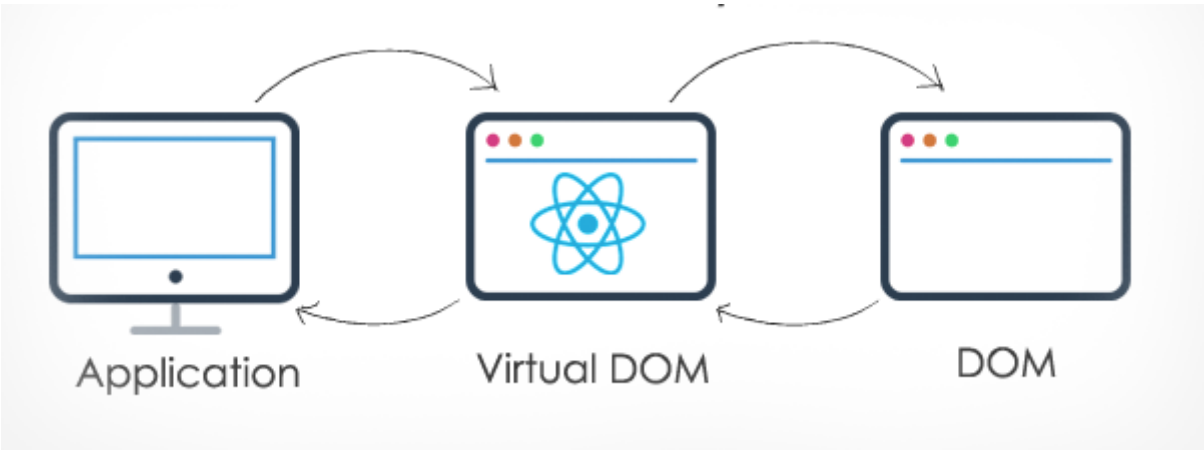
React (appelé aussi React.js) est un moteur de rendu JavaScript qui se démarque par une architecture voulue **efficace et performante.**

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

La logique *best of breed*

React cherche à offrir la meilleure réponse technologique à une problématique précise. **Rendre le DOM rapidement**

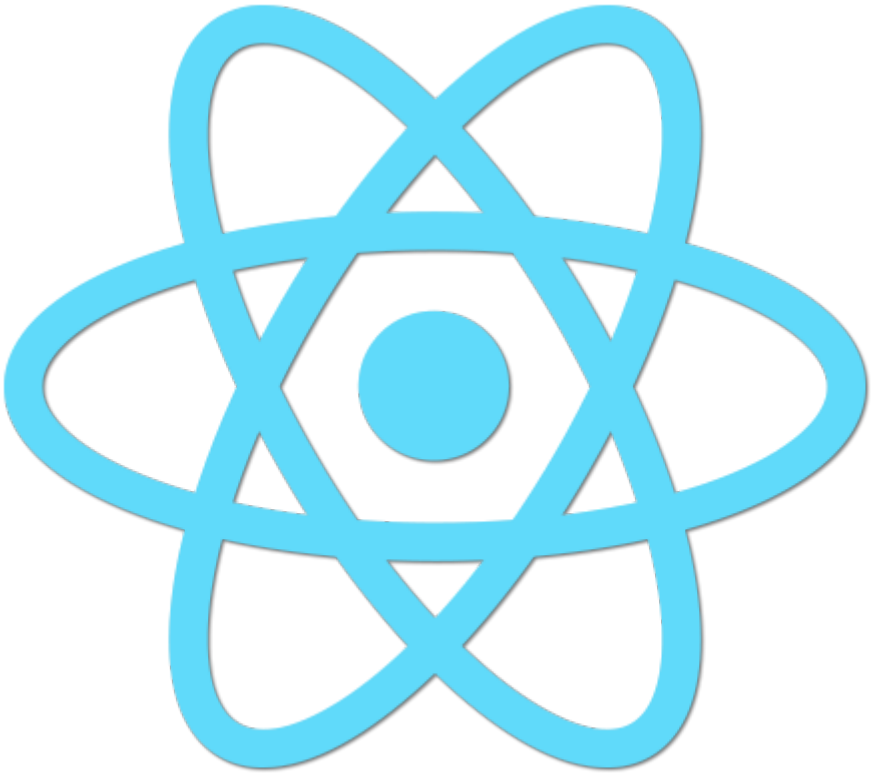
React agit comme un **moteur de rendu** intermédiaire. Il s'agit d'**une librairie JavaScript** et non d'un framework. En termes de performance, **React optimise les opérations** sur le DOM en utilisant un **DOM virtuel.**



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

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

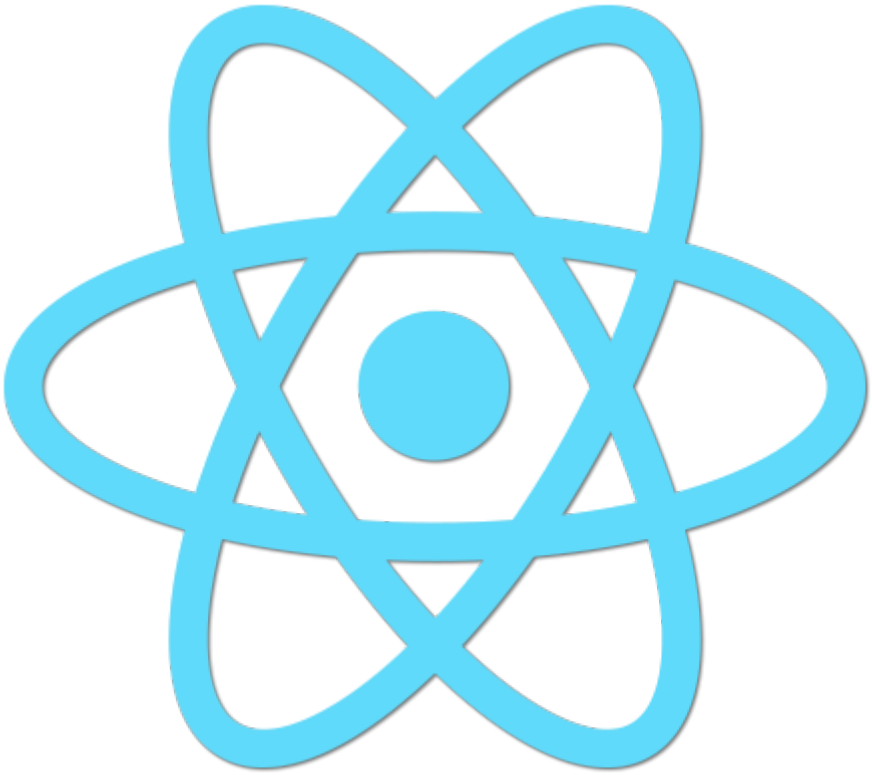
> En tant que **Librairie JavaScript ReactJS** statisfait aux problématiques de développement en utilisant `l'écosystème industrialisé moderne`



Sommaire

Table des matières.

[TOC]



Rappels des composants des RIA.

Rappels des composants des RIA

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

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

Support des fonctionnalités HTML5

> **Référence HTML**

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

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

> **Référence CSS**

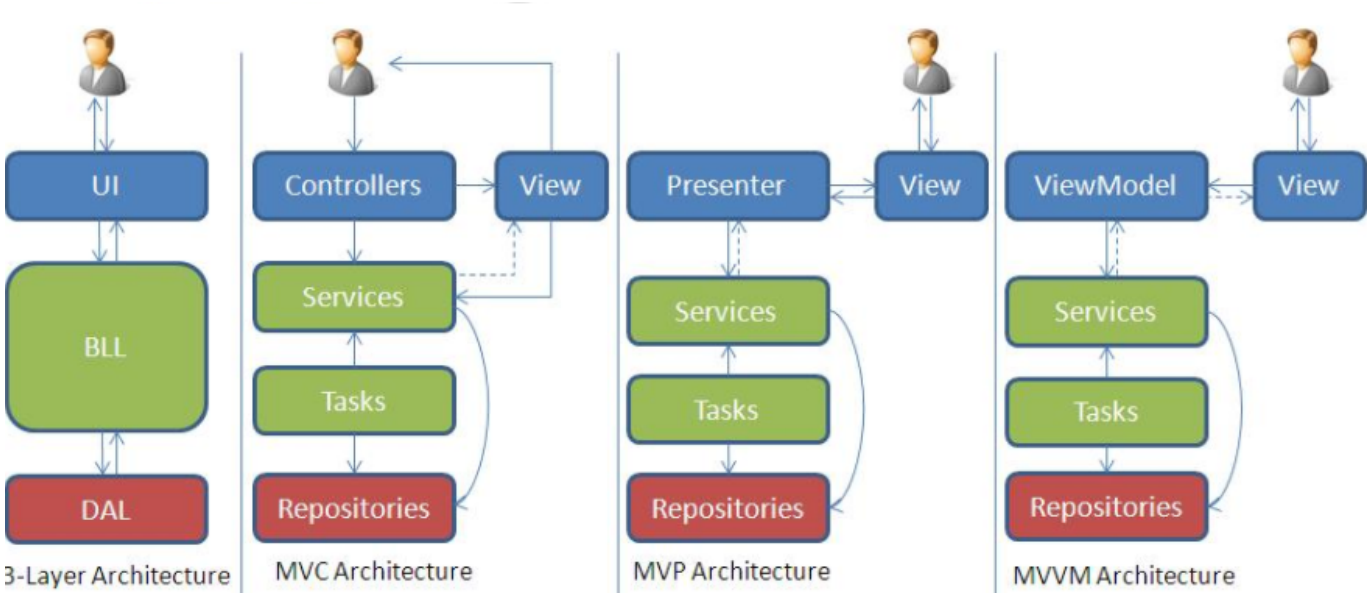
Le standard pour JavaScript est ECMAScript. En 2012, tous les navigateurs modernes supportent complètement ECMAScript 5.1.

Tables de compaitibilité JavaScript

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

> **Référence JavaScript**
>
> JavaScript (qui est souvent abrégé en "JS") est un langage de script léger, orienté objet, principalement connu comme le langage de script des pages web.
>
> Il est aussi utilisé dans de nombreux environnements extérieurs aux navigateurs web tels que node.js ou Apache CouchDB.
>
> C'est un langage à objets utilisant le concept de prototype, disposant d'un typage faible et dynamique qui permet de programmer suivant plusieurs paradigmes de programmation : **fonctionnelle, impérative et orientée objet. Apprenez-en plus sur JavaScript.**

Design patterns applicatifs classiques. Limitations des applications JavaScript.



Architecture 3 tiers

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

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

> C'est une extension du modèle client-serveur.

Architecture MVC (Modèle-Vue-Contrôleur)

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

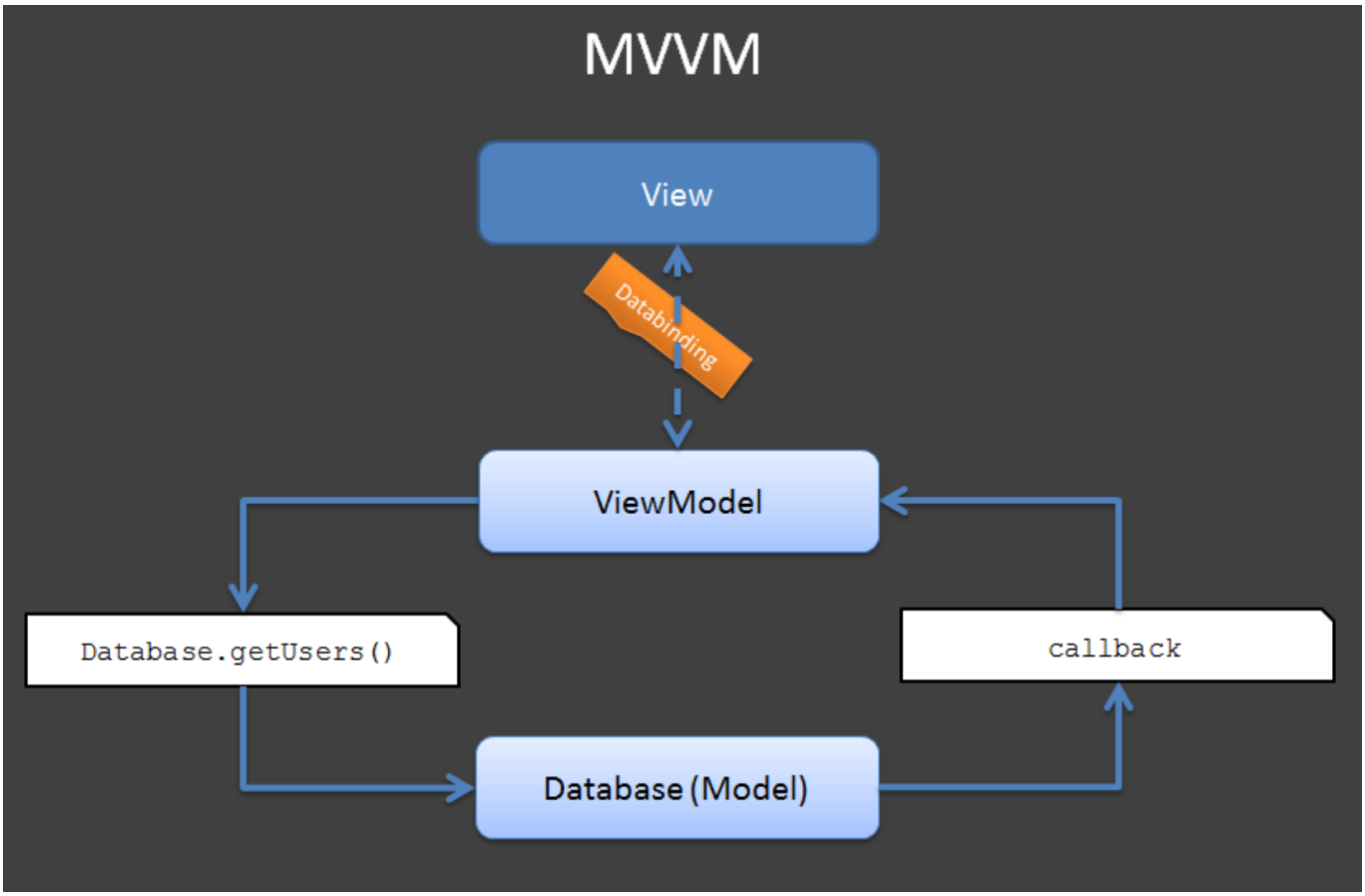
- **Modèle :** Encapsulation lds données ainsi que de la logique relative : validation, lecture et enregistrement.
- **Vue :** Partie visible d'une interface graphique.
- **Contrôleur :** Module de traitement des actions utilisateur, modifiant les données du modèle et appelant le rendu de la vue.

Architecture MVP (Model View Presenter)

Le modèle-vue-présentation est considéré comme un **dérivé de l'architecture modèle-vue-contrôleur.** Il garde les mêmes principes que MVC en éliminant les interactions entre la vue et le modèle qui seront effectuées par le **biais de la présentation, organisant les données à afficher dans la vue.**

Architecture MVVM (Model View ViewModel)

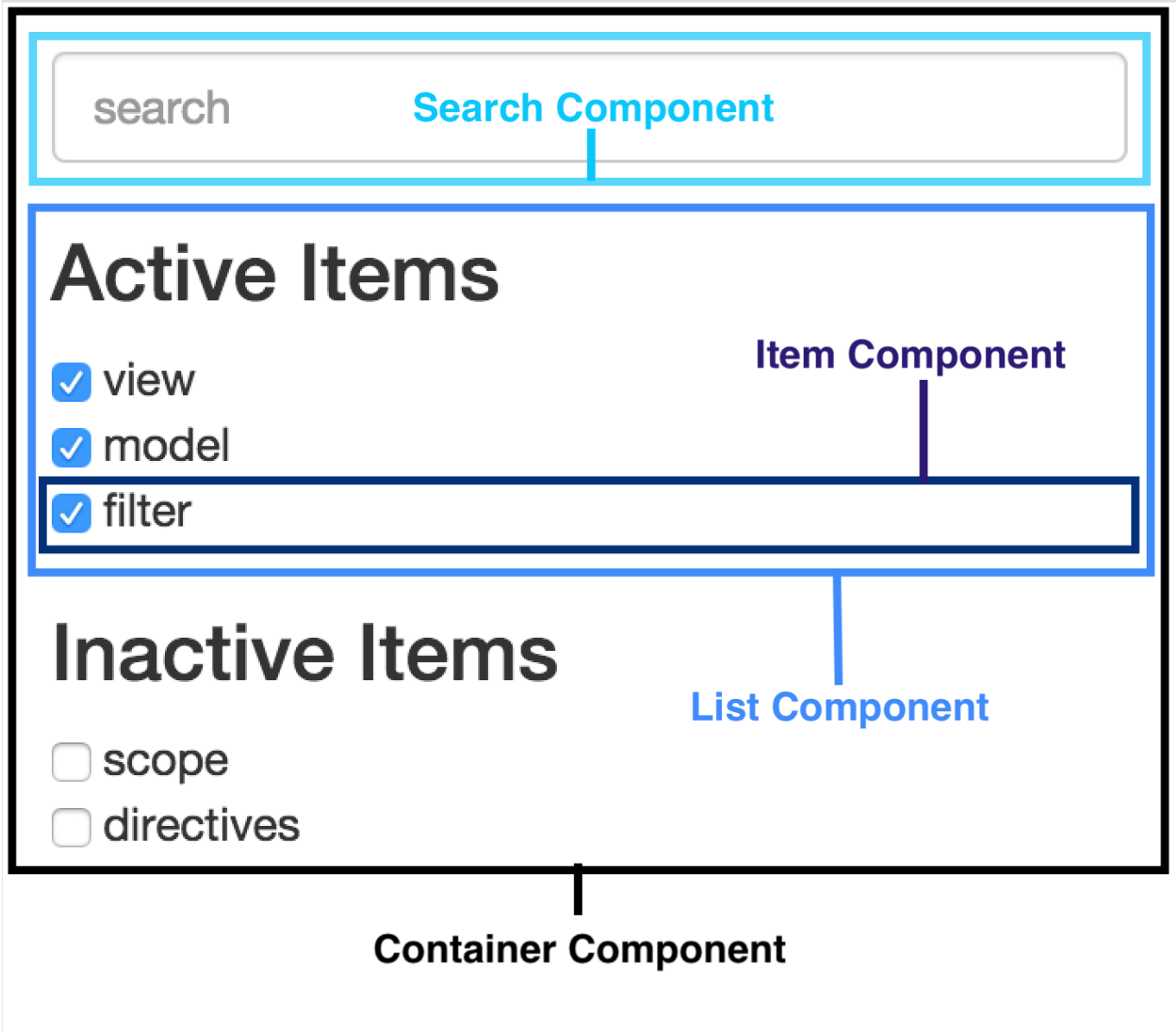
Le modèle-vue-vue modèle est une architecture et une méthode de conception.



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

Component-based Architecture

La conception basée sur les composants vise la séparation des préoccupations par rapport aux fonctionnalités de l'application. **Il s'agit d'une approche basée sur la réutilisation** pour définir, implémenter et composer des composants indépendants à couplage libre dans les systèmes.



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

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

A propos des Web Components

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

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

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

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

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

Une démonstration minimaliste dans chrome

Cet exemple utilisae la syntaxe ES6

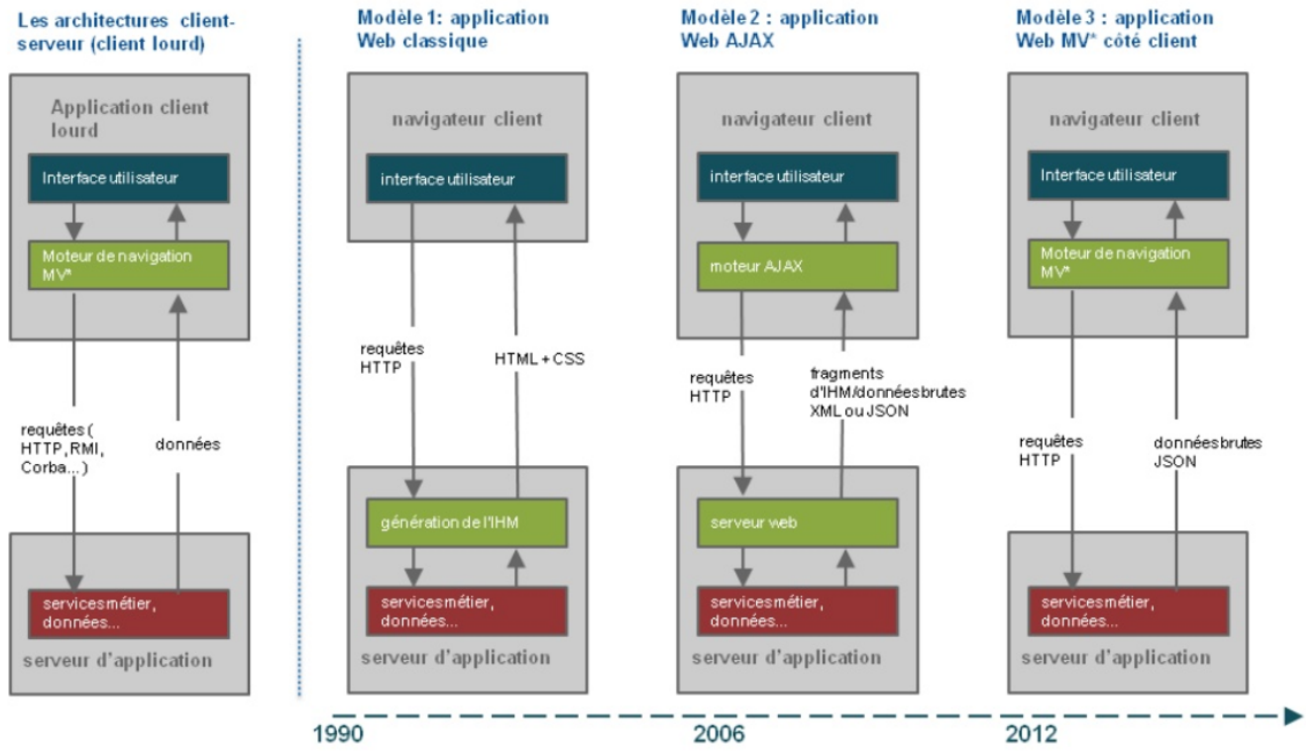
```
<body>
    <simple-increment data-step="5"></simple-increment>
    <script>
    class SimpleIncrement extends HTMLElement {
        constructor(args) {
            super();
        }

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

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

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

        attributeChangedCallback(){} // 3*  Called when one of attributes
of the element is changed.

        detachedCallback(){} // 4 Called when the element is detached from
the document.

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

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

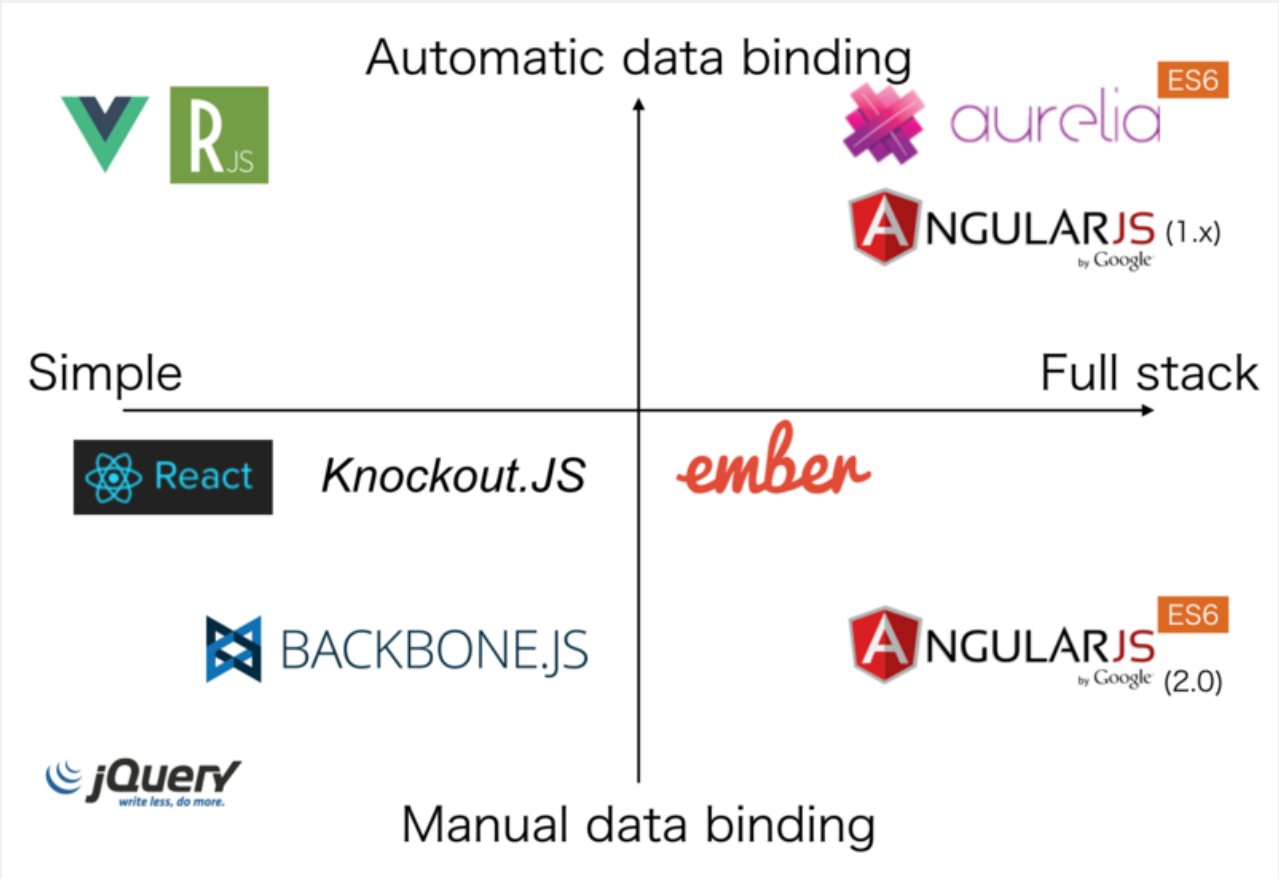
- Un modèle générique `HTMLElement`.
- Un bloc logique `class` .
- Un sytème de rendu `document.registerElement` ici le DOM.
- Un cycle de vie exposé par le système de rendu.

Ecosystème des frameworks JavaScript.

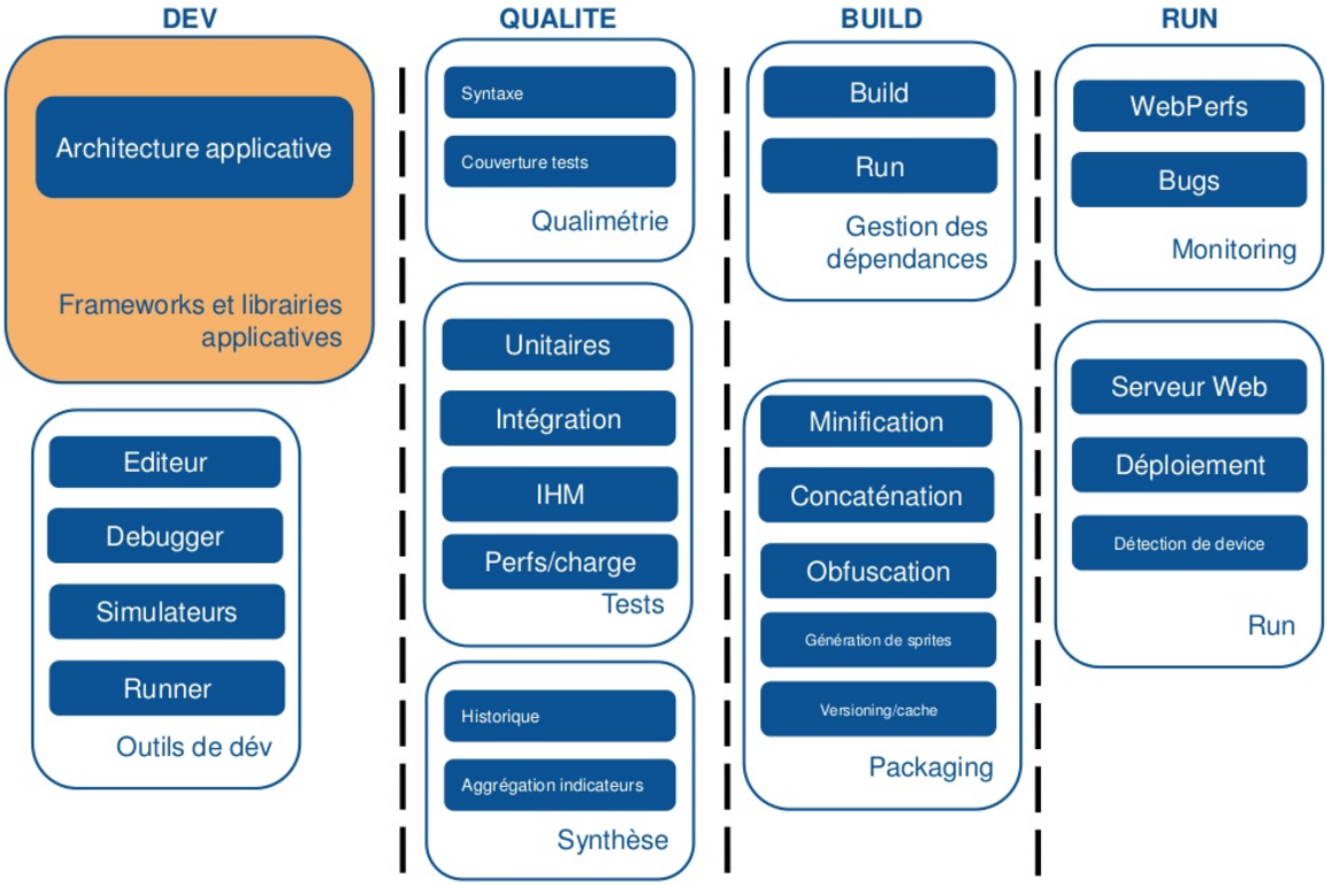
Evolution des développements Front End.



L'actuel "marché" des Frameworks UI/MVC JavaScript est très riche.

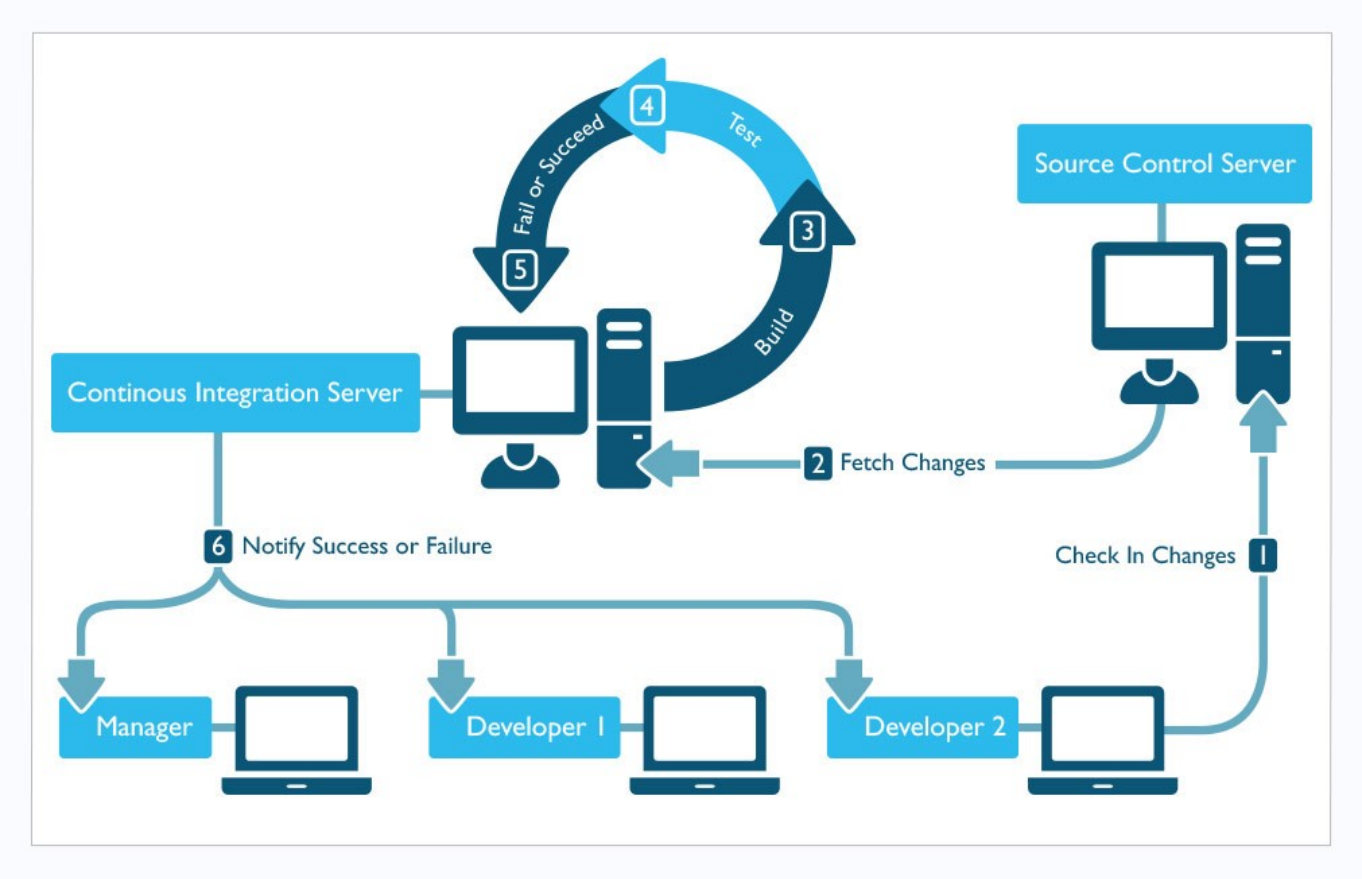
> Chaque solution représente un choix d'implémentaion particulier.
>
> 
>

Un grand nombre d'utilitaires permettent d'automatiser les tâches découlant de l'évolution de ces pratiques de développement.Saviez-vous que JavaScript utilise portée



Industrialisation de la production.

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



Outils indispensables pour le développeur Front End.

Il est possible de classer les outils selon trois catégories :

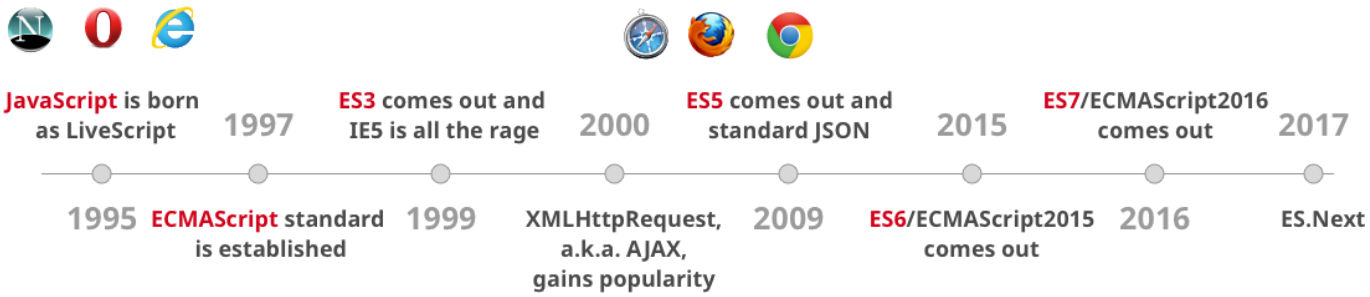
- **Logique** dépendances tiers ,telles que les différents frameworks.
- **Technique** ayant un impact sur la plate-forme de développement pour l'automatisation et le contrôle des tâches par exemple.
- **Productivité** toute autre solution non indispensable permettant d'accélérer le développement, allant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

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

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

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

Prendre en considération l'évolution du langage

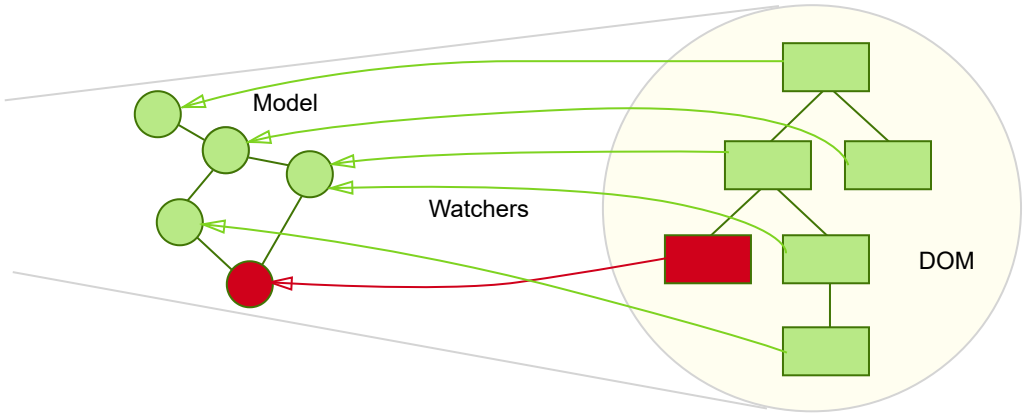


> **Par anticipation** et les autres frameworks JavaScript se rapprochent de la `future pattern de développement` qu'apporteront les **Web Components**

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

Dirty-checking (vérifications massives)

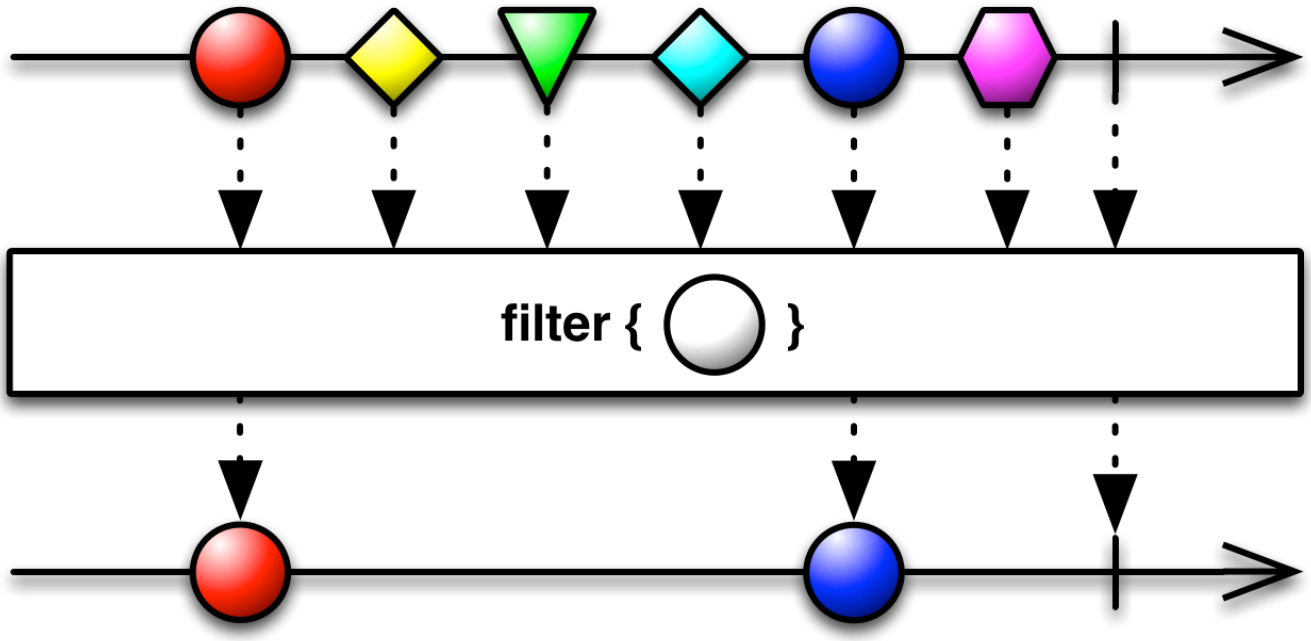
L'idée de base du dirty-checking est que dès qu'une modification des données peut avoir eu lieu, la bibliothèque va examiner l'intégralité du modèle de données pour déterminer s'il a changé, en utilisant un cycle de vie basé sur des sommes de contrôle ou sur les valeurs brutes.



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

Observable

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



Pour mieux saisir le concept il est possible d'utiliser les graphiques interactifs

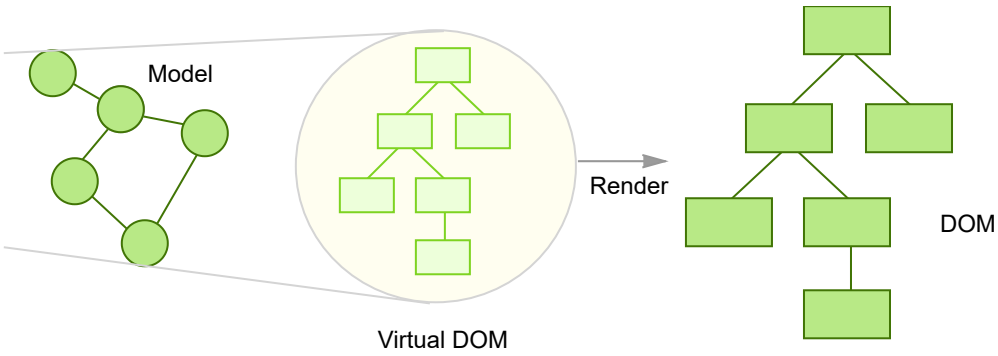
> Les Observables sont un des concepts clés de la librairie RxJS

Virtual-Dom

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

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

Le moteur de rendu conserve au moins deux versions de l'arborescence, celle qui correspond au DOM réel et la nouvelle version qu'on veut afficher.
Puis il calcule la différence entre les deux versions et interagit lui-même avec le DOM pour trouver la façon minimale de le modifier pour obtenir la page souhaitée.



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

ReactJS, positionnement et philosophie.

ReactJS est une bibliothèque (library) à l'opposé de la philosophie des frameworks plus larges de type AngularJS et EmberJS. Quand on sélectionne ReactJS on est libre de choisir la meilleure library complémentaire moderne pour régler telle ou telle difficulté.

Le Javascript avance rapidement, et React permet de passer d'un petit bout de code à une application embarquant des bibliothèques plus importantes en un clin d'oeil.

Identifier rapidement les erreurs

Une erreur de syntaxe dans ReactJS ne compilera pas. Cela signifie que vous êtes en mesure d'identifier immédiatement quelle est la ligne comportant une erreur. Le compilateur JSX nous indique immédiatement quand un élément n'est pas fermé par exemple ou quand une propriété est mal utilisée ou inexistante dans le contexte de la page.

> Cette approche fluidifie considérablement les phases de développement.

JavaScript Centric

> ReactJS met HTML dans le JavScript.

Cela impacte fondamentalement l'expérience de développement. **Javascript est bien plus puissant que le HTML.** Il semble donc plus logique de faire supporter le markup HTML par Javascript que le contraire. Le HTML et le javascript ont besoin de travailler ensemble, ils sont liés.

Conception Simplifiée

Puisque Javascript supporte de base les boucles, **le JSX de ReactJS réutilise toute la puissance de Javascript** directement telles que les maps, les filtres etc...

```
<ul>
  { heroes.map(hero => <li key={hero.id}>{hero.name}</li>)}
</ul>
```

La syntaxe de ReactJS est donc une syntaxe conceptuellement assez simple en comparaison aux autres frameworks qui proposent des spécificités.

- Ember: `{{# each}}`
- Angular 1: `ng-repeat`
- Angular 2: `ngFor`
- Knockout: `data-bind=”foreach”`
- React: `Just Use JavaSscript ☺ !`

Expérience/Coût de développement

> Le support de l’autocomplétion pour le JSX, la validation en temps réel et la richesse des messages d’erreurs permet d’accélérer considérablement le développement.

- Angular 2 : 764k minified
- Ember : 435k
- Angular 1: 143k
- React + Redux : 151k minified

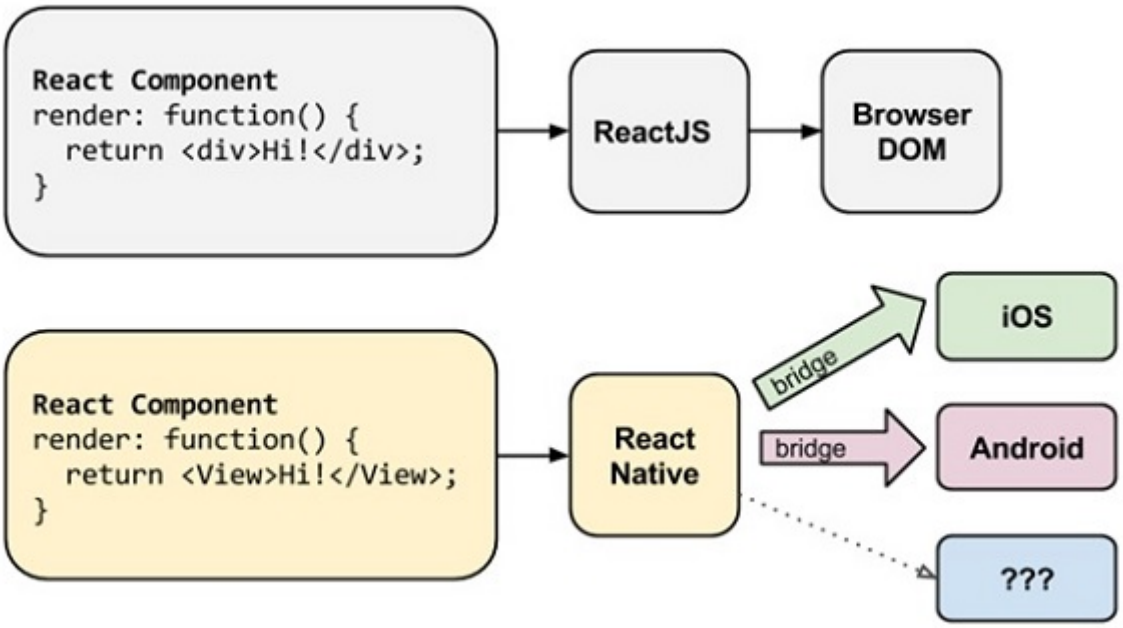
JSX, présentation. Mise en oeuvre "Transpilers".

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

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

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

> La syntaxe JSX induit le besoin de transformation du code.



Environnement de développement. IDE et plug-ins.

Outils de développement les autres logiciels :

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

> Nous utiliserons VS Code.

cf. pratique

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

Vérifier des installations :

cf. pratique

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

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

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

$> npm --version
x.x.x

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

** √ Installation réussie !**

Installer des modules tiers

> Il existe deux modes d'installation avec npm :

global (machine)

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

local (dossier courant)

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

Installation des modules

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

Initialisation

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

> **npm init**

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

La commande démarre une série de questions interactives. Certaines réponses seront pré-remplies, par exemple si un dépôt Git ou un fichier README sont détectés.

À l'issue de la série de questions, le fichier `package.json` sera créé dans le répertoire courant. Ensuite libre à vous de le compléter avec d'autres éléments optionnels de configuration.

Configuration

Que votre projet soit public ou non, il est important de renseigner les champs décrits ci-après. Ils indiquent aux utilisateurs les intentions du projet ainsi que l'emplacement des ressources..

- *name* : il s'agit de l'identifiant du module lorsqu'il est chargé via la fonction `require()`. Ce sera également l'identifiant npm si vous publiez ce module dans un registre public ou privé. Par exemple, si la propriété *name* vaut *nodebook*, le module se chargera via `require('nodebook')` et s'installera avec la commande `npm install nodebook` ;
- *description* : une indication textuelle des objectifs et fonctionnalités du module, écrite généralement en anglais ;
- *version* : chaîne respectant la sémantique *semver* — par exemple `1.0.0`. Nous verrons un peu plus loin dans ce chapitre comment utiliser intelligemment cette valeur pour assurer des mises à jour tout en préservant la compatibilité descendante au sein des projets dépendant de ce module ;
- *main* : emplacement du fichier Node chargé par défaut lors d'un appel à `require(<name>)`. S'il n'est pas spécifié, Node tentera de charger par défaut le fichier `index.js` ;
- *repository* : objet spécifiant le type de dépôt de code ainsi que son URL. Présent essentiellement à titre informatif ;
- *preferGlobal* : booléen indiquant si ce module a davantage vocation à être installé globalement au niveau du système ou non (`false` par défaut) ;
- *bin* : emplacement du fichier. npm effectue un lien symbolique pour rendre `<name>` disponible en tant qu'exécutable système lors d'une installation globale ;
- *private* : boolean spécifiant que le module ne doit pas être publié dans un registre npm (`false` par défaut) ;
- *dependencies* : objet représentant respectivement en clé/valeur les noms/versions des modules dont le projet dépend ;
- *engines* : objet spécifiant des contraintes de compatibilité suivant la sémantique *semver* dans lesquelles le projet s'exécute sans accroc.

Dépendances

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

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

npm en résumé

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

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

Structure de projet

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

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

La suggestion d'organisation ci-avant s'explique de la manière suivante :

- *bin* : fichiers exécutables depuis un shell ;
- *config* : environnements de configuration pour éviter d'écrire ces valeurs en dur dans le code source ;
- *data* : données diverses (type binaires ou CSV) nécessaires au fonctionnement de l'application ;
- *dist* : artéfacts produits après une compilation ou un résultat de *build* — souvent une bibliothèque Node prête à l'usage pour le navigateur ou une arborescence d'application prête à être déployée ;
- *doc* : fichiers de documentation relatifs à la version courante de l'application ;
- *lib* : bibliothèque et modèles utilisées par l'application. Ce code peut typiquement grossir suffisamment pour ainsi justifier qu'il soit extrait en tant que projet(s) indépendant(s) ;
- *node_modules* : modules tiers installés automatiquement via la commande npm. Autrement dit, ne créez jamais de fichiers dans ce répertoire autrement que par la commande npm ;
- *src* : code source spécifique au projet, des routes aux vues/templates en passant par les images et le code à compiler (Sass, LESS, JSX etc.) ;
- *tests* : tests unitaires, fonctionnels et *fixtures* nécessaires à leur fonctionnement ;
- *package.json* : fichier de configuration précédemment décrit dans cet ouvrage ;
- *README* : présentation, description et documentation minimaliste — mais suffisamment pour installer, faire fonctionner et contribuer au projet.

Ajout de dépendances

Le répertoire *node_modules* contient les dépendances requises par la fonction `require()` (Le mécanisme principal d'installation est la commande `npm install`.

L'installation d'un module est par défaut *locale* au projet. Mais elle peut également être globale au système — nous le verrons plus tard.

> Il est toutefois recommandé d'installer les modules localement, afin de limiter leur portée uniquement au projet tout en maintenant une dépendance explicite et gérable via le fichier *package.json*.

```
npm install --save async yargs
```

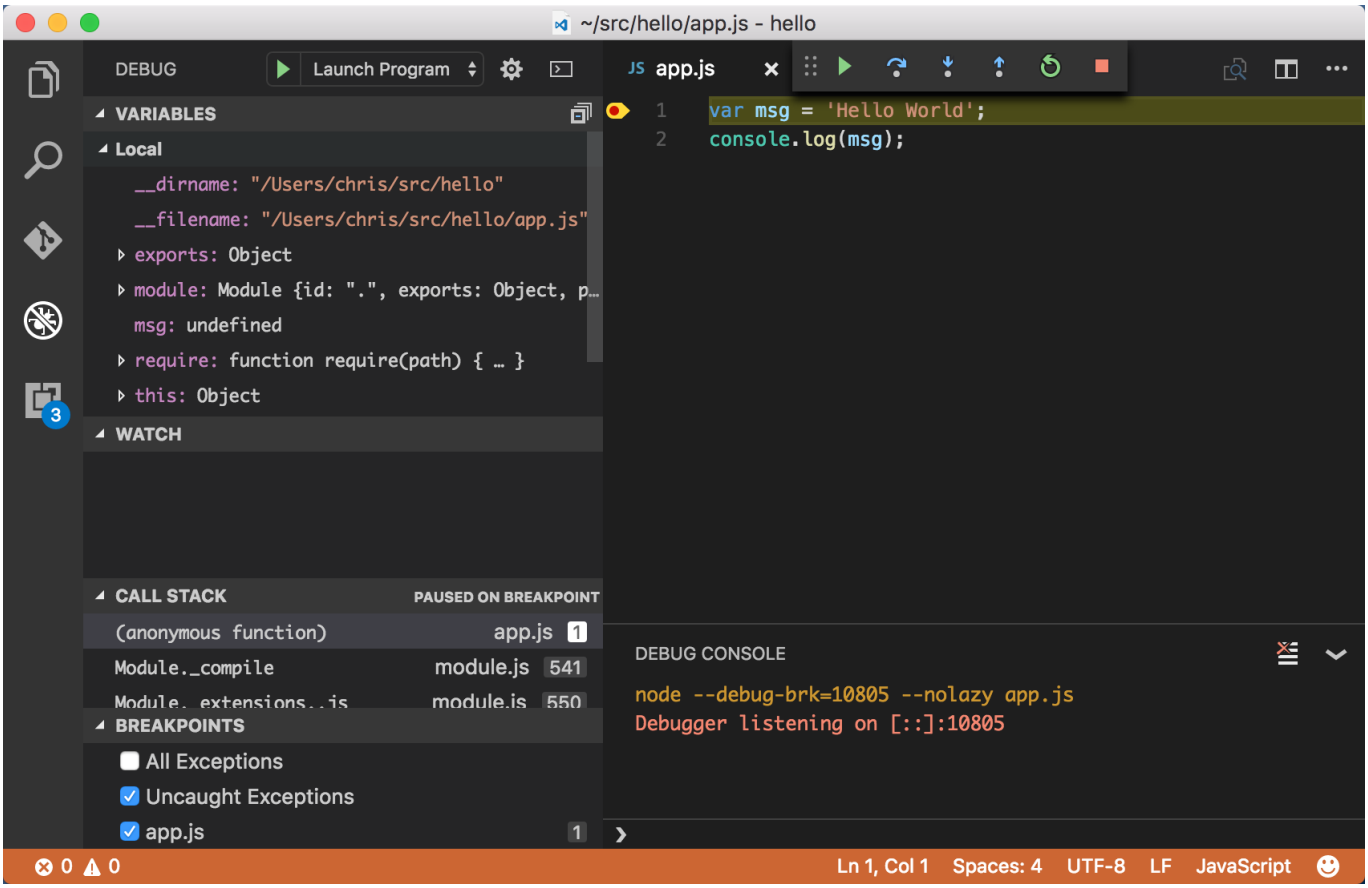
La commande précédente effectue plusieurs opérations :

1. requête du registre *npmjs.com* à propos des deux modules *async* et *yargs* ;
2. si les modules existent, la version compatible la plus récente est retournée (équivalent à `npm view async version` et `npm view async version` — respectivement `0.9.0` et `1.3.1`) ;
3. téléchargement et décompression des paquets dans les répertoire `node_modules/async` et `node_modules/yargs` ;
4. introspection récursive des dépendances de ces modules et si besoin est, téléchargement et décompression dans leur répertoire *node_modules* respectif (ici `node_modules/async/node_modules` et `node_modules/yargs/node_modules`) ;
5. inscription de *async* et de *yargs* dans la configuration *dependencies* de notre fichier `package.json`.

- *--save* : enregistre le module dans la clé *dependencies* ;
- *--save-exact* : idem que `--save` mais ne rajoute pas de préfixe au numéro de version (exemple : `1.3.1` au lieu de `~1.3.1`) ;
- *--save-dev* : enregistre le module dans la clé *devDependencies* ;
- *--save-optional* : enregistre le module dans la clé *optionalDependencies*.

1.3/ Découverte de l'éditeur VS Code

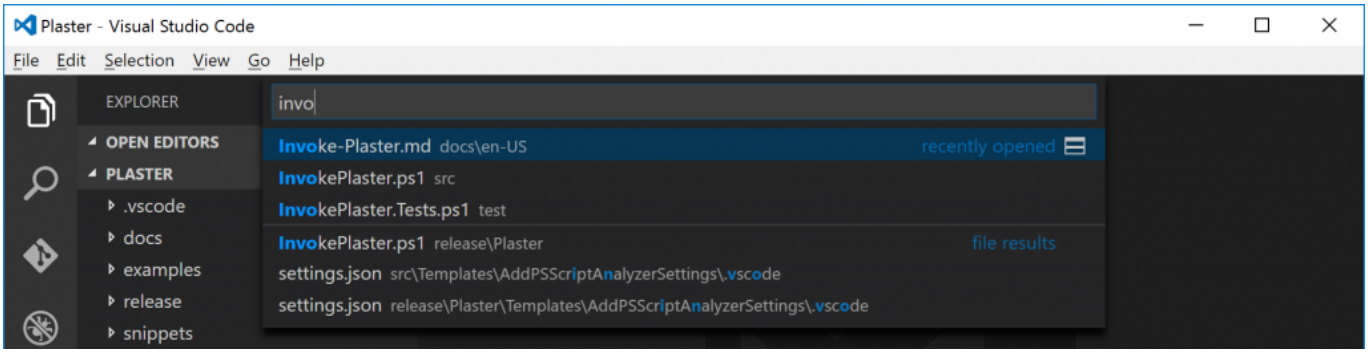
VS Code apporte toutes les fonctionnalités d'un éditeur moderne, sans nécessiter un investissement de formation.



Fonctionalités principales:

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

La palette de commande



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

- Utilisation des commandes
- Saisie intuitive

Installation de plugins.

> VS Code propose un lien direct vers les extensions disponibles.

Plugins pour Angular2 (ES6 TypeScript)

- Bootstrap 3 Snippets
- React-Native/React/Redux snippets for es6/es7
- React Toolbox Snippets
- React Redux ES6 Snippets
- JavaScript (ES6) code snippets
- AutoFileName

Manipulation

> Découverte et utilisation des packages

React Developer Tools

Afin de disposer d'outils spécifiques à `react` dans votre navigateur web, installez **React Developer Tools** :

- React Developer Tools pour Google Chrome
- React Developer Tools pour Mozilla Firefox

1.4/ Retour sur les fondamentaux JavaScript.

- **Etes-vous familier avec les fondamentaux JavaScript ?**

> Quelque questions simples pour confirmer le socle de compétences.

- **Types & Syntaxe**: Connaissez-vous les types **natifs de JS** ? Comment vous sentez-vous avec les nuances syntaxique en JS ?
- **Async & Performance**: Comment utilisez-vous les `callbacks pour gérer votre asynchrone? Pouvez-vous expliquer ce qu'est une promesse et pourquoi / comment il résout le "callback hell"?

> JavaScript ou ECMAScript ? On peut lire les termes *JavaScript* et *ECMAScript* comme équivalent. **JavaScript et ECMAScript sont la même chose**.

JavaScript a été inventé en 1995 par Brendan Eich alors qu'il était employé de la société *Netscape Communications*.

La spécification est validée par l'organisme *Ecma International* en juin 1997 sous le nom d'*ECMAScript*, standard ECMA-262.

L'utilisation du terme *JavaScript* est resté dans le vocabulaire courant.

Standard ECMA-262 Edition 5

ECMAScript a été standardisé dans sa version 5 en décembre 2009. Il s'agit de la version d'ECMAScript supportée depuis les débuts de Node. La révision 5.1 de juin 2011 est une correction mineure de la spécification.

Il s'agit d'une évolution majeure, dix ans après sa précédente édition, ECMAScript 3.

ECMAScript 5 introduit le mode strict limitant fortement les effets de bord indésirables, de nouvelles fonctionnalités pour `Object` et `Array`, le support natif de *JSON* et `Function.prototype.bind`.

Standard ECMA-262 Edition 2015

La spécification *ECMAScript 2015* (*ES2015*), successivement été appelée *ECMAScript Harmony* puis *ECMAScript 6*, a été publiée en juin 2015 et succède à *ECMAScript 5*.

La page Web suivante référence l'état de l'implémentation d'ECMAScript 2015 sur différentes plates-formes, dont Node :

- https://kangax.github.io/compat-table/es6/
- http://www.ecma-international.org/ecma-262/6.0/

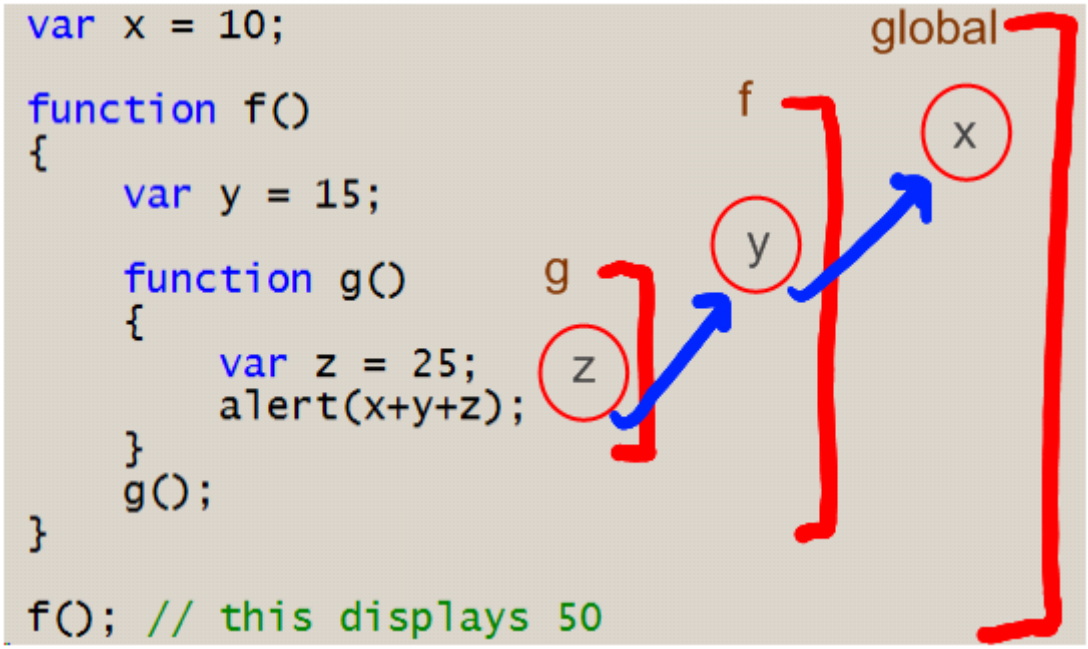
Fondamentaux JavaScript

JavaScript est un langage riche et *dynamique* qui peut sembler déroutant comparé à d'autres langages syntaxiquement proches. Les fondamentaux JavaScript sont les points clés de la compréhension du langage.

Scope Chain et code hoisting

Le code contenu dans le bloc déclaratif d'une fonction crée une portée (ou scope) qui est invisible au contexte parent de l'exécution de cette fonction.

> Le contexte parent d'une fonction est fixé avant l'exécution (code hoisting) par la position du mot clé `function`.



Ex 1 : Scope Chain

```
// Prédire le résultat du script suivant.
var x = 10, y = 11, z = 12;

foo();

function foo(y){

	bar();

	function bar(){
		z = 20;
		console.log(x,y,z);
	}
}

console.log(x);
console.log(y);
console.log(z);
```

Code Hoisting

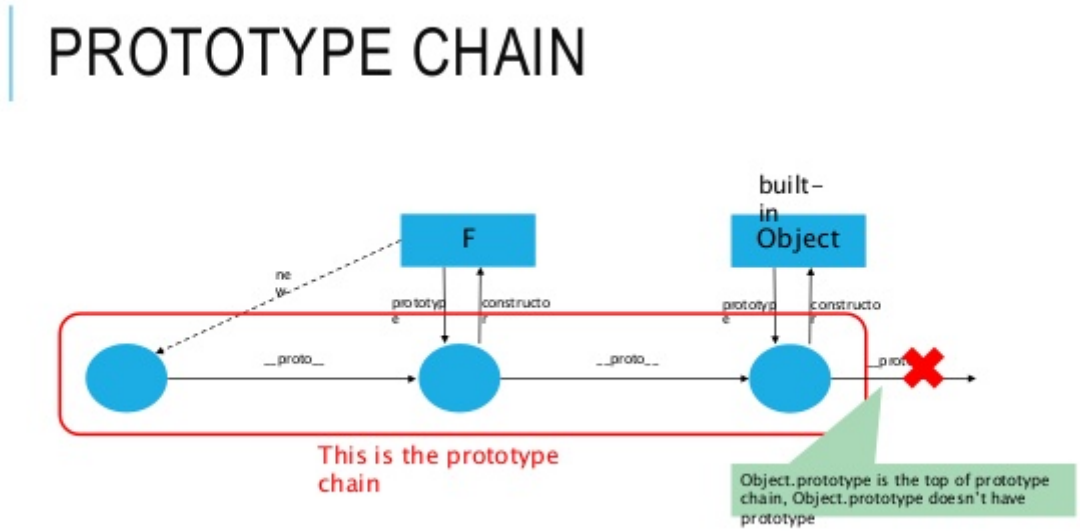
Hoisting (hissage en anglais). Phase d'optimisation du code avant son éxécution (AST) **les déclaration sont identifiées et ""hissées"**.

1. Phase : les déclarations (`var`) de vairiables uniquement.
2. Phase : les ligne déclaratives de fonction (ligne commeçant par `function`).

> Rappel : l'objet **window** représente le contexte global d'éxécution du navigateur. Ce contexte peut changer selon l'implémentation des moteurs JavaScript.

Object

Dans ECMAScript, tout est objet. C'est le prototype qui détermine le comportement dudit objet. Les objets peuvent être créés de manière littérale, avec la fonction Object.create ou via un constructeur.



> Les `Object` sont transmis par référence.

Primitive

Une primitive (valeur primitive ou structure de donnée primitive) est une donnée qui n'est pas un objet et n'a pas de méthode. En JavaScript, il y a 6 types de données primitives: string, number, boolean, null, undefined, symbol (nouveauté d'ECMAScript 2015).

Primitives JavaScript encapsulées dans des objets Excepté dans les cas de `null` ou `undefined`, pour chaque valeur primitive il existe un objet équivalent qui la contient:

- *String* pour la primitive string
- *Number* pour la primitive number
- *Boolean* pour la primitive boolean
- *Symbol* pour la primitive symbol

> Les `Primitive` sont transmises par valeur.

ES5 "use strict" et méthodes moins connues.

> Tout vos script devraient utiliser la directive **'use strict';**

Le mode strict de ECMAScript 5 permet d'éxécuter un mode restrictif du moteur JavaScript.

```
// Tenter d'éxécuter ce code
"use strict";
msg = "Allo ! Je suis en mode strict !";
eval('alert(msg)');
```

Le mode strict apporte quelques changements à la sémantique « normale » de JavaScript.

> **Premièrement**, le mode strict élimine quelques erreurs silencieuses de JavaScript en les changeant en erreurs explicites.

> **Deuxièmement**, le mode strict corrige les erreurs limitant l'optimisation du code qui sera exécuté plus rapidement en mode strict.

> **Troisièmement**, le mode strict interdit les mot-clés susceptibles d'être définis dans les futures versions de ECMAScript.

ES5 méthodes moins connues.

> Les fonctionnalités dépréciées

Object.create

La méthode `Object.create()` crée un nouvel objet avec un prototype donné et des propriétés données.

```
var model = {msg:'Hello World'}
var o = Object.create(model, {
  // name est une propriété de donnée
  name: { writable: true, configurable: true, value: 'ES6' },
  // age est une propriété d'accesseur/mutateur
  age: {
    configurable: false,
    get: function() { return 10; },
    set: function(value) { console.log('Définir o.name à', value); }
  }
});
console.log(o.name,o.age,o.msg);
```

Object.defineProperty

La méthode `Object.defineProperty()` permet de définir une nouvelle propriété ou de modifier une propriété existante, directement sur un objet.

> Object.defineProperty(obj, prop, descripteur) **obj** L'objet sur lequel on souhaite définir ou modifier une propriété. **prop** Le nom de la propriété qu'on définit ou qu'on modifie. **descripteur** Le descripteur de la propriété qu'on définit ou qu'on modifie.

Les descripteurs de données et d'accesseur sont des objets.

```
var obj = {};
// en utilisant __proto__
Object.defineProperty(obj, "clé", {
  __proto__: null, // aucune propriété héritée
  value: "static"  // non énumérable
                   // non configurable
                   // non accessible en écriture
                   // par défaut
});

// en étant explicite
Object.defineProperty(obj, "clé", {
  enumerable: false,
  configurable: false,
  writable: false,
  value: "static"
});

var valeurB = 38;
Object.defineProperty(o, "b", {get : function(){ return valeurB; },
                               set : function(nouvelleValeur){ valeurB =
nouvelleValeur; },
                               enumerable : true,
                               configurable : true});
```

Object.seal

La méthode Object.seal() scelle un objet afin d'empêcher l'ajout de nouvelles propriétés, en marquant les propriétés existantes comme non-configurables.

Les valeurs des propriétés courantes peuvent toujours être modifiées si elles sont accessibles en écriture.

```
var obj = {
    prop: function () {},
    msg: "Hello"
  };

// On peut ajouter de nouvelles propriétés
// Les propriétés existantes peuvent être changées ou retirées
obj.msg = "World";
obj.name = "Bob";
delete obj.name;

var o = Object.seal(obj);
o === obj; // true
Object.isSealed(obj); // true

obj.coincoin = "mon canard"; // la propriété n'est pas ajoutée
delete obj.msg; // la propriété n'est pas supprimée
```

Array.prototype.map

La méthode map() crée un nouveau tableau composé des images des éléments d'un tableau par une fonction donnée en argument.

Array.prototype.filter

La méthode filter() crée et retourne un nouveau tableau contenant tous les éléments du tableau d'origine pour lesquels la fonction callback retourne true.

Array.prototype.reduce

La méthode reduce() applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite) afin de la réduire à une seule valeur.

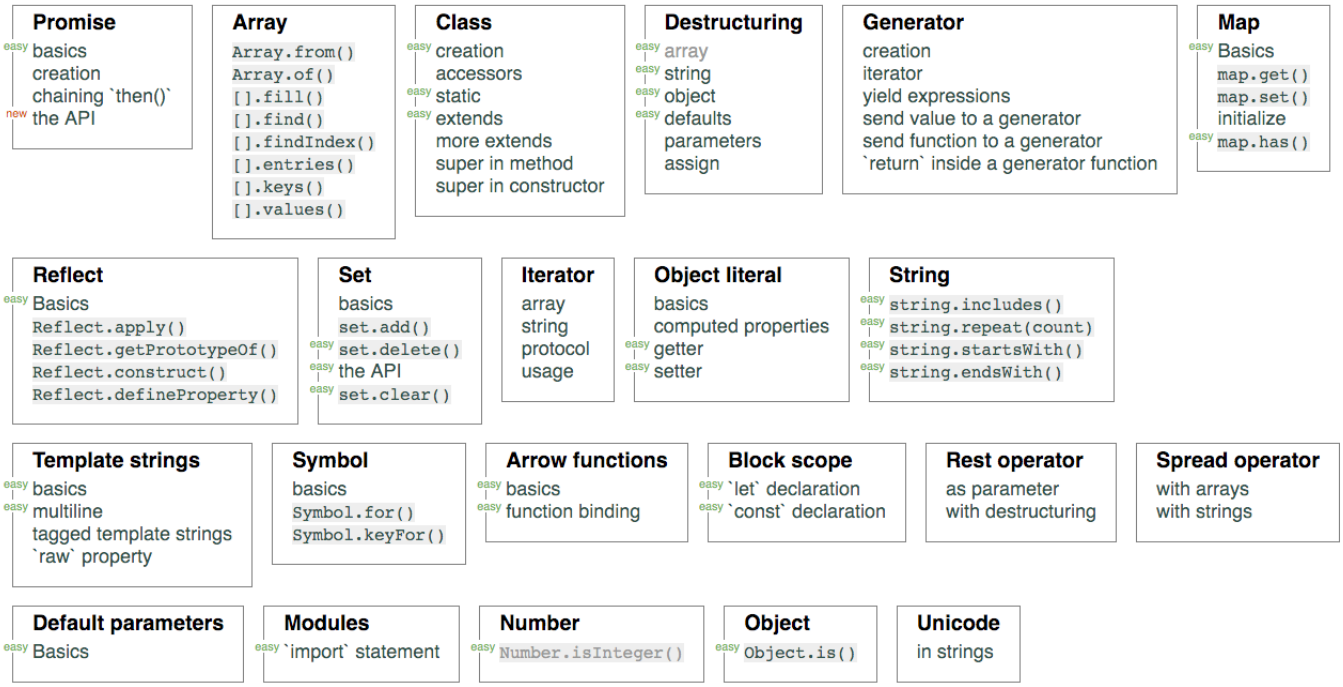
```
var users = [{name:'Bill',age:'10'},{name:'Bob',age:'19'},
{name:'Marine',age:'25'}]

users.filter(function(e,i,a){return e.age > 18 })
     .map(function(e,i,a){ var n = e; n.name = n.name.toUpperCase(); return
n; })
     .reduce(function(e1, e2, i , a) {
         return e1.name.concat(e2.name)
      });
```

Ecma-Script 6 / 2015

La syntaxe ES6 apporte une plus grande concision au code et résoud certaines problématiques liées au scope.

- **cf.** Node.js Support de ES6
- **cf.** Table de compatibilité ES6

Aperçu général ES6.



On notera particulièrement :

- `let/const`
- `class`
- Arrow Function Expression
- `Promise`

Variables de bloc.

L'instruction `let` permet de déclarer des variables dont la portée est limitée à celle du bloc dans lequel elles sont déclarées.

> Au niveau le plus haut (la portée globale), let crée une variable globale alors que var ajoute une propriété à l'objet.

```
var x = 'global';
let y = 'global2';
console.log(this.x); // "global"
console.log(this.y); // undefined
console.log(y);      // "global2"
```

Rappel : Le mot-clé `var` permet de définir une variable globale ou locale à une fonction (sans distinction des blocs utilisés dans la fonction).

```
for (let i = 1; i <= 5; i++) {
  setTimeout(function(){console.info(i)},1000)
}

for (var i = 1; i <= 5; i++) {
  setTimeout(function(){console.warn(i)},1000)
}
```

Redéclarer une même variable au sein d'une même portée de bloc entraîne une exception `TypeError`.

```
if (x) {
  let myVar;
  let myVar; // TypeError
}

function foo() {
  let myVar;
  let myVar; // Cela fonctionne.
}
```

Faire référence à une variable dans un bloc avant la déclaration de celle-ci avec `let` entraînera une exception `ReferenceError`.

> **Zone morte temporaire (temporal dead zone / TDZ)** concerne les erreurs liées à `let` et `const`.
>
> La variable est placée dans une « zone morte temporaire » entre le début du bloc et le moment où la déclaration est traitée.

```
function foo() {
  console.log(myVar); // ReferenceError
  let myVar = true;
}
```

Constantes

La déclaration const permet de créer une constante nommée accessible uniquement en lecture.

> **Attention:** Cela ne signifie pas que la valeur contenue est immuable, uniquement que l'identifiant ne peut pas être réaffecté.

Les constantes font partie de la portée du bloc (comme les variables définies avec let).

```
const myVar = true;
myVar = false;

const myObj = {};
myObj.property = false;
myObj = {};
```

- Il est nécessaire d'initialiser une constante lors de sa déclaration.
- Il est impossible d'avoir une constante qui partage le même nom qu'une variable ou qu'une fonction. (Au sein d'une même portée)

> **Usage:** On préférera `let` pour les variables et `const`pour les fonctions.

```
const foo = function foo() {
  let myVar = true;
  console.log(myVar);
};
```

Paramètres par défaut de fonction.

En JavaScript, les paramètres de fonction non renseignés valent `undefined`. En ES6 il est possible de définir une valeur par défaut.

```
//ES5
unction multiplier(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;

  return a*b;
}

multiplier(5); // 5
```

> Code allégé avec ES6

```
function multiplier(a, b = 1) {
  return a*b;
}

multiplier(5); // 5
```

Paramètres "rest".

La syntaxe des paramètres du `rest` permet de représenter un nombre indéfini d'arguments contenus dans un tableau.

```
function multiplier(facteur, ...lesArgs) {
  return lesArgs.map(x => facteur * x);
}

var arr = multiplier(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Opérateur "spread".

L'opérateur `spread` permet de développer une expression lorsque plusieurs arguments ou plusieurs éléments sont nécessaires (respectivement pour les appels de fonctions et les littéraux de tableaux).

Syntaxe

Pour l'utilisation de l'opérateur dans les appels de fonction :

```
foo(...objetIterable);
```

Pour les littéraux de tableaux :

```
[...objetIterable, 4, 5, 6]
```

Pour la décomposition :

```
[a, b, ...objetIterable] = [1, 2, 3, 4, 5];
```

Utilisation

Tout argument passé à une fonction peut être décomposé grâce à l'opérateur et l'opérateur peut être utilisé pour plusieurs arguments.

```
function foo(v, w, x, y, z) { console.log([v, w, x, y, z])}
var args = [0, 1];
foo(-1, ...args, 2, ...[3]);

//Ignorer des paramètres

foo(...[,,2,,4]);
```

Pour créer un nouveau tableau composé du premier, on peut utiliser un littéral de tableau avec la syntaxe de décomposition, cela devient plus succinct :

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

Pour la création d'objet :

```
var champsDate = lireChampsDate(maBaseDeDonnées);
var d = new Date(...champsDate);
```

Pour optimiser les méthodes :

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1.push(...arr2);
```

"Arrow Function" : portée lexicale. Usages.

> Les **Arrows Function** sont un raccourci syntaxique pour la création de fonction utilisant le symoble `=>` .

Les fonction ainsi crées sont syntaxiquement similaire à la fonction en C#, Java 8 et CoffeeScript.

Elles définissent le corps (bloc de code) et la valeur de retour. Contrairement aux fonctions classiques, les **Arrows function** partagent la même valeur lexicale pour le mot clé `this` que leur code environnant.

```
var odds = evens.map(v => v + 1);
var nums = evens.map((v, i) => v + i);
var pairs = evens.map(v => ({even: v, odd: v + 1}));

nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});

var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
}
```

Une expression de fonction fléchée permet d'utiliser une syntaxe plus concise que les expressions de fonctions classiques. La valeur `this` est alors **liée lexicalement.**

> Les **Arrows** sont un raccourci pour la création de fonction en utilisant le => . Les fonction ainsi crées sont syntaxiquement similaire à la fonction en C#, Java 8 et CoffeeScript.
>
> Elles définissent le corps (bloc de code) et la valeur de retour.

Modèles de classe et héritage. Méthodes statiques.

Les classes ont été introduites dans JavaScript avec ECMAScript 6 et sont un **sucre syntaxique** de l'héritage prototypal. Le mot clé `class` fournit une syntaxe plus claire pour utiliser un modèle et gérer l'héritage.

> **Cette syntaxe n'introduit pas un nouveau modèle d'héritage dans JavaScript !** JavaScript est un langage à paradigme multilpe : Orienté Objet (prototypal), impératif et fonctionnel. **ES6 n'introduit pas de paradigme Orienté Objet basé sur les classes**

Les classes sont des *fonctions spéciales* e la même façon qu'il y a des expressions de fonctions et des déclarations de fonctions, on aura deux syntaxe :

Déclarations de classes

```
class Polygone {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
}
```

Expressions de classes

Si on utilise un nom dans l'expression, ce nom ne sera accessible que depuis le corps de la classe.

```
// anonyme
var Polygone = class {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
};

// nommée
var Polygone = class Polygone {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
};
```

Remontée des déclarations (hoisting)

> Les déclarations de classes ne sont pas remontées dans le code, il est nécessaire de d'abord **déclarer la classe avant de l'utiliser**.

```
var p = new Polygone(); // ReferenceError

class Polygone {}
```

Corps d'une classe et définition des méthodes

Le corps d'une classe, partie contenue entre les accolades, définit les propriétés d'une classe comme ses méthodes ou **son constructeur**.

> Si la classe contient plusieurs occurences d'une méthode constructor, cela provoquera une exception `SyntaxError`.

```
class Polygone {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }

  get area() {
    return this.calcArea();
  }

  calcArea() {
    return this.largeur * this.hauteur;
  }
}

const carré = new Polygone(10, 10);

console.log(carré.area);
```

> **A noter** Il n'y pas de séparateur entre les membre d'une classe. **get/set** Permet l'utilisation de méthodes sous la forme de propriété.

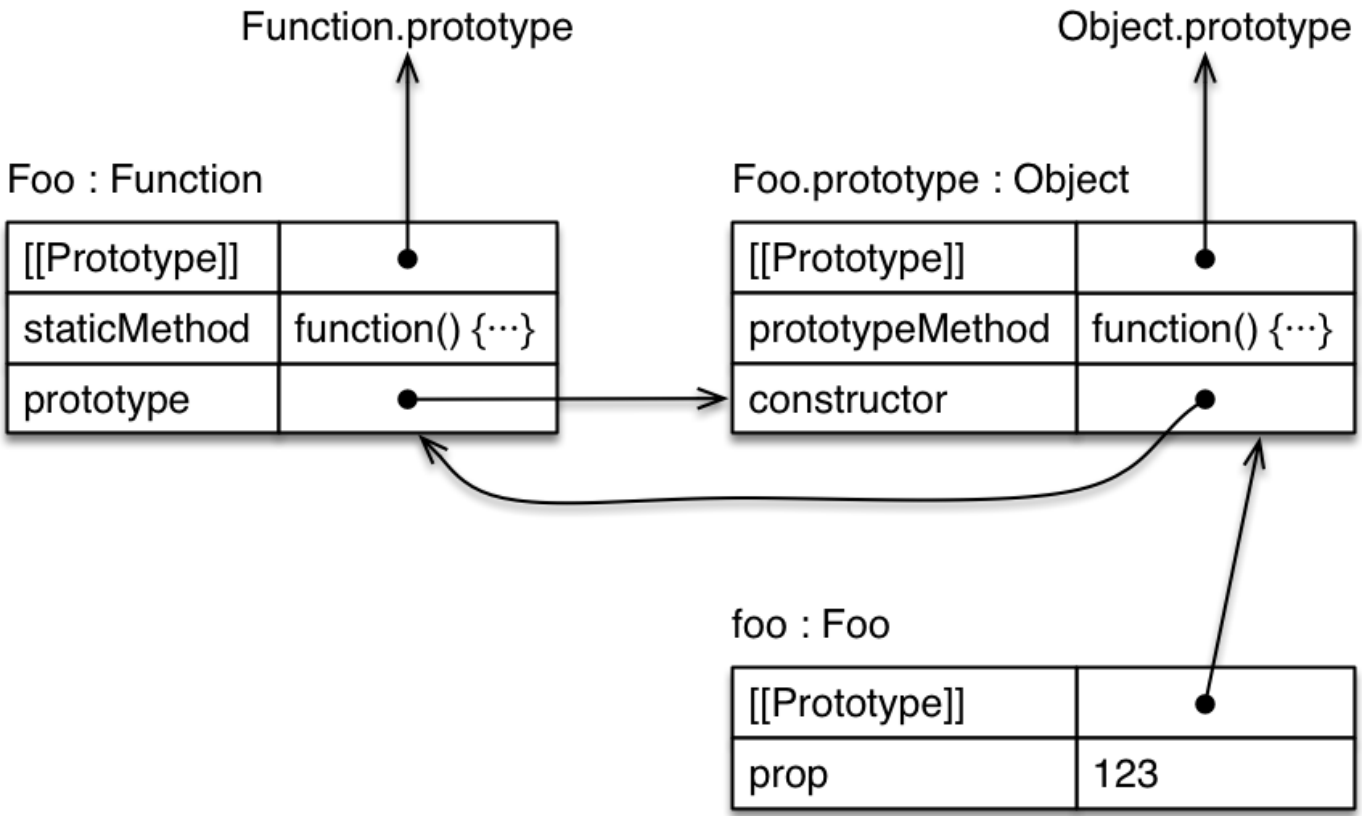
Méthodes statiques

Le mot-clé `static` permet de définir une méthode statique pour une classe.

> **Les méthodes statiques** sont appelées par rapport à la classe entière et non par rapport à une instance donnée.

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;

        return Math.sqrt(dx*dx + dy*dy);
    }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);

console.log(Point.distance(p1, p2));
```

Comprendre les méthodes statiques



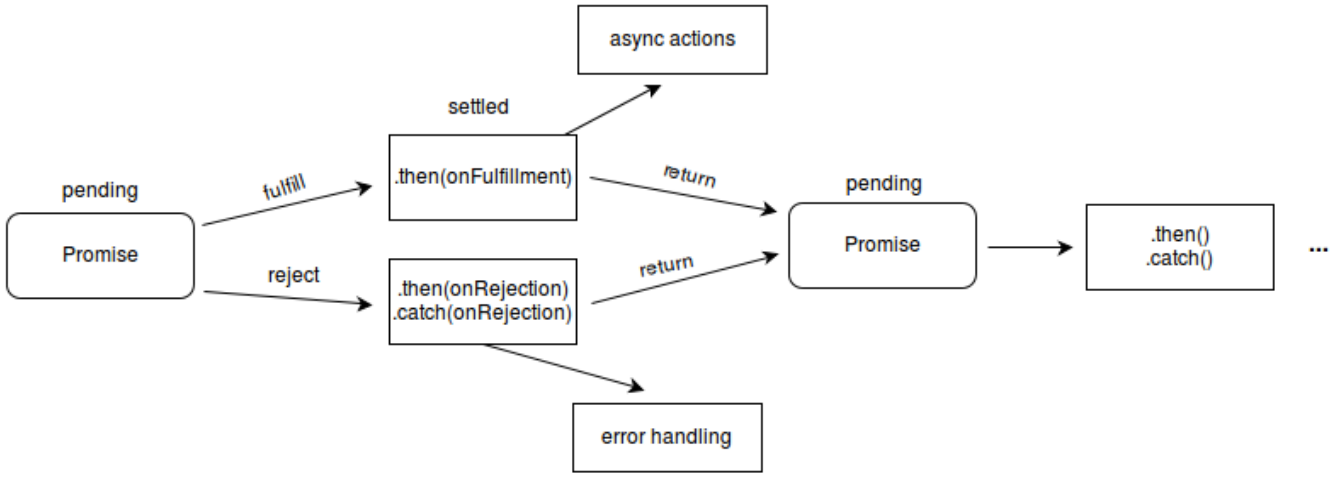
```
class Person {
    constructor(name) {
        this.name = name;
    }
    toString() {
        return `Person named ${this.name}`;
    }
    static logNames(persons) {
        for (const person of persons) {
            console.log(person.name);
        }
    }
}

class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    toString() {
        return `${super.toString()} (${this.title})`;
    }
}

const jane = new Employee('Jane', 'CTO');
console.log(jane.toString()); // Person named Jane (CTO)
```

Promise : gestion des traitements asynchrones.

L'objet Promise (pour « promesse ») est utilisé pour réaliser des opérations de façon asynchrone. Une promesse est dans un de ces états :

- en attente : état initial, la promesse n'est ni remplie, ni rompue
- tenue : l'opération a réussi
- rompue : l'opération a échoué
- acquittée : la promesse est tenue ou rompue mais elle n'est plus en attente.

```
new Promise(function(resolve, reject) { ... });
```



Promise : méthodes.

Promise.all(itérable) Renvoie une promesse qui est tenue lorsque toutes les promesses de l'argument itérables sont tenues. Si la promesse est tenue, elle est résolue avec un tableau contenant les valeurs de résolution des différentes promesses contenues dans l'itérable.

Promise.race(itérable) Renvoie une promesse qui est tenue ou rompue dès que l'une des promesses de l'itérable est tenue ou rompue avec la valeur ou la raison correspondante.

Promise.reject(raison) Renvoie un objet Promise qui est rompu avec la raison donnée.

Promise.resolve(valeur) Renvoie un objet Promise qui est tenue (résolue) avec la valeur donnée.

> Gérer les appels asyncrhrones

```
const get = (url) => {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.onload = () => resolve(xhr.responseText);
    xhr.send();
  });
};

/* getTweets (Generator) */

const getTweets = (function* () {
  // 1er
  yield get('https://api.myjson.com/bins/2qjdn');
  // 2nd
  yield get('https://api.myjson.com/bins/3zjqz');
  // 3rd
  yield get('https://api.myjson.com/bins/29e3f');

})();

// Initialisation et différentes consommation

Promise.all([...getTweets]).then((valeur)=> console.log(valeur), (raison)
=> console.log(raison));
```

Système natif de gestion des modules.

Avec le système natif le code du module est traité différemment pour gérer les exportations et les importations.

> `<script type = "module">` est introduit pour distinguer le code de script définissant des modules.

Les convention de nommage autorise les nom de module à utiliser des url.

```
import $ from 'jquery'
//équivaut à
import $ from 'https://code.jquery.com/jquery.js'
```

Export :

L'instruction export est utilisée pour permettre d'exporter des fonctions et objets ou des valeurs primitives à partir d'un fichier (ou module) donné.

La syntaxe ES6 est similaire à celle des module CJS

```
export function someMethod() {

}

export var another = {};

class Module {

  constructor(args='cool') {
    console.warn(args);
    this.name = args;
  }
  done(){
    console.info(this.name);
  }
}

//alias
export {Module as User}
```

> **A noter** le nom de la varibale ou function est réutilisé comme nom d'export. Mais il est possible de rédéfinir des alias

Consommation (Importation) :

L'instruction import est utilisée pour importer des fonctions, des objets ou des valeurs primitives exportées depuis un module externe ou un autre script.

L'import peut se faire par nom et/ou en redéfinissant des alias

```
import { someMethod, another as newName } from './exporter';

someMethod();
typeof newName == 'object';
```

> Le chemin de fichier ne nécessite pas l'extension

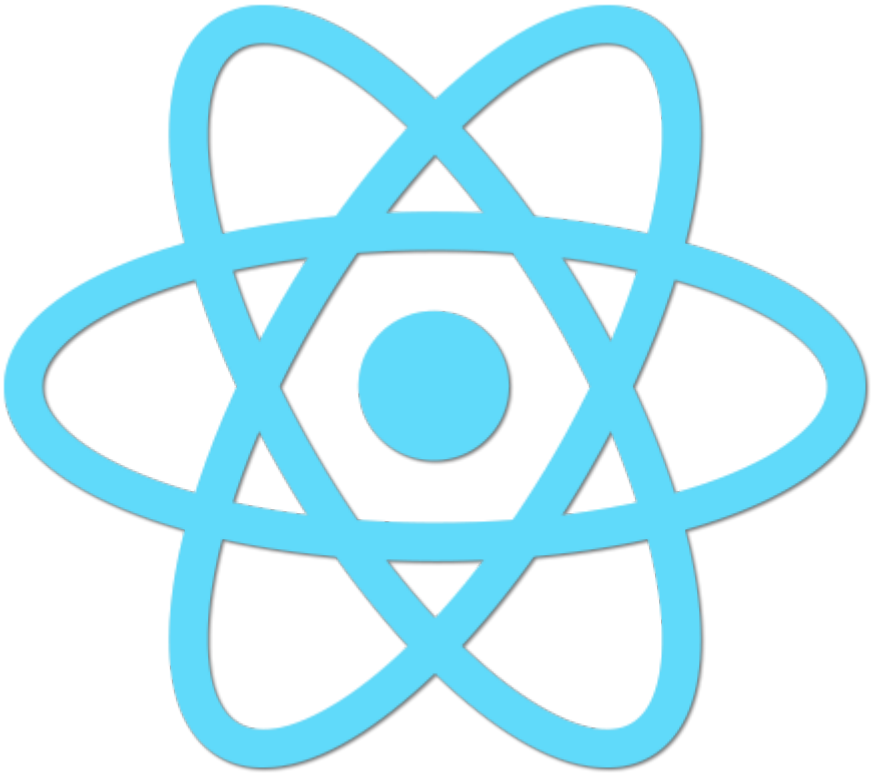
Import/Export par défaut :

Il est possible de définir par défaut un export simplifié

```
//export-default.js:
 export default function foo() {
    console.log('foo');
  }
```

```
//import-default.js:
import customName from './export-default';

customName(); // -> 'foo'
```

```
//Variation de la syntaxe :
import 'jquery';                          // importe un module
import $ from 'jquery';                   // importe l'export par défaut
import { $ } from 'jquery';               // importe un export nommé $
import { $ as jQuery } from 'jquery';     // importe un export nommé $ dans
l'alias jQuery

export var x = 42;                        // exporte une variable
export function foo() {};                 // export une fonction

export default 42;                        // exporte par défaut de variable
export default function foo() {};         // exporte par défaut de function

export { encrypt };                       // exporte la variable existante
encrypt
export { decrypt as dec };                // exporte la variable existante
encrypt sous le nom dec
export { encrypt as en } from 'crypto'; // exporte la variable existante
encrypt du module crypto sous le nom en
export * from 'crypto';                   // exporte tous les exports nommés
du module crypto
                                          // (sauf l'export par défaut)
import * as crypto from 'crypto';         // importe tous les exports nommés
du module crypto
```



Développer avec ReactJS.

Développer avec ReactJS

> React repose sur la composition.

Composition

Un composant est une partie unitaire de l'application. Chacune de ces parties peuvent être elles-mêmes **composées d'autres composants.** Une page web n'est alors rien d'autre qu'un **arbre de composants** dont les feuilles sont des balises html classiques.

Les composants sont isolés

Cette composition permet d'isoler des éléments de l'interface, leur comportement, leur état, leur forme et même, dans une certaine mesure, leur style. L'isolation des composants permet d'intégrer des éléments React dans une application existante. **On peut grâce à la composition migrer une application progressivement.**

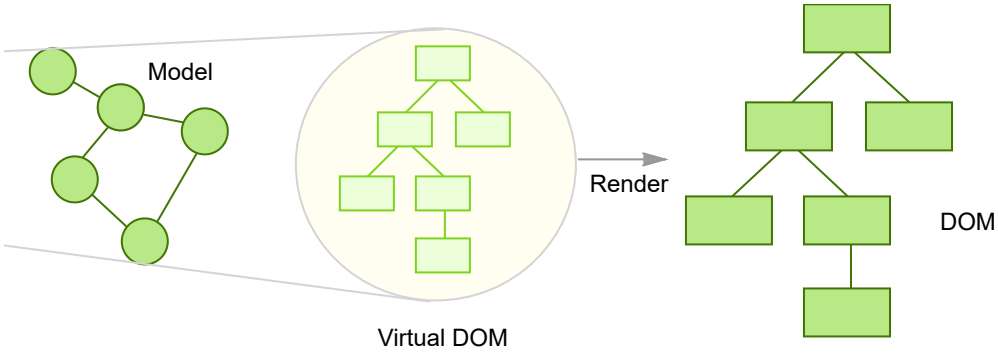
Les composants sont réutilisables, testables

Un composant isolé du reste d'un layout est plus facilement réutilisable, il suffit de le brancher aux bons endroits, avec la bonne source de données. Il est plus facilement testable et vous permet d'avoir une bonne couverture de tests unitaires. **Il suffit de tester le comportement du composant en faisant abstraction de son contexte.**

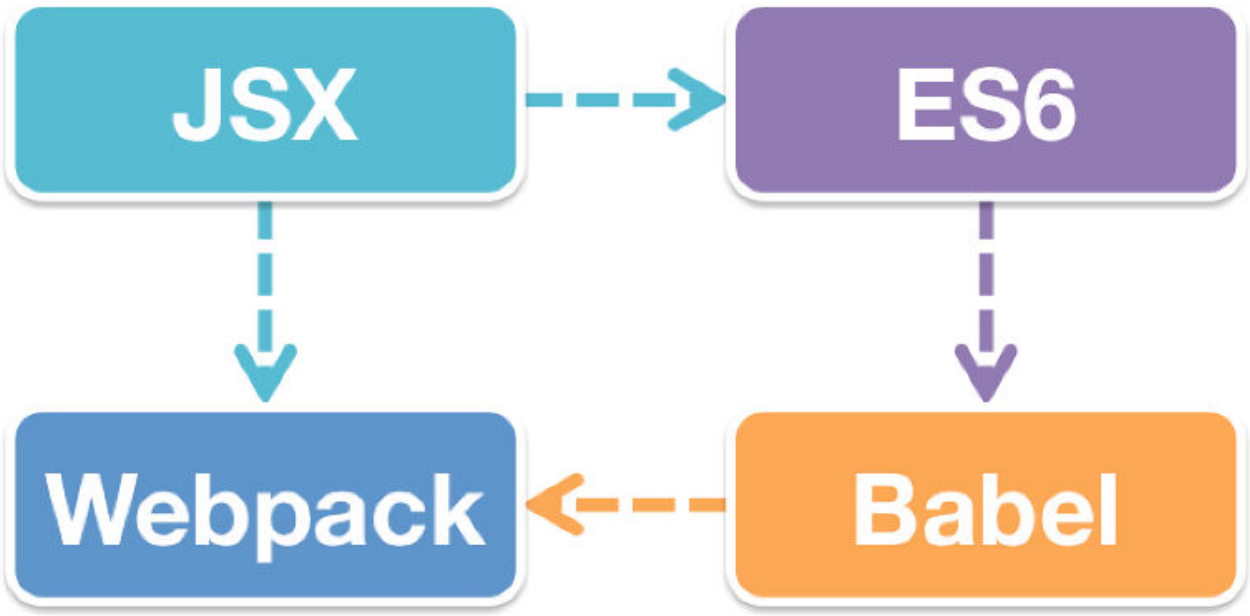
> Lorsque vous modifiez un composant, les effets de bord sont limités à ce composant uniquement. Votre application est plus maintenable.

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

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

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



Utiliser JavaScript ou JSX.



Fondamentalement, JSX fournit juste du sucre syntaxique pour la fonction `React.createElement(component, props, ...children)`.

> Le code JSX:

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

> Devient

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

Comprendre JSX en détail.

Déclaration de composants.

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

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

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

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

Vous pouvez également vous **référer à un composant React en utilisant la notation pointée à partir de JSX.** Cela est pratique si vous avez un seul module qui exporte de nombreux composants React.

> Par exemple, si MyComponents.DatePicker est un composant, vous pouvez l'utiliser directement à partir de JSX avec:
>
> ```
> import React from 'react';
>
> const MyComponents = {
>  DatePicker: function DatePicker(props) {
>    return Imagine a {props.color} datepicker here. ;
>  }
> }
>
> function BlueDatePicker() {
>   return <MyComponents.DatePicker color="blue" />;
> }
> ```

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

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

```
import React from 'react';

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

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

Choix du type à l'exécution

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

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

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

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

Initialiser les propriétés de composants.

Il existe plusieurs façons de spécifier les propriétés en JSX.

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

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

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

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

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

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

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

Composants descendants (enfants).

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

Il existe plusieurs façons de transmettre des enfants:

- String Literals
- JavaScript Expressions
- JSX Children
- function

> Booleans, Null, and Undefined seront ignorés.

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

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

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

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

```
<div>{undefined}</div>
<div>{true}</div>
```

Template conditionnel basé sur les valeurs ignorées

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

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

> **Attention:** Certaines valeurs "*falsy*", telles que le nombre 0, sont toujours rendues par React.

Rendu des éléments.

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

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

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

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

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

Rendu d'un élément dans le DOM

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

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

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

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

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

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

ReactDOM.render (
  elementRef,
  Document.getElementById ('root')
);
```

Essayez sur CodePen.

Mise à jour de l'élément rendu

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

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

Considérez cet exemple d'horloge tic-tac:

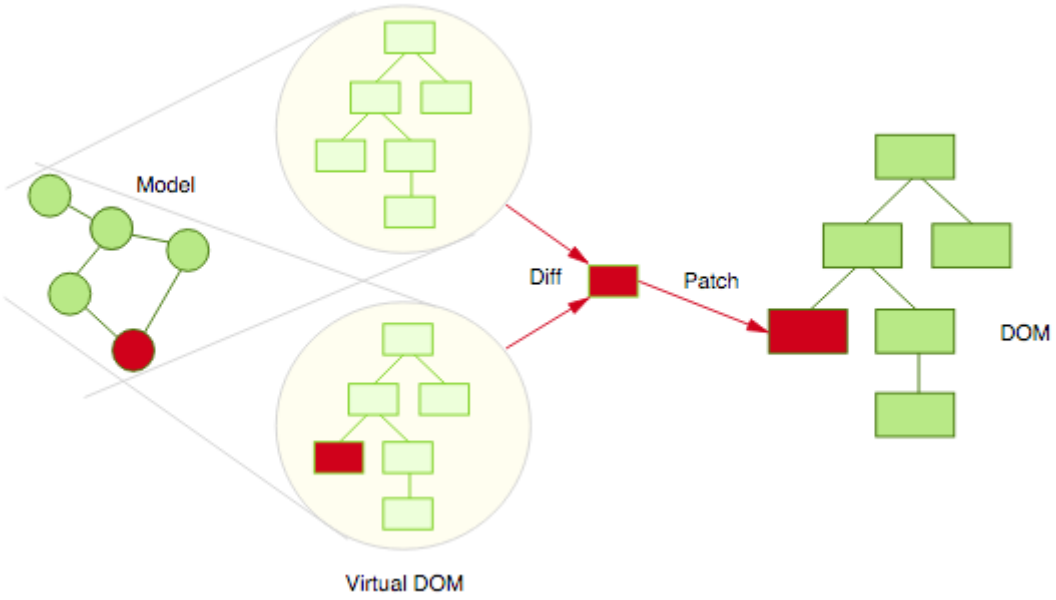
```
function tick () {
  const element = (
  <Div>
  <H1> Bonjour, monde! </ H1>
  <H2> C'est {new Date().ToLocaleTimeString ()}. </ H2>
  </ Div>
  );
  ReactDOM.render (
    element,
    Document.getElementById ('root')
  );
}

setInterval(tick, 1000);
```

Essayez sur CodePen.

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

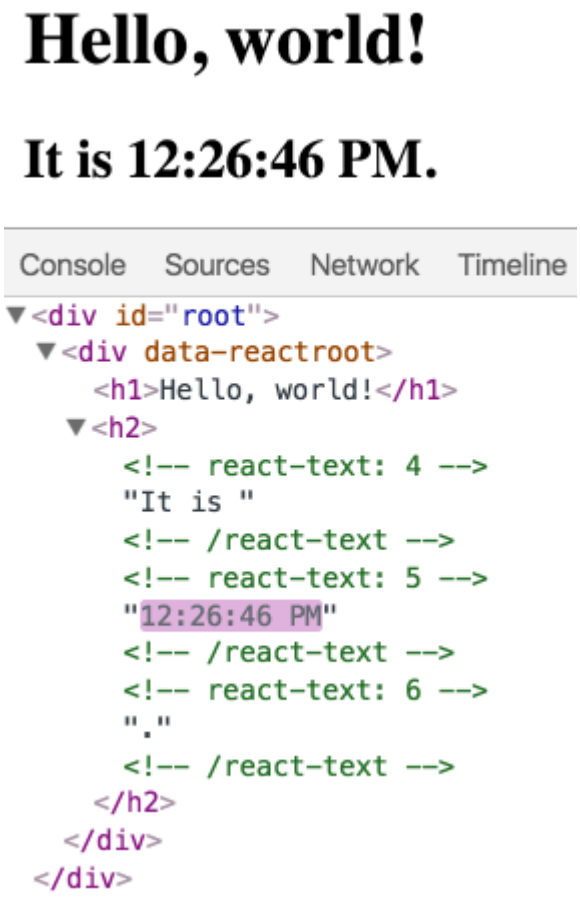
> **Note:**
>
> En pratique, la plupart des applications React n'appellent que `ReactDOM.render()` une fois.

React ne met à jour qui est nécessaire



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

Vous pouvez vérifier en examinant le dernier exemple avec les outils du navigateur:



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

Méthodes principales de l'API.

> L'API React est simple et concise.

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

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

Composants

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

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

Composants fonctionnels et de classe

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

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

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

Vous pouvez également utiliser une classe ES6 pour définir un composant:

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

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

Rendu d'un composant

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

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

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

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

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

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

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

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

Essayez sur CodePen.

Reprenons ce qui se passe dans cet exemple:

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

> **Attention:**
>
> Prefixez toujours les noms des composants avec une majuscule.
>
> Par exemple, `<div />` représente une balise DOM, mais `<Welcome />` représente un composant et nécessite que `Welcome` soit dans la portée.

Composants

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

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

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

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

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

Essayez sur CodePen.

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

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

Extraction de composants

> **Diviser les composants en composants plus petits.**

Par exemple, considérez ce composant `Comment`:

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

Essayez sur CodePen.

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

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

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

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

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

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

Nous pouvons maintenant simplifier `Comment` un petit peu:

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

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

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

Cela nous permet de simplifier encore `Comment`:

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

```
        </div>
      </div>
    );
  }
```

Essayez sur CodePen.

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

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

Les `props` sont en lecture seule.

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

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

> De telles fonctions sont appelées "pure" car elles n'essayent pas de modifier leurs entrées et renvoient toujours le même résultat pour les mêmes entrées.

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

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

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

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

Création de composant de vues. Cycle de vie.

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

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

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

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

setInterval(tick, 1000);
```

Essayez sur CodePen.

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

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

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

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

Le **"state"** est similaire aux **"propos"**, mais il est privé et entièrement contrôlé par le composant.

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

Conversion d'une fonction en classe

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

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

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

Essayez sur CodePen.

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

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

Ajout d'un état local à une classe

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

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

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

2. Ajoutez un [constructeur de classe] (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes# Constructor) qui attribue l'état initial `this.state`:

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

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

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

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

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

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

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

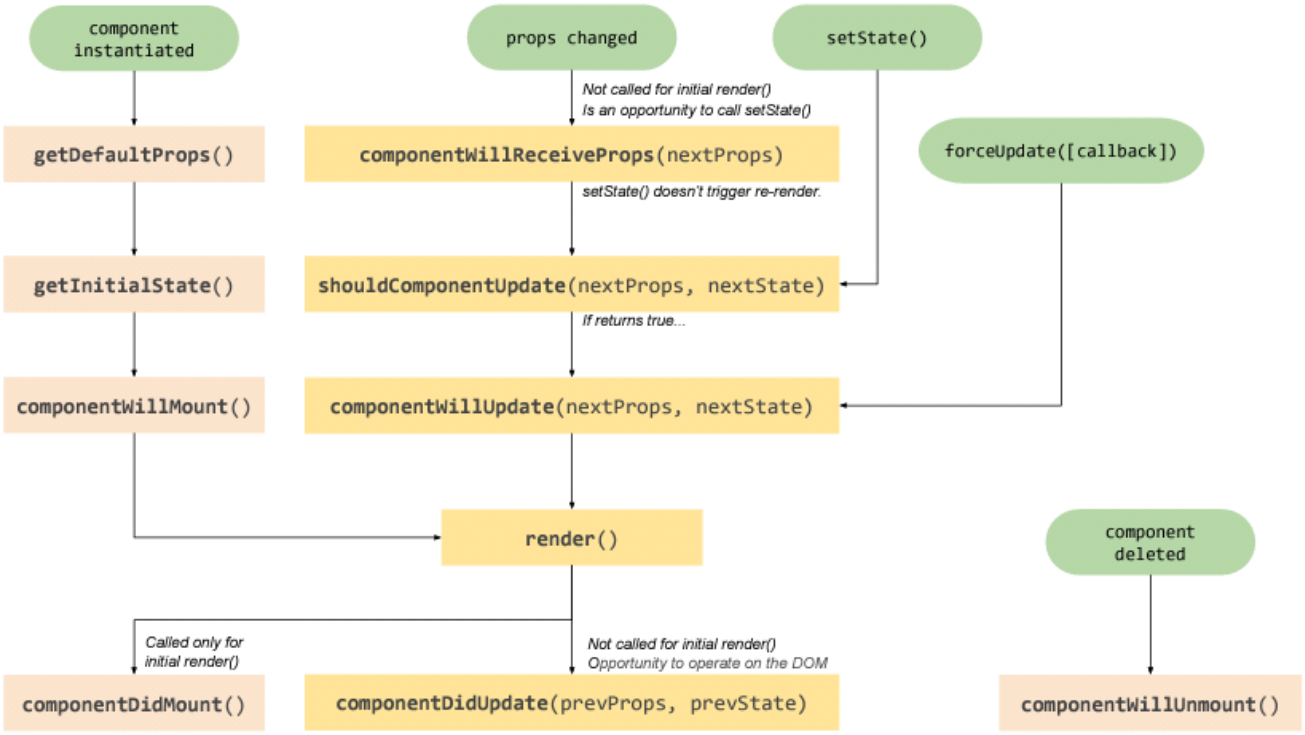
Nous ajouterons ensuite le code DU `timer` au composant lui-même.

Le résultat est le suivant:

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

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

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

Essayez sur CodePen.

Ajout de méthodes de cycle de vie à une classe



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

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

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

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

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

  componentDidMount() {

  }

  componentWillUnmount() {

  }

  render() {
    return (
      <div>
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Essayez sur CodePen.

Récapitulif:

1. Lorsque `<Clock />` est passé à `ReactDOM.render()`, React appelle le constructeur du composant `Clock`. Puisque `Clock` a besoin d'afficher l'heure actuelle, il initialise `this.state` avec un objet incluant l'heure actuelle.

2. React appelle la méthode `render()` du composant `Clock` pour mettre à jour le DOM.

3. Lorsque le composant `Clock` est inséré dans le DOM, React appelle le crochet `componentDidMount()` du cycle de vie.

4. Chaque seconde, le navigateur appelle la méthode `tick()` qui planifie une mise à jour de l'interface utilisateur en appelant`setState ()`. Grâce à l'appel `setState ()`, React sait que l'état a changé et appelle la méthode `render ()` pour apprendre ce qui devrait être à l'écran.

5. Si le composant `Clock` est supprimé du DOM, React appelle `componentWillUnmount()`.

Utilisation correcte de l'état

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

Ne pas modifier l'état directement

Par exemple, cela ne rendra pas un composant:

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

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

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

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

Les mises à jour d'état peuvent être asynchrones.

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

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

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

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

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

ou

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

Mises à jour d'état sont fusionnées

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

Par exemple, votre état peut contenir plusieurs variables indépendantes:

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

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

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

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

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

Flux de données descendants.

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

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

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

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

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

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

Le composant `FormattedDate` reçoit la `date` dans ses **"props"** sans en connaître la provenance.

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

Essayez sur CodePen.

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

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

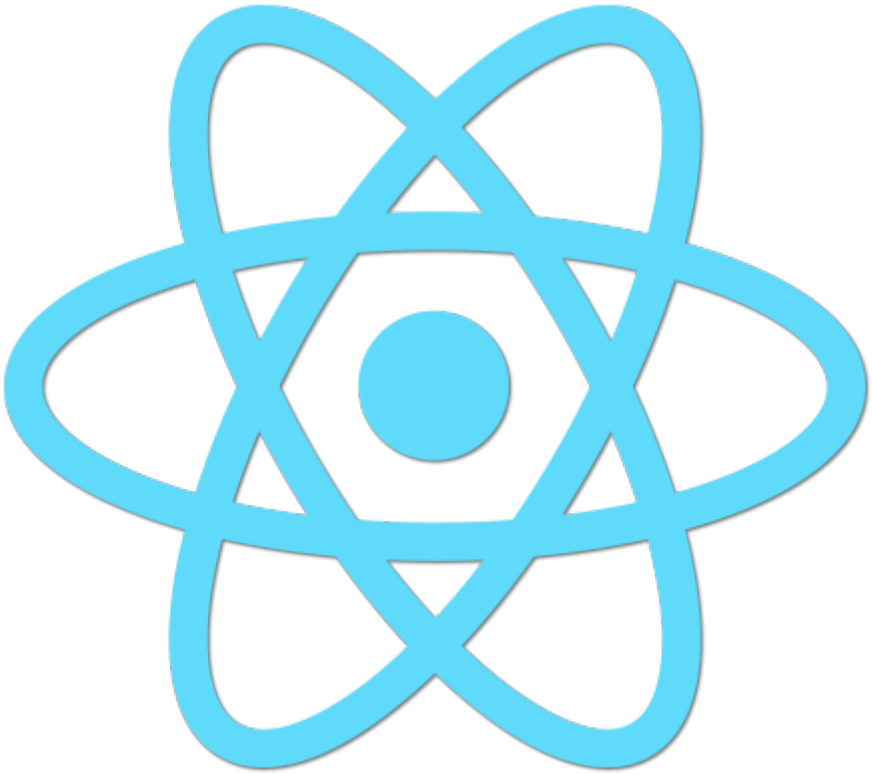
Pour montrer que tous les composants sont vraiment isolés, nous pouvons créer un composant `App` qui rend trois `<Clock>`s:

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

ReactDOM.render(
  <App />,
```

```
  document.getElementById('root')
);
```

Essayez sur CodePen.

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



Développer avec ReactJS.

Interactivité des composants

Gestion des événements. "autobinding" et délégation.

La gestion des événements avec React éléments est très similaire à la manipulation des événements sur les éléments DOM. Il y a quelques différences syntaxiques:

- **Les événements React sont nommés en utilisant camelCase**, plutôt que de minuscules.
- Avec **JSX vous passez une fonction en tant que gestionnaire d'événements**, plutôt qu'une chaîne.

Par exemple, le code HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

est légèrement différent dans React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Une autre différence est que **vous ne pouvez pas retourner `false` pour empêcher le comportement par défaut dans React.** Vous devez appeler `preventDefault` explicitement.

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

Ici, `e` est un événement synthétique. React définit ces événements synthétiques selon la spécification W3C, de sorte que vous ne devez pas vous soucier de la compatibilité cross-browser.

Lors de l'utilisation React vous devriez pas appeler `addEventListener` pour ajouter des écouteurs à un élément du DOM après sa création.

Lorsque vous définissez un composant à l'aide d'une classe ES6, il est commun d'utiliser une méthode de la classe comme gestionnaire d'événements.

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

Essayez-le sur CodePen.

Vous devez être prudent sur le sens de `this` dans callbacks JSX. En JavaScript, les méthodes de classe ne sont pas liées par défaut. Si vous oubliez de lier `this.handleClick` et le transmettre à `onClick`, `this` aura `undefined` lorsque la fonction est effectivement appelé.

Si vous appelez `bind` vous ennuie, vous pouvez utiliser la syntaxe expérimentale d'initialisation de la propriété :

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
```

```
    );
  }
}
```

Cette syntaxe est activée par défaut dans create-react-app.

Vous pouvez également utiliser une `Arrow Function`:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={(e) => this.handleClick(e)}>
        Click me
      </button>
    );
  }
}
```

Design Pattern : stratégie pour les composants à état.

Souvent, plusieurs éléments doivent refléter les mêmes données. **Il est recommandé de remonter les `state` dans un ancêtre commun.

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    const value = this.props.value;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={value}
               onChange={this.handleChange} />
      </fieldset>
    );
```

```
  }
}
```

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {value: '', scale: 'c'};
  }

  handleCelsiusChange(value) {
    this.setState({scale: 'c', value});
  }

  handleFahrenheitChange(value) {
    this.setState({scale: 'f', value});
  }

  render() {
    const scale = this.state.scale;
    const value = this.state.value;
    const celsius = scale === 'f' ? tryConvert(value, toCelsius) : value;
    const fahrenheit = scale === 'c' ? tryConvert(value, toFahrenheit) :
value;

    return (
      <div>
        <TemperatureInput
          scale="c"
          value={celsius}
          onChange={this.handleCelsiusChange} />
        <TemperatureInput
          scale="f"
          value={fahrenheit}
          onChange={this.handleFahrenheitChange} />
        <BoilingVerdict
          celsius={parseFloat(celsius)} />
      </div>
    );
  }
}
```

Essayez-le sur CodePen.

Peu importe quelle entrée vous modifiez, `this.state.value` et `this.state.scale` dans le `Calculator` sont mis à jour.

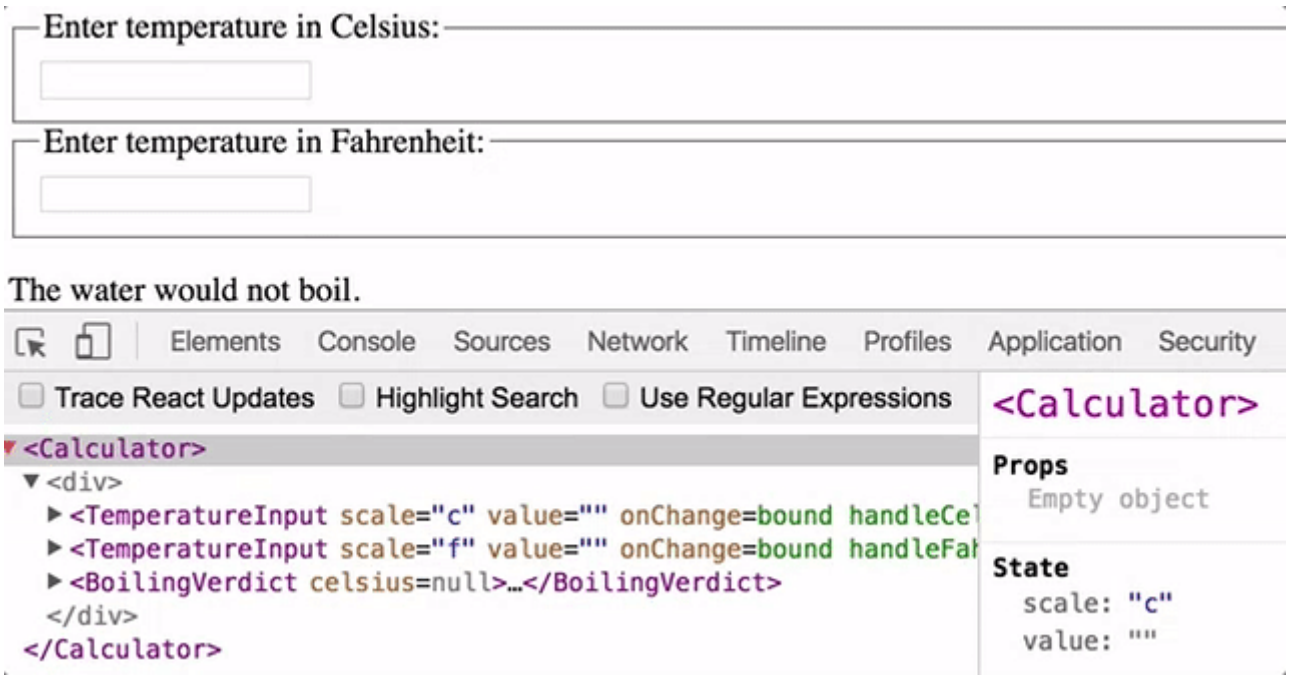
Récapitulons ce qui se passe lorsque vous modifiez une entrée:

- React appelle la fonction spécifiée comme `onChange` sur le DOM`<input>`. Dans ce cas, cela est la méthode `handleChange` dans `TemperatureInput`.
- La méthode `handleChange` dans le composant `TemperatureInput` appelle `this.props.onChange()` avec la nouvelle valeur souhaitée. Ses props, y compris `onChange`, ont été fournis par son composant parent, le `Calculator`.
- Quand il déjà rendu, le `Calculator` a précisé que `onChange` de Celsius `TemperatureInput` est laméthode `handleCelsiusChange` de `Calculator`, et `onChange` de Fahrenheit `TemperatureInput` est la méthode `handleFahrehnheitChange`.
- A l'intérieur de ces méthodes, le composant `Calculator` demande React un nouveau rendu en appelant `this.setState ()` avec la nouvelle valeur d'entrée.
- React appelle la méthode de `render()` du `Calculator`.
- React appels les méthodes `render()` des composants individuels de `TemperatureInput` avec leurs nouveaux props spécifiés par le `Calculator`.
- React met à jour le DOM pour faire correspondre les valeurs d'entrée souhaitées.

Chaque mise à jour passe par les mêmes étapes de sorte que les entrées restent synchronisés.

> **Conseils**
>
> Il devrait y avoir une seule source pour les données qui change dans une application React.

> Vous pouvez utiliser les ReactDeveloper Tools pour inspecter les `props` et remonter l'arborescence jusqu'à trouver le composant responsable de la mise à jour du Etat.

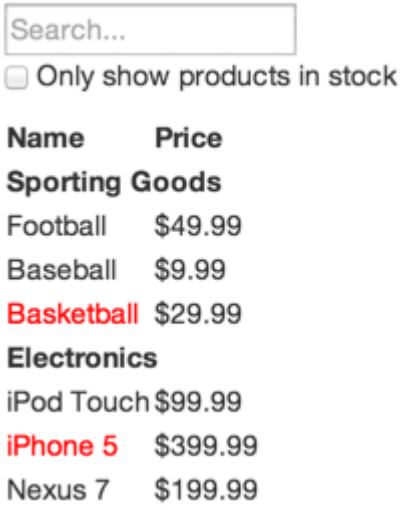


Composer par ensembles.

L'un des bénéfices de React est la façon dont il permet de concevoir les applications.

Commencez avec un Mock

Imaginez que nous avons déjà une API JSON et une maquette de notre designer. La maquette ressemble à ceci:



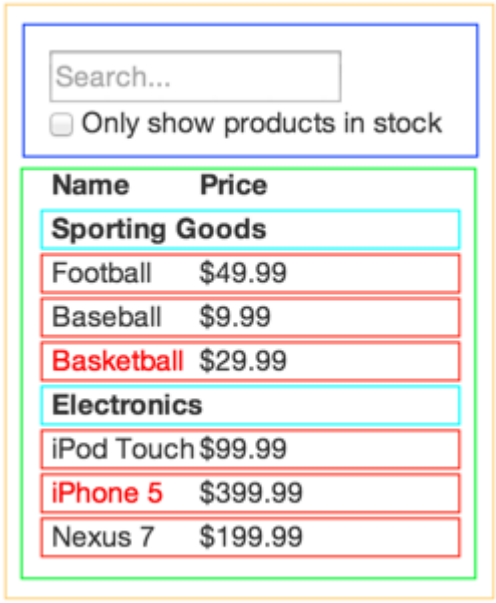
Notre API JSON renvoie des données qui ressemble à ceci:

```
[
  {Category: "Articles de sport", prix: "49,99 $", approvisionné: true, le
nom: "Football"},
  {Category: "Articles de sport", prix: "$ 9.99", approvisionné: true, le
nom: "Baseball"},
  {Category: "Articles de sport", prix: "29,99 $", approvisionné: false, le
nom: "Basketball"},
  {Catégorie: "Electronique", prix: "99,99 $", approvisionné: true, le nom:
"iPod Touch"},
  {Catégorie: "Electronique", prix: «399,99 $», rempli: false, le nom:
"iPhone 5"},
  {Catégorie: "Electronique", prix: «199,99 $», approvisionné: true, le
nom: "Nexus 7"}
]
```

Etape 1: Définir l'interface utilisateur dans une hiérarchie de composants

La première chose que vous voulez faire est de dessiner des boîtes autour de chaque composant dans la maquette et leur donner tous les noms.

Mais comment savez-vous ce qui devrait être sa propre composante? Il suffit d'utiliser la technique du principe de responsabilité unique, **"un composant devrait idéalement seulement faire une chose"**.



Ici que nous avons cinq composants simples pour l'application .

1. `FilterableProductTable` **(orange):** contient l'intégralité de l'exemple.
2. `SearchBar` **(bleu):** reçoit toutes les *entrées utilisateur*.
3. `ProductTable` **(vert):** affiche et filtre la collecte de données **basé sur l'entrée d'utilisateur**
4. `ProductCategoryRow` **(turquoise):** affiche une rubrique pour chaque catégorie.
5. `ProductRow` **(rouge):** affiche une ligne pour chaque produit.

Maintenant que nous avons identifié les composants dans notre maquette, nous allons les organiser dans une hiérarchie. **Les composants qui apparaissent dans un autre composant dans la maquette doit apparaître comme un enfant dans la hiérarchie:**

- `FilterableProductTable`
  - `SearchBar`
  - `ProductTable`
    - `ProductCategoryRow`
    - `ProductRow`

Etape 2: Construire une version statique dans React

Maintenant que vous avez votre hiérarchie de composants, il est temps de mettre en œuvre votre application. La meilleure façon est de construire une version qui prend votre modèle de données et rend l'interface utilisateur, mais n'a aucune interactivité.

Vous pouvez construire `top-down` ou `bottom-up`. Autrement dit, vous pouvez commencer par la construction des composants plus haut dans la hiérarchie ou avec celles des plus faibles.

A la fin de cette étape, vous aurez une bibliothèque de composants réutilisables qui rendent votre modèle de données.

Les composants définieront seulement les méthodes `render()` car cela est une version statique de votre application.

Étape 3: Identifier la représentation minimale (mais complète) de l'état d'interface

> *Pour rendre l'interface utilisateur interactive, vous devez être en mesure de déclencher des changements à votre modèle de données sous-jacent.*

Pour construire votre application correctement, vous devez d'abord penser à l'ensemble minimal de l'état mutable de vosapplications.

Pensez à tous les éléments de données dans notre exemple d'application :

- La liste initiale des produits.
- Le texte de recherche l'utilisateur a entré.
- La valeur de la case à cocher.
- La liste filtrée des produits.

> Il suffit de poser trois questions sur chaque élément de données:

1. **Est-il passé dans d'un parent par l'intermédiaire d'une `prop` ?**

- Si oui, il n'est probablement pas un état.

2. **Est-ce qu'il reste inchangé au fil du temps ?**

- Si oui, il n'est probablement pas un état.

3. **Pouvez-vous le calculer sur la base de tout autre État ou des `prop` dans votre composant ?**

- Si oui, n'est pas un état.

*La liste initiale des produits est transmise comme des `prop` , de sorte que ce n'est pas état.

Le texte de recherche et la case à cocher semblent être l'état car elles changent au fil du temps et ne peuvent pas être calculés à partir de rien.

Et enfin, la liste filtrée des produits ne sont pas des états, car ils peuvent être calculés en combinant la liste initiale des produits avec le texte de recherche et la valeur de la case à cocher.*

Finalement, les états sont:

- Le texte de recherche que l'utilisateur a entré.
- La valeur de la case à cocher.

Étape 4: déterminer où votre État doit vivre

Pour chaque état dans votre application:

- Identifier tous les composants qui ont un rendu basé sur cet état.
- Trouver un composant propriétaire commun (un seul composant au-dessus de tous les composants qui ont besoin de l'état dans la hiérarchie).
- Soit le propriétaire commun ou un autre composant plus haut dans la hiérarchie doit posséder l'état.
- Si vous ne pouvez pas trouver un composant où il est logique de posséder l'état, créer un nouveau composant simplement pour maintenir l'état et l'ajouter quelque part dans la hiérarchie au-dessus du

composant commun du propriétaire.

Etape 5: Ajouter un flux de données Inverse.

React rend le flux de données explicites pour le rendre facile la compréhension comment du programme.

"Component Data Flow" : propriétaire, enfants et création dynamique.

Dans React, vous pouvez créer des composants distincts qui encapsulent le comportement dont vous avez besoin.

Ensuite, vous pouvez rendre seulement certains d'entre eux, en fonction de l'état de votre application.

> Tenez compte de ces deux éléments:

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

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

Nous allons créer un composant `Greeting` qui affiche l'un de ces composants si un utilisateur est connecté ou non:

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

ReactDOM.render(
  // Try changing to isLoggedIn={true}:
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

Essayez-le sur CodePen.

Cet exemple rend un message d'accueil différent en fonction de la valeur de `prop isLoggedIn`.

Variables Element

Vous pouvez utiliser des variables pour stocker des éléments.

> Considérez ces deux nouveaux composants représentant `Logout` et `Login`:

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

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

Dans l'exemple ci-dessous, nous allons créer un composant appelé `LoginControl`.

Il rendra soit `<LoginButton/>` ou `<LogoutButton/>` en fonction de son état actuel et `<Greeting/>` de l'exemple précédent:

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

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

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

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

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

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
```

```
          {button}
        </div>
      );
    }
  }

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

Essayez-le sur CodePen.

`Inline If` avec l'opérateur logique `&&`

Vous pouvez intégrer toutes les expressions dans JSX en les enveloppant dans des accolades.

```
  function Mailbox(props) {
    const unreadMessages = props.unreadMessages;
    return (
      <div>
        <h1>Hello!</h1>
        {unreadMessages.length > 0 &&
          <h2>
            You have {unreadMessages.length} unread messages.
          </h2>
        }
      </div>
    );
  }

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

Essayez-le sur CodePen.

Inline If-Else avec l'opérateur conditionnel

Une autre méthode pour rendre conditionnellement éléments en ligne est d'utiliser l'opérateur conditionnel JavaScript.

Dans l'exemple ci-dessous, nous l'utilisons pour rendre conditionnellement un petit bloc de texte.

```
  render() {
    const isLoggedIn = this.state.isLoggedIn;
    return (
```

```
      <div>
        The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
      </div>
    );
  }
```

Il peut également être utilisé pour des expressions plus grandes:

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

Prévention des composants de rendu

Dans de rares cas, vous voudrez peut-être cacher un composant. Pour ce faire, retournez `null` au lieu de sa sortie de rendu.

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

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

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

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

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

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

Essayez-le sur CodePen.

Rendre plusieurs composants

Vous pouvez construire des collections d'éléments et les inclure dans JSX en utilisant des accolades `{}`.

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

Nous incluons l'ensemble `listItems` l'intérieur d'un élément `<ul>`:

```
ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

Essayez-le sur CodePen.

Liste de composants de base

> Habituellement, vous rendriez les listes à l'intérieur d'un élément.

Nous pouvons refactoriser l'exemple précédent dans un composant qui accepte un tableau de `numbers` et émet une liste non ordonnée d'éléments.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

> Lorsque vous exécutez ce code, vous recevrez un avertissement qu'une clé devrait être fournie pour les éléments de liste. Une «clé» est un attribut spécial que vous devez inclure lors de la création de listes d'éléments.

Assignons un `key` à nos éléments de liste à l'intérieur`numbers.map()` pour fixer la question clé manquante.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Essayez-le sur CodePen.

Clés

Les **clés** aident React à identifier les éléments qui ont changé, sont ajoutés ou sont supprimés.

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

La meilleure façon de choisir une clé est d'utiliser une chaîne qui identifie de manière unique un élément de liste parmi ses frères et sœurs.

> **il est recommandé d'utiliser des indetifiants spécifiques**

Lorsque vous ne disposez pas d'ID stables pour les articles rendus, vous pouvez utiliser l'index de l'élément comme une clé comme un dernier recours:

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

Extraction de Composants avec Clés

** Exemple: Correct **

```
function ListItem(props) {
  // Correct! There is no need to specify the key here:
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correct! Key should be specified inside the array.
    <ListItem key={number.toString()}
              value={number} />
  );
  return (
    <ul>
```

```
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Essayez-le sur CodePen.

Composants réutilisables : contrôle et transfert de propriétés.

React a un modèle de composition puissant, l'utilisation de la composition est recommandée à la place de l'héritage.

Confinement

Certains composants ne connaissent pas que leurs enfants à l'avance.

Ces composants utilisent `props.children` pour passer des éléments enfants directement dans leur rendu:

> `props.children` **est une propriété spéciale permettant de projeter du contenu dans un composant.**

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

Cela permet à d'autres composants de passer les enfants arbitrairement:

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

Essayez-le sur CodePen.

Bien que ce soit moins fréquent, parfois vous pourriez avoir besoin de plusieurs "`placeholder`" dans un composant. Dans de tels cas, vous pouvez définir votre propre convention au lieu d'utiliser `props.children`:

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

```
        </div>
        <div className="SplitPane-right">
          {props.right}
        </div>
      </div>
    );
}

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

Essayez-le sur CodePen.

Spécialisation

Parfois, nous pensons à des composants comme étant «cas spéciaux» d'autres composants. Par exemple, nous pourrions dire que `WelcomeDialog` est un cas particulier de `Dialog`.

Dans React, cela est également gérer par la composition, où un composant plus «spécifique» rend un autre plus «générique» et le configure avec des `props`:

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

function WelcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacecraft!" />
  );
}
```

Essayez-le sur CodePen.

Composition fonctionne aussi pour les composants définis comme des classes:

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

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

  render() {
    return (
      <Dialog title="Mars Exploration Program"
              message="How should we refer to you?">
        <input value={this.state.login}
               onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
          Sign Me Up!
        </button>
      </Dialog>
    );
  }

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

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}
```

Essayez-le sur CodePen.

A propos de l'héritage?

> *A Facebook, nous utilisons à des milliers de composants React, et on n'a pas trouvé de cas d'utilisation où nous vous recommandons de créer des hiérarchies d'héritage de composants.*

Contrôle des composants de formulaire.

Les éléments de formulaire HTML fonctionnent un peu différemment des autres éléments du DOM dans React, parce que les éléments de formulaire gardent naturellement un état interne.

Par exemple, ce formulaire en HTML accepte un seul nom:

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Un formulaire HTML possède un comportement par défaut (soummission).

Composants contrôlés

En HTML, les éléments de formulaire tels que `<input>`, `<textarea>` et `<select>` maintiennent généralement leur propre état et de le mettent à jour en automatiquement.

Dans React, l'état mutable est généralement maintenu dans la propriété de `state` des composants, et seulement mis à jour avec `setState()`.

Un élément de formulaire dont la valeur est contrôlée par React de cette façon est appelé «composant contrôlé».

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
```

```
          Name:
          <input type="text" value={this.state.value} onChange=
{this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Essayez-le sur CodePen.

La balise textarea

En HTML, un élément `<textarea>` définit son texte par ses enfants:

```
<textarea>
  Hello there, this is some text in a text area
</textarea>
```

Dans React, un `<textarea>` utilise un attribut `value` à laplace. De cette façon, une formulaire utilisant un `<textarea>` peut être écrit:

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <textarea value={this.state.value} onChange={this.handleChange}
```

```
/>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Notez que `this.state.value` est initialisée dans le constructeur, de sorte que la zone de texte commence avec un texte définit.

La balise select

`<select>` crée une liste déroulante. Par exemple, ce code HTML crée une liste déroulante de saveurs:

```
<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>
```

Notez que l'option de *coconut* est initialement sélectionnée, en raison de l'attribut `selected`. React, au lieu d'utiliser cet attribut `selected`, utilise un attribut `value` sur la balise `select` racine.

Ceci est plus commode dans un composant contrôlé parce que vous avez seulement besoin de le mettre à jour en un seul endroit.

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
```

```
          <label>
            Pick your favorite La Croix flavor:
            <select value={this.state.value} onChange={this.handleChange}>
              <option value="grapefruit">Grapefruit</option>
              <option value="lime">Lime</option>
              <option value="coconut">Coconut</option>
              <option value="mango">Mango</option>
            </select>
          </label>
          <input type="submit" value="Submit" />
        </form>
      );
    }
  }
```

Essayez-le sur CodePen.

Manipulation Entrées multiples

Lorsque vous avez besoin de gérer plusieurs éléments `input` , vous pouvez ajouter un attribut `name` à chaque élément et laisser la fonction de gestionnaire choisir ce qu'il faut faire sur la base de la valeur de `event.target.name`:

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked :
target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input
```

```
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Essayez-le sur CodePen.

Notez la syntaxe ES6 pour mettre à jour la clé de l'état correspondant au nom d'entrée donné :

```
this.setState({
  [name]: value
});
```

Il est équivalent à ce code ES5:

```
var partialState = {};
partialState[name] = value;
this.setState(partialState);
```

Alternatives aux composants contrôlés

Il peut parfois être fastidieux d'utiliser des composants contrôlés, car vous avez besoin d'écrire un gestionnaire d'événements pour tout changement de données.

> **Vous pouvez utiliser une `ref` pour lier une donnée provenant du DOM**

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleSubmit(event) {
```

```
    alert('A name was submitted: ' + this.input.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text"
          defaultValue="Bob" //specifies un default value
          ref={(input) => this.input = input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Manipulation du DOM.

> React met en œuvre un système de DOM indépendant du navigateur pour la compatibilité des performances et cross-browser. Nous en avons profité pour nettoyer quelques bords rugueux dans les implémentations navigateur DOM.

Dans React, toutes les propriétés DOM et attributs (y compris les gestionnaires d'événements) devraient être en notation CamelCase.

Par exemple, l'attribut HTML `tabindex` correspond à l'attribut `tabIndex` dans React. A l'exception des attributs `aria-*` et `data-*`, qui devraient être minuscule.

Différences dans les attributs

Il y a un certain nombre d'attributs qui fonctionnent différemment entre React et HTML:

Checked

L'attribut `checked` est pris en charge par les composants`<input>` de type `checkbox` ou `radio`. Vous pouvez l'utiliser pour définir si le composant est coché.

Ceci est utile pour la construction de composants contrôlés. `defaultChecked` est l'équivalent non contrôlé, qui définit si le composant est coché quand il est monté en premier.

className

Pour spécifier une classe CSS, utilisez l'attribut `className`. Cela vaut pour tous les éléments DOM et SVG réguliers comme `<div>`, `<a>`, et d'autres.

Si vous utilisez React avec des composants Web (ce qui est rare), utilisez `class` a la place.

dangerouslySetInnerHTML

`dangerouslySetInnerHTML` est de le remplacement pour l'utilisation de `innerHTML`.

Ainsi, vous pouvez définir HTML directement à partir React, mais vous devez taper `dangerouslySetInnerHTML` et passer un objet avec une clé `__html`, pour vous rappeler qu'il est dangereux.

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

htmlFor

Puisque `for` est un mot réservé en JavaScript, les éléments React utilisent `htmlFor` place.

style

L'attribut de `style` accepte un objet JavaScript avec des propriétés écrits en notation CamelCase plutôt que d'une chaîne de CSS.

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

Notez que les styles ne sont pas autoprefixed. Pour prendre en charge les navigateurs plus anciens, vous devez fournir des propriétés de style correspondantes:

```
const divStyle = {
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix
};

function ComponentWithTransition() {
  return <div style={divStyle}>This should work cross-browser</div>;
}
```

suppressContentEditableWarning

Normalement, il y a un avertissement quand un élément avec les enfants est également marqué comme `contentEditable`, parce que cela ne fonctionnera pas. Cet attribut supprime cet avertissement.

value

L'attribut `value` est pris en charge par les balises `<input>` et `<textarea>` . Vous pouvez l'utiliser pour définir la valeur du composant. Ceci est utile pour la construction de composants contrôlés.

`defaultValue` est sont équivalent non contrôlé, qui fixe la valeur du composant lorsqu'il est monté en premier.

Tous les attributs HTML pris en charge

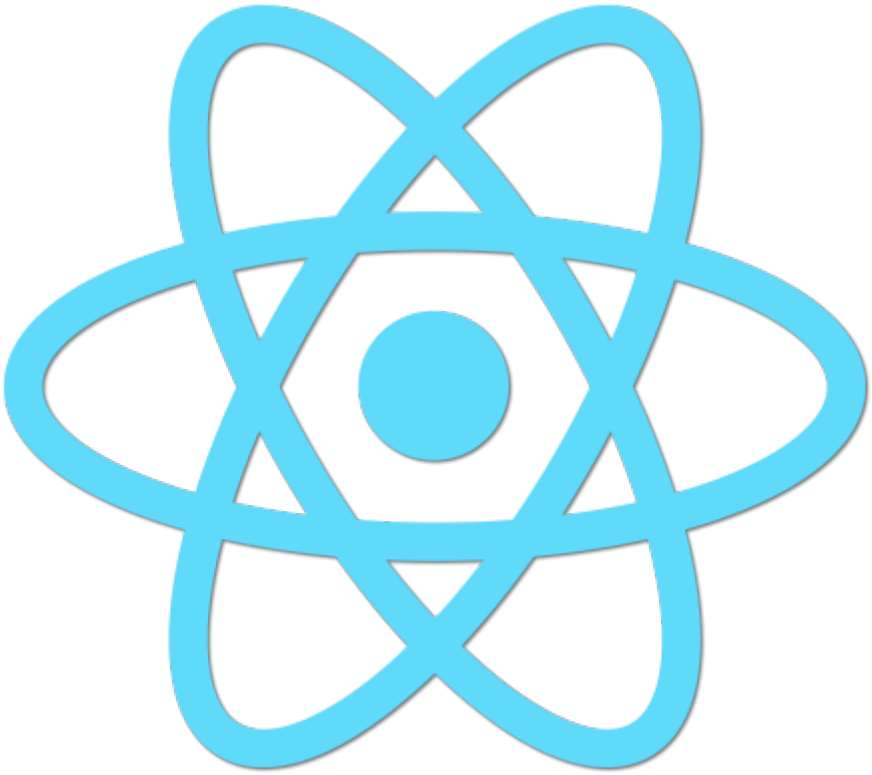
React supporte les attributs `data-*` et `aria- *` , ainsi que ces attributs

En outre, les attributs non standard suivants sont supportés:

- `autoCapitalize autoCorrect` Mobile Safari.
- `color` pour `<link rel="mask-icon"/>` dans Safari.
- `itemProp itemScope itemType itemRef itemID` pour les micro data.
- `security` pour les anciennes versions d'Internet Explorer.

- `unselectable` pour Internet Explorer.
- `results autoSave` pour les champs de type `search` de WebKit/Blink.

> Tous les attributs SVG pris en charge



Application monopage avec ReactJS et Flux ou Redux

Flux/Redux : présentation. Propagation de données.

Flux & Redux

Face au besoin continu d'ajout de nouvelles features, l'équipe frontend de Facebook s'est vite retrouvée bloquée à cause d'architectures Modèle-Vue-Contrôleur. Pour eux, le MVC n'est pas un pattern prévu pour scaler : la maintenabilité d'une application basée sur le design pattern MVC est laborieuse. Plus on ajoute de modèles, de contrôleurs et de vues, plus la complexité augmente.

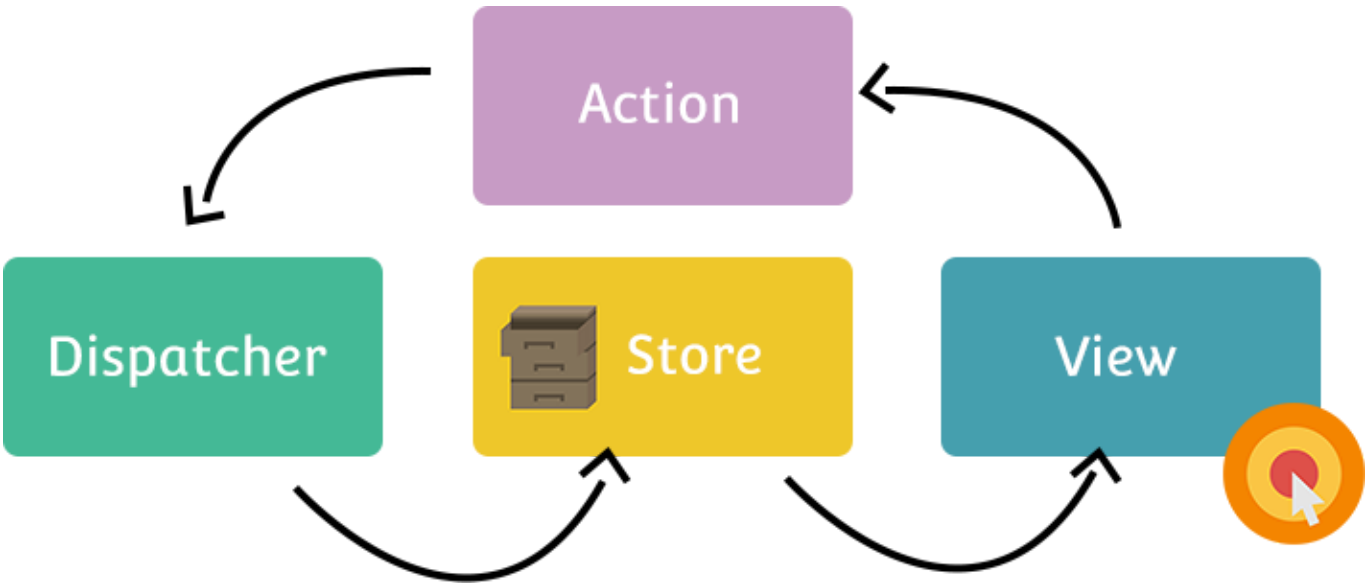
Pour surmonter cette difficulté, Facebook a créé **une « nouvelle » architecture appelée Flux**. Contrairement au MVC, celle-ci est une architecture garantissant (en théorie) un **flux unidirectionnel**, permettant à un développeur d'identifier rapidement le chemin critique d'un événement et ses conséquences.

Redux est une implémentation dérivée de Flux. Ça permet de créer un Store qui contient un état, réagit à des actions dispatchées, et auquel on peut souscrire pour être notifié des changements. Il permet également l'ajout de middlewares, qui peuvent en quelque sorte pre-process les actions.

Structurer les composants avec Flux

> Flux s'appuie sur plusieurs concepts :

- `action`;
- `dispatcher`;
- `store`;
- `view` (composants React).



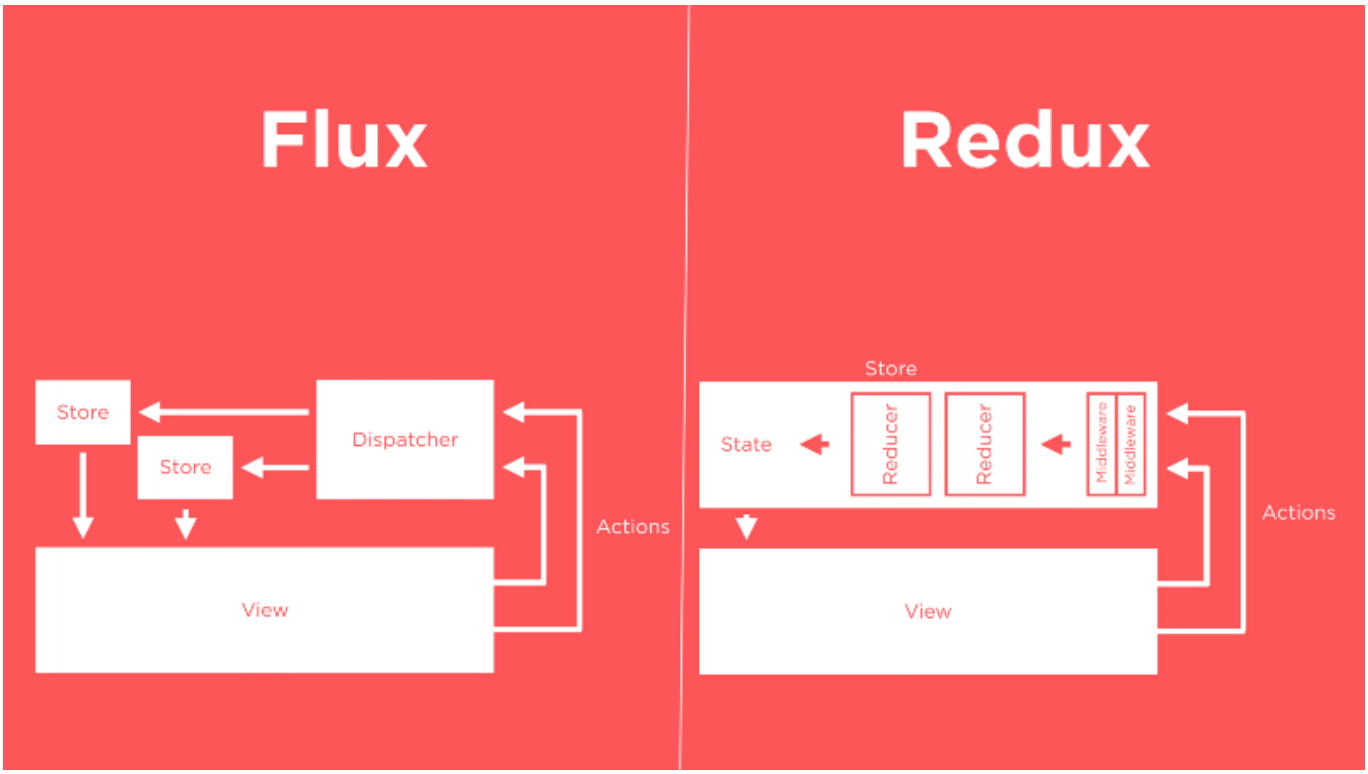
Chaque interaction de l'utilisateur sur la `view` déclenche une `action`, celle-ci passe ensuite dans l'unique `dispatcher` qui notifie les `store`. **Le store prévient le composant qu'il a été modifié et celui-ci se met à jour.**

Un store gère l'ensemble de données et la logique métier d'un domaine de l'application. Le `dispatcher` est quant à lui le **seul point d'entrée des actions**, ce qui permet de garder la main sur le code flow et de prévenir d'éventuels effets de bord.

Comparaison des architectures.

Dans **Redux, il n'y a qu'un seul `store` pour l'état de l'application.** Ce `store` est mis à jour par des `action` utilisant des `reducer`.

Dans Flux il ya plusieurs `store` qui peuvent avoir des dépendances internes. Cette petite différence rend plus facile de raisonner sur l'état en utilisant Redux.


Flux
Store
Store
Dispatcher
Actions
View
Redux
Store
State
Reducer
Reducer
Middleware
Middleware
Actions
View

Création de vues et contrôleurs dans Flux.

La principale idée de flux est de faire passer le moindre évènement de votre application au travers d'une boucle qui va parcourir tous vos stores (les éléments qui contiennent les données de votre application).

> Le code de Flux vous ne se compose en fait que d'un dispatcher (et de quelques helpers). Flux n'est pas du code mais plutôt une nouvelle façon de penser son code.

Avec Flux votre codebase se décompose de la façon suivante :

- `store`: l'endroit où votre modèle va être contenu.
- `action`: représentant toutes les actions possibles.
- `dispatcher` : unique, qui notifie les stores des actions effectuées.
- `view-controller`: qui transforme et affiche les données transmise.

Les vues

L'idée principale du `view-controller` est de regrouper ensemble le contrôleur et sa vue associée.

> Facebook distingue **deux type de vues différentes** dans Flux : **les views simples et les containers.**

Les containers

Les containers sont des `view-controller` un spéciaux : **ils écoutent les changements d'un store.**

L'idée du container est de centraliser les données relatives à une partie de l'application à un seul endroit.

Le container passera ensuite ces données à ses enfants pour affichage.

Pour reprendre le cas de notre application de TODOS on va avoir envie d'avoir un container qui à accès à l'utilisateur loggé et un autre qui s'occupe des TODOS. Ils écouteront respectivement les userStore et todosStore.

```
React.createClass({
    //...
    componentWillMount: function () {
         // à la création du composant, on enregistre le listener
         userStore.addListener(this.onChange);
    },

    componentWillUnmount: function () {
         // à la suppression du composant, on retire le listener
        userStore.removeListener(this.onChange);
    },

    //ce callback est appelé à chaque fois que le userStore change
    onChange: function () {
        this.setState({
            user: userStore.get()
        });
```

```
    },

    render: function () {
        //on a ici la dernière valeur du user
        return (<App user={this.state.user}/>);
    }
});
```

Les view-controller

Les `view-controller` ne sont là que pour transformer les données brutes et les afficher. **Il reçoivent donc toutes leurs données via des propriétés et sont censés être complètement stateless.**

> On retrouve, avec les vues, un concept propre à la programmation fonctionnelle : une fonction retournera toujours le même résultat si on lui donne les mêmes paramètres. On appelle ça les fonctions pures.

D'ailleurs avec la nouvelle syntaxe de composant que React v0.14 a introduit récemment, vous pouvez déclarer un composant comme étant une simple fonction qui prend en paramètre des props et retourne du JSX. Avec cette syntaxe, pas de state et on est alors obligé de faire un composant stateless.

```
var App = function (props) {
    var user = props.user;
    return (
        <div className="header">
            <span>{user.firstname}</span>
            <span>{user.lastname}</span>
            <LogoutButton user={user} />
        </div>
        <TodoContainer />
    );
}
```

Rôle du "Dispacther" dans Flux pour les actions.

Le `dispatcher` est là pour faire transiter absolument tout ce qu'il se passe sur l'application (évènements) par les `store`.

> **Lors du bootstrap de l'application tous les stores devront donc être enregistrés auprès de ce dispatcher unique.**

Les "Stores", gestionnaire d'états logique dans Flux.

> **Les stores sont en fait une représentation complète de l'état, à un instant donné, de l'application. Il ne doit y avoir aucun de vos modèles qui vit en dehors d'un store.**

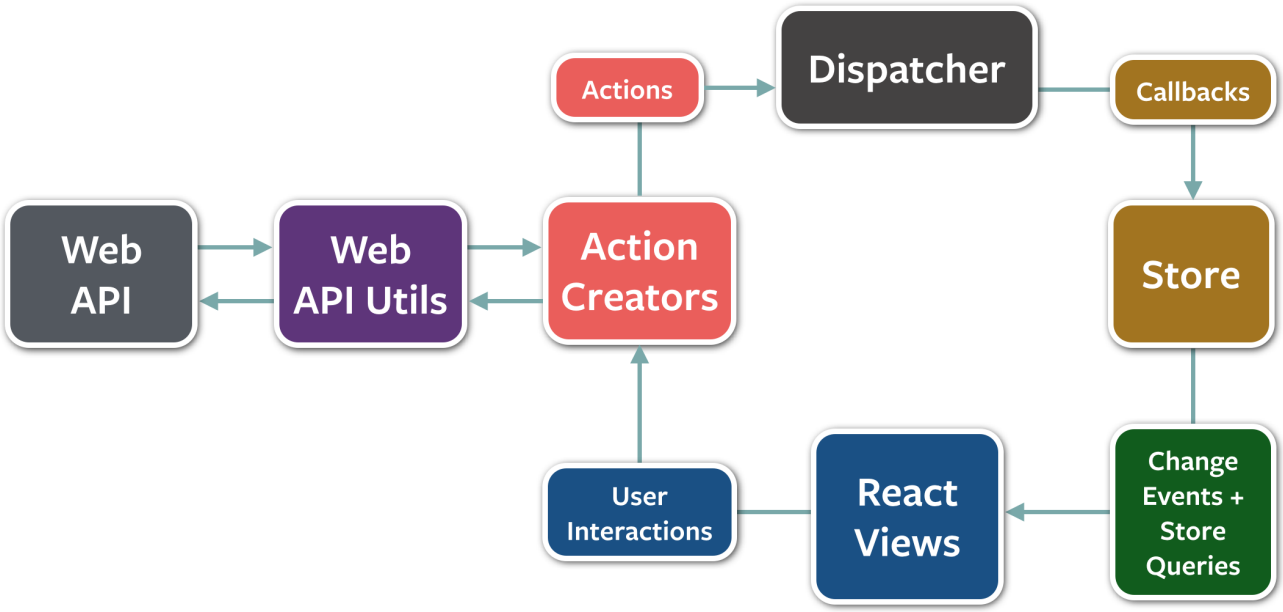
Si vous vous pliez à cette règle, il vous suffira de faire une sauvegarde des données de vos stores et la recharger plus tard pour pouvoir retrouver l'application dans l'état exact dans laquelle vous l'aviez laissée.

> Un `store` se comportent comme un modèle observable (il dispose de *getters* et d'une méthode *addListener*) à la différence près qu'il n'a pas de setter.
>
> **Seul le `store` peut mettre à jour ses propres données.**

```
var events = {
    //quand l'application reçoit une liste de todos du serveur (ou
d'ailleurs)
    RECEIVED_TODOS: "RECEIVED_TODOS",

    //quand l'utilisateur ajoute un nouveau todo
    TODO_ADDED: "TODO_ADDED",

    //quand les todos ont été sauvegardés sur le serveur
    TODOS_SAVED: "TODOS_SAVED"
    //[...] plein d'autres évènements que nous ne traiterons pas ici
};

//l'instance unique de notre dispatcher de l'application
var appDispatcher = require("../dispatcher/appDispatcher"),
    //les évènements définis précédemment
    events = require("./events");

//nos todos. inaccessibles depuis l'extérieur et vide pour le moment
//c'est une action qui viendra remplir tout ça
var todos = [];

var TodosStore = {
    //l'unique méthode accessible depuis l'extérieur
    //qui nous retourne simplement les todos
    get: function () {
        return todos;
    }

    //ps il faut rajouter ici les méthodes d'ajout/suppresion
    //d'event listeners (addListener/removeListener)
    //qui serviront pour prévenir les vues que quelque chose
    //a changé
});

//c'est ici que la magie s'opère. On enregistre un callback qui sera appelé
dès que quelque chose se passe
appDispatcher.register(function (payload) {
```

```
    //dans le payload on a tous les détails de l'évènement (type et données
qui va avec)
    var action = payload.actionType,
        data = payload.data;

    //on répond uniquement aux évènements qui nous intéressent
    switch (action) {
        //quand on reçoit les todos, on remplace simplement nos todos
        //par les todos reçus
        case events.RECEIVED_TODOS:
            todos = data.todos;
            this.emitChange(); //stay tuned, on parle de ça bientôt
            break;

        //quand un todo est créé sur le serveur on l'ajoute dans notre
        //liste de todos
        case events.TODO_ADDED:
            todos.push(data.todo); //on ajoute la nouvelle todo dans la
liste
            this.emitChange(); //stay tuned, on parle de ça bientôt
            break;

        //par défaut, on ne fait rien
        //vous noterez par exemple qu'on ne répond pas ici
        //à l'évènement TODOS_SAVED car il n'aurait aucune influence
        //sur les données brute de ce store
        default:
            break;
    }
}.bind(todosStore));

module.exports = todosStore;
```

Flux en résumé



- **Les stores contiennent toutes les données brutes de l'application ;**
- Les stores représentent l'état de l'application à l'instant donné et sont donc complètement synchrone ;
- Seuls les stores peuvent modifier leurs données ;
- **Les stores réagissent aux évènements du dispatcher ;**
- Les stores sont comme **un modèle observable mais sans setter** ;
- Les vues écoutent les changements d'un (ou plusieurs) store(s).
- **Une vue qui est reliée à un store est appelée "container"**.
- Le dispatcher informe les stores.
- Les actions sont là pour appeler le dispatcher et **éventuellement des services externes (API, localStorage ...) ;**

> Un simple Flux Flow

Définition du Functionnal Programming.

> La programmation fonctionnelle est un paradigme de programmation de type déclaratif qui considère le calcul en tant qu'évaluation de fonctions mathématiques.

Comme le changement d'état et la mutation des données ne peuvent pas être représentés par des évaluations de fonctions la programmation fonctionnelle ne les admet pas, au contraire elle met en avant l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état.

Approche avec Redux. Le "Reducer".

Redux est un "state container" crée pour être totalement prévisible, et donc de pouvoir simplifier des tests et le debug. Pour cela, il se base sur une philosophie très stricte et rigide **basé sur 3 principes :**

> - Les **données** dynamiques de l'application **existent en un seul endroit.**
> - *Modifier les données de l'application ne peut se faire qu'à travers une action, et les actions ne peuvent faire autre chose que modifier les données de l'application.*
> - Modifier les données crée une nouvelle "version" des données.

Ça permet de créer un Store qui contient un état, réagit à des actions dispatchées, et auquel on peut souscrire pour être notifié des changements.

Il permet également l'ajout de middlewares, qui peuvent en quelque sorte pre-process les actions.

Utilitaire Redux

> Redux reprend les concepts de Flux mais en simplifiant beaucoup le processus de développement. Cette simplification est en partie due au fait que Redux utilise des concepts liés à la programmation fonctionnelle et s'inspire d'Elm pour changer l'état de l'application.

> Redux en exemples

```
import { createStore } from 'redux'

/**
 * Le reducer est une fonction dite "pure" ayant (state, action) => state
comme signature.
 * Il va décrire comment une action transforme le state (l'état) de
l'application
 * en un nouvel état.
 *
 * L'implémentation de l'état de l'application dépend totalement de votre y
* use case et peut être une primitive, un tableau, un objet, ou bien même
 * une structure de données
 * immutable (basé sur Immutable.js par exemple).
 * La seule chose à retenir est que cette partie ne DOIT PAS modifier
 * l'objet correspondant à l'état de l'application lorsque l'état change.
 * Dans cet exemple, on utilise un switch et des strings, mais on pourra *
très bien utiliser un helper qui va suivre une autre manière de faire.
 */
```

```
function counter(state = 0, action) {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1
  case 'DECREMENT':
    return state - 1
  default:
    return state
  }
}

// On crée un Redux store, qui va garder l'état de notre app.
// L'api correspond à trois fonctions { subscribe, dispatch, getState }.
let store = createStore(counter)

// On peut s'abonner manuellement ou bien lier l'état à une vue
automatique// ment à l'aide du binding.
store.subscribe(() =>
  console.log(store.getState())
)

// Le seul moyen de modifier l'état de l'application est de dispatcher des
actions.
// Les actions peuvent être serialisées, loggées ou sauvegardées pour plus
tard.
store.dispatch({ type: 'INCREMENT' })
// 1
store.dispatch({ type: 'INCREMENT' })
// 2
store.dispatch({ type: 'DECREMENT' })
// 1
```

Plutôt que de modifier l'état de l'application directement, on spécifie les modifications qui peuvent arriver avec de simple objets appelés actions.

Puis on écrit une fonction appelée reducer, qui se chargera de décider comment chaque action transforme l'état de l'application.

DIFFÉRENCES AVEC FLUX ?

> - Redux n'a pas de dispatcher
> - Redux n'a pas la possibilité de définir plusieurs Stores.

A la place, nous avons un `store` unique avec un seul `reducer`. Au fur et a mesure que l'application se complexifie, l'unique reducer va être découpé en plusieurs petit `reducer` indépendants.

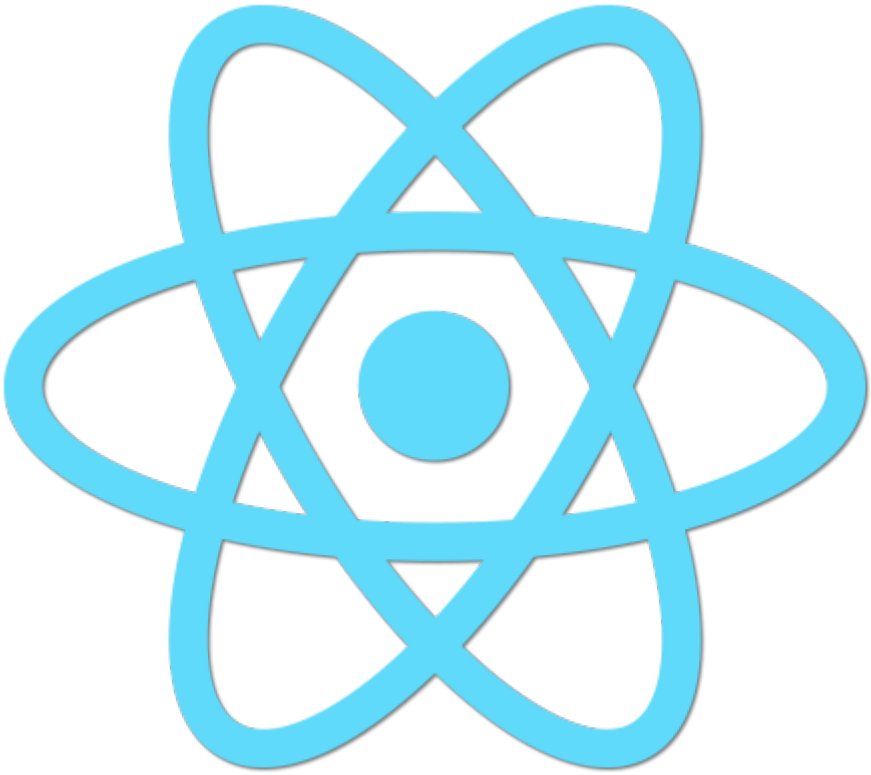
Extension pour ReactJS : "hot-loader".

React Hot Loader permet la modification à chaud d'un composant React.

Le **hot-reload** est rendu possible par les fonctionnalités conjugués de 3 composantes:

- **Webpack**, qui permet le remplacement à chaud de n'importe quel module CommonJS (HMR).
- **react-hot-loader**, qui intervient en cas de modification de composant React, il agit comme un proxy sur les méthodes du composant.
- **React** lui-même, qui propose une fonction de rendu pure : le render ne dépend que des propriétés en entrée (props / state). De fait, le remplacement de cette fonction par react-hot-loader permet à lui seul d'obtenir le nouveau rendu.

React Hot Loader



Application isomorphique

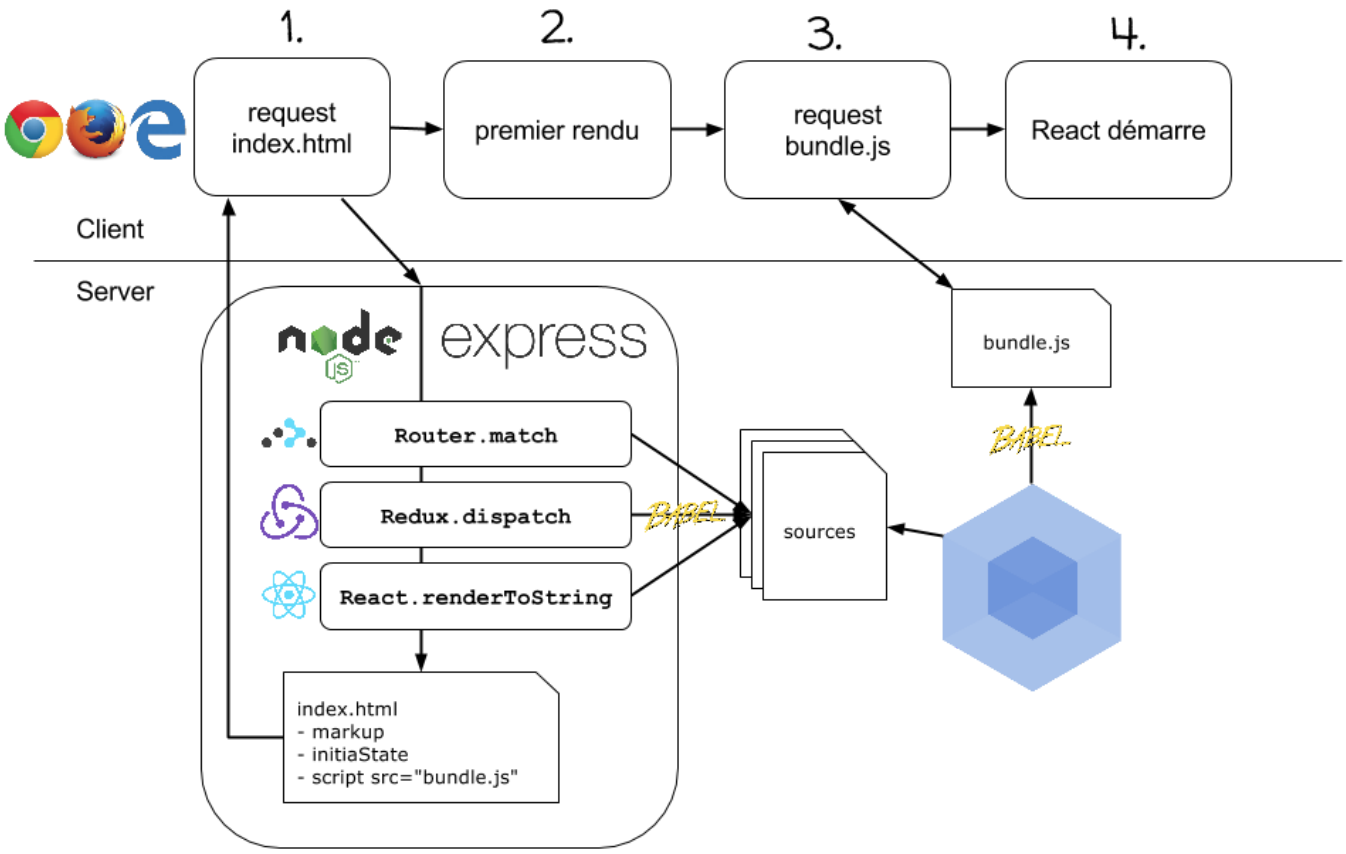
Application isomorphique

> **La performance**

Cycle de d'initialisation d'une SPA coté client :

1. Chargement du fichier HTML

- Chargement des différents Assets (Css, image, scripts JS application et librairies/frameworks)
- Délai : Parsing / Execution.
- Délai : Démarrage et configuration de l'état.



Principe et bénéfices du développement isomorphique.

Le mot *isomorphisme* vient des racines grecques *isos* pour égal et *morph* pour forme. L'isomorphisme désigne donc **deux entités de même forme dans un contexte différent.**

En informatique, et plus particulièrement dans le domaine du développement web, on dit qu'une **application est isomorphique lorsqu'elle partage le même code coté client (navigateur) et coté serveur.**

Concrètement, les avantages de l'isomorphisme sont multiples:

- Un seul code, pour toute une application (serveur et client)
- Les moteurs de recherche voient le contenu (plus totalement vrai: Google interprète le JavaSscript)
- Rapide, plus nécessaire d'attendre le téléchargement du JS pour voir la page.

- Plus facile à maintenir (une seule codebase)
- Un état identique (partagé) entre client et serveur = un debug plus simple.

> Cependant, mettre en place de l'isomorphisme "from-scratch" n'est pas simple, c'est pourquoi il est souvent nécessaire d'utiliser un framework qui supporte ce mode de fonctionnement.

React et l'isomorphisme

Voici un exemple d'un code en JSX et de son équivalent en utilisant la notation classique de React (en ES6).

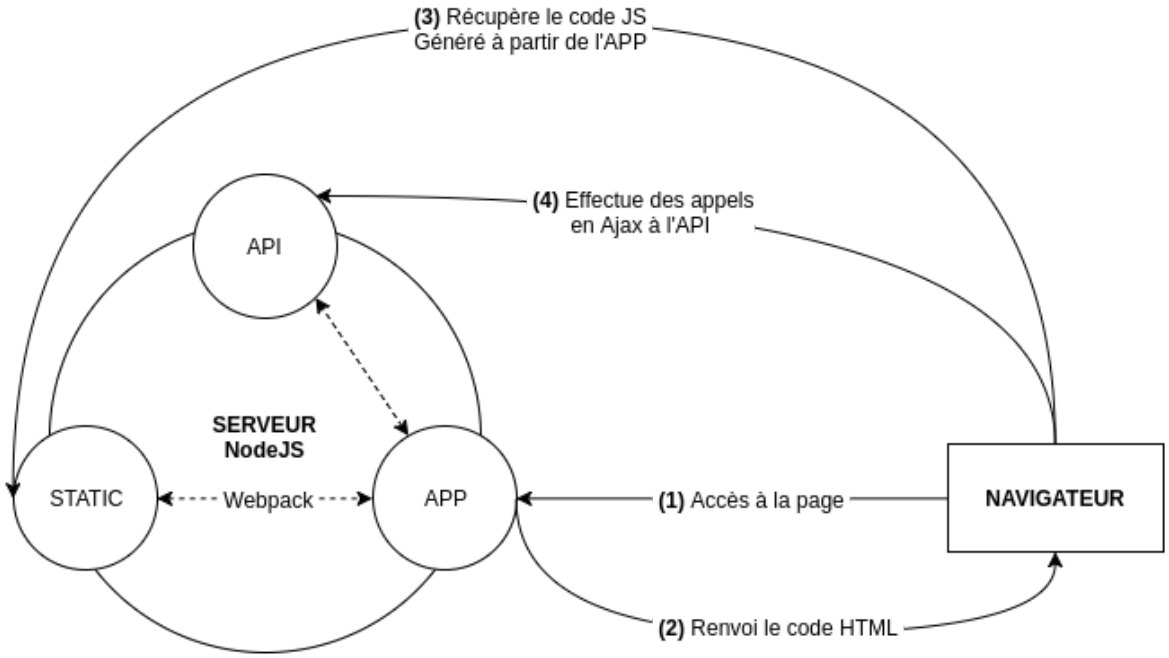
> Il est possible d'utiliser 3 notations bien distinctes.

```
// Création de l'élément en JSX
class Button extends React.Component {
    render() {
        return (
            <button className={"button"}>
              <b>OK!</b>
            </button>
        );
    }
}

// Création de l'élément à l'aide des "helpers" React
class Button extends React.Component {
    render() {
        return React.createElement("button", { className: "button" },
            React.createElement("b", {}, "OK!")
        )
    }
}

// Création de l'élément en "brut" (objet JS)
class Button extends React.Component {
    render() {
        return {
            type: 'button',
            props: {
                className: 'button',
                children: {
                  type: 'b',
                  props: {
                    children: 'OK!'
                  }
                }
            }
        };
    }
}
```

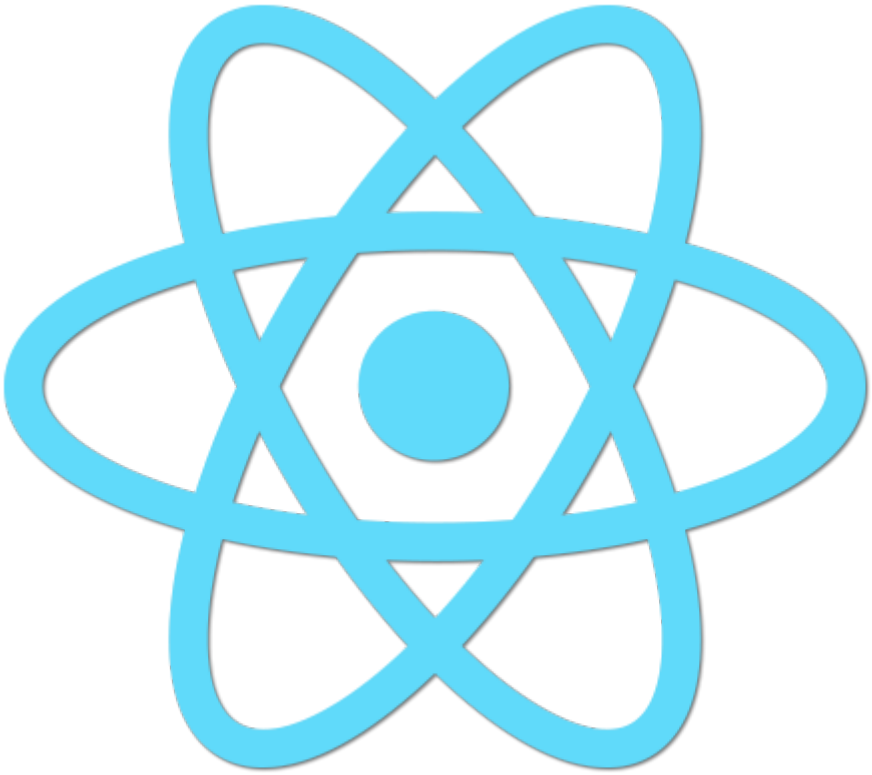
Ecosystème du JavaScript côté serveur.

L'utilisation d'un "DOM" virtuel permet à React d'être totalement "context agnostic", ce qui lui permet de générer un rendu coté serveur

Workflow d'application isomorphique.



Examen du code source



Introduction à React Native

Introduction à React Native

React Native est un framework mobile hybride développé par Facebook depuis début 2015. Il continue d'évoluer avec le soutient de nombreux contributeurs sur Github.

Facebook a présenté la solution de sa Keynote en 2015.

"Learn once, write everywhere"

Le but de React Native est de pouvoir réutiliser le maximum de code entre les différentes plateformes (iOS et Android). Il offre un gain de temps considérable par rapport à du développement spécifique, tout en étant aussi performant.

L'écriture en javascript permet aux développeurs web de construire une application mobile native, contrairement à Cordova qui encapsule l'application dans une webview.

React Native utilise le moteur JavaScriptCore avec le transpileur Babel, il est compatible ES5, ES6 ou ES7.

Positionnement, différences avec Cordova.

À l'instar de Cordova, il existe de nombreux plugin React-Native. Beaucoup sont encore à l'état d'expérimentation, surtout pour Android.

Un outil permet d'automatiser l'installation de plugins : React Node Module Packager (rnpm). Par contre l'approche diffère grandement d'un projet **cordova.**

Spécificités par rapport au développement pour le web

> Pas de DOM

Inutile de faire appel à window ou document, React Native n'utilise pas de DOM. Attention donc à la compatibilité de certaines librairies JS.

> Pas de balises HTML

Les tags spécifiques au DOM ne sont donc pas admis non plus (`<div>`, `<section>`, `<article>`,...). L'interface doit être construite à partir des composants React-Native uniquement (ou de composants ppersonanlisés) : `<View>`, `<Text>`, `<Image>`,...)

Voir la liste complète dans la documentation

> Tout doit être packagé

Toutes les ressources doivent être appelée, soit par un chemin absolu, soit par `require : {{uri: urlDeMonImage}} ou {require('./chemin/de/mon/image')}`.

> Styles

On déclare le style d'un composant avec l'attribut style, puis on crée un objet StyleSheet pour les définir.

```
const Picture = React.createClass({
    render: function() {
        return (
            <View style={styles.container}>
            <Image source={{uri: this.props.url}} />
            <Text>Ceci est une légende</Text>
            </View>
        );
    }
});
var styles = StyleSheet.create({
    container: {
        flex: 1
    },
});
```

De React aux composant iOS natifs.

Procédure de démarrage :

```
brew install node
brew install watchman

npm install -g react-native-cli
```

Prérequis

- OS X, Linux ou Windows peuvent développer pour Android
- OS X et Xcode (> 7.0) sont nécessaire pour iOS

- Git, Node et npm
- Android JDK et SDK
- Watchmen pour OS X ou Linux

Camera

- react-native-camera

Cartographie

- react-native-maps (Google Maps pour Android, Plans pour iOS)
- React-Native MapboxGL (Android encore mal supporté)

Système de fichier

- react-native-fs