

EcmaScript 6/2015 à TypeScript, maîtriser la Programmation Orientée Objet



Intervenant :



Renaud Dubuis

[View profile](#)

 LinkedIn®

Approche pédagogique.

Participants. (Tour de table)

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

Evaluation des pré-requis.

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

Récapitulatif (matinal).

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

Concertation personnelle.

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

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

Travaux Pratiques.

L'acquisition des concepts abordés est découpée proportionnellement

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

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

1. Enoncé > 2. Démonstration > 3. Manipulation

Version numérique du support de formation.

Pour renforcer le confort de lecture et de manipulation (**copier/coller, liens cliquables, coloration du code**) le support est également distribué en version numérique. **au format PDF.**

Références à l'ouvrage et autres références

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

Autres références

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

Présentation modèle de l'intervenant. Présentation des participants.

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

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

Exemple

(identité) *Bonjour je suis Renaud Dubuis.*

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

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

(précision) Etant dyslexique, je parfois des soucis d'orthographe, je vous prie de m'en excuser. Je tutoie également les développeurs avec qui je code, merci de me dire si vous préférez le vouvoiement.

Presentation personnelle

identité :

rôle :

pré requis: (HTML, CSS, JavaScript)

attente(s):

précision:

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

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

9:00 : Début de formation.

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

Pause déjeuner: (choisir un horaire et une durée)

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

17:00 : Temps des questions spécifiques.

17:30 : fin de journée.

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

Les questions posées font soit:

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

Introduction du modèle de formation :

L'acquisition des concepts abordés est découpée proportionnellement :

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

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

1. Ennoncé > 2. Démonstration > 3.Manipulation.

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

2. Démonstration Le formateur illustre pratiquement le point expliqué.

Durant ces deux premières phases il est recommandé de :

Se concentrer sur le concept abordé. Prendre des notes.

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

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



Présentation de la structure du support de formation :

Dans la mesure du possible le support se décompose en trois points.

1. Théorie > 2. Exemple > 3. Validation.

1. Théorie Explications théoriques internes au support ou basées sur une documentation en ligne.

Pourquoi utilisée une documentation en ligne :

Les solutions de développement telles que **ionic** évoluent très rapidement pour répondre au besoins techniques du marché et aux attentes des développeurs.

Les traductions peuvent souvent être approximative et très rapidement obsolètes. Beaucoup de solutions et de langages sont créés en langues anglaises **les concepts y sont souvent plus aisément présentés.**

Votre formation s'appuiera sur l'utilisation, la compréhension et l'usage de la documentation en ligne

2. Exemple Exemple local ou illustré en ligne.

3. Validation Espace de validation basé sur des questions écrites ou orales.



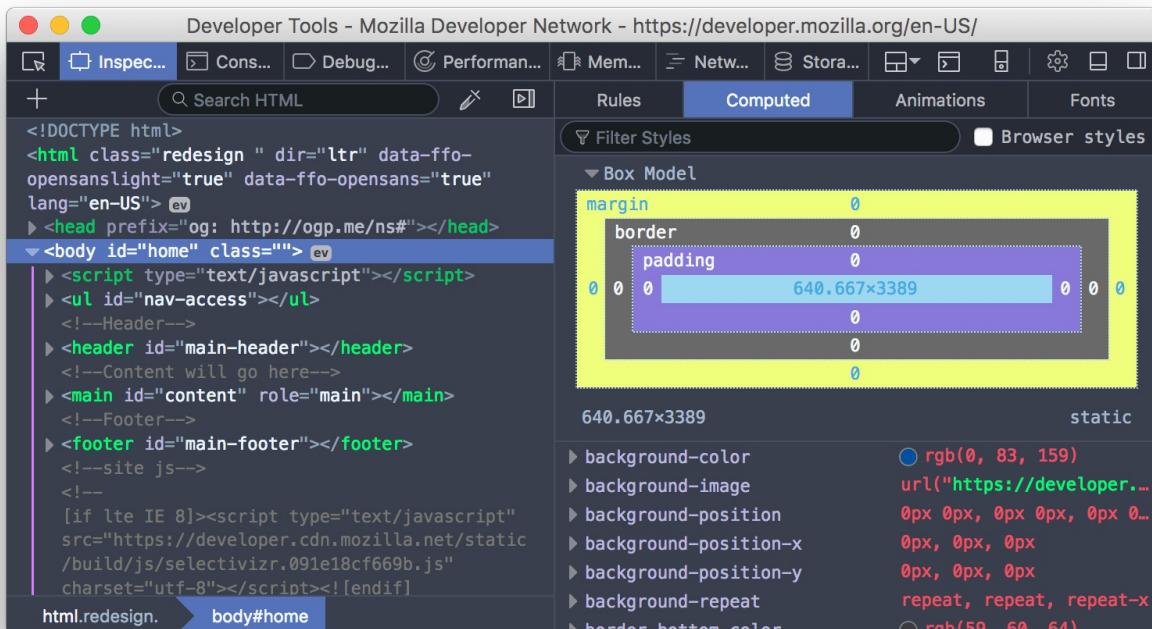
Configurer un environnement de développement moderne.

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

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

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

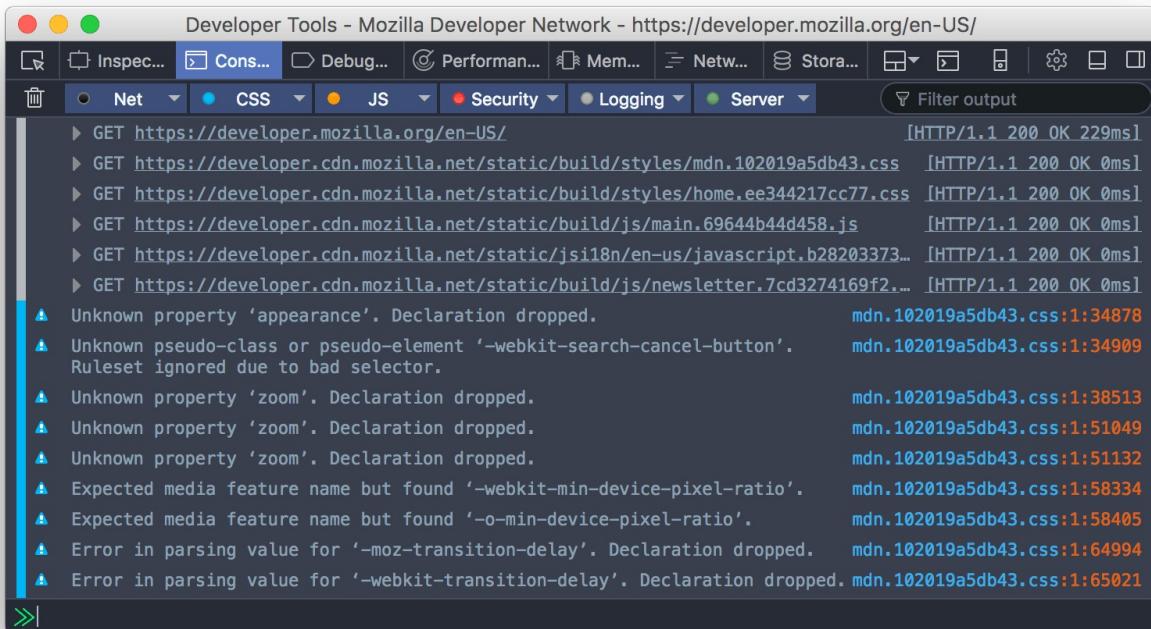
Inspecteur



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

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

Console Web



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

Débogueur JavaScript

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

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

Réseau

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

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

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

Outils indispensables pour le développeur.

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

Il est possible de classer les outils selon trois catégories :

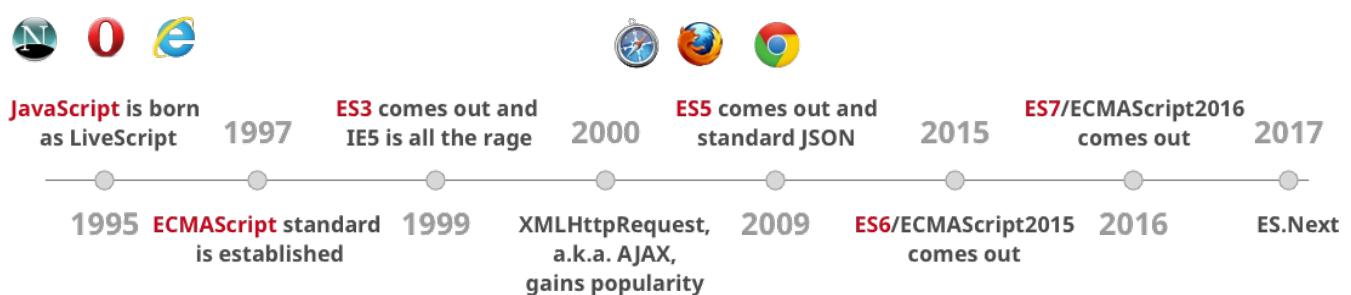
- **Logique** Dépendances tiers , tel que les différents frameworks.
- **Technique** ayant un impact sur la plate-forme de développement pour l'automatisation et le contrôle des tâches par exemple.
- **Productivité** toute autre solution non indispensable permettant d'accélérer le de développement, allant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

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

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

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

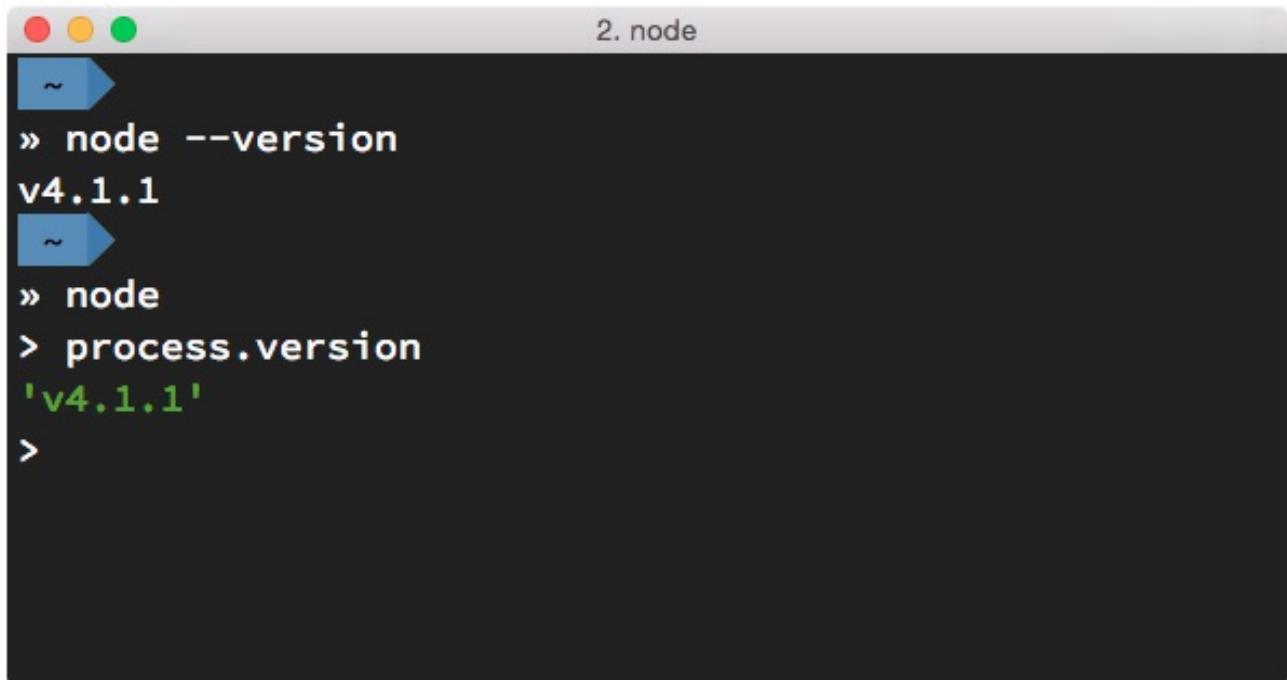
Prendre en considération l'évolution du langage



Node.js utilitaire de développement.

Installation de Node.JS :

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

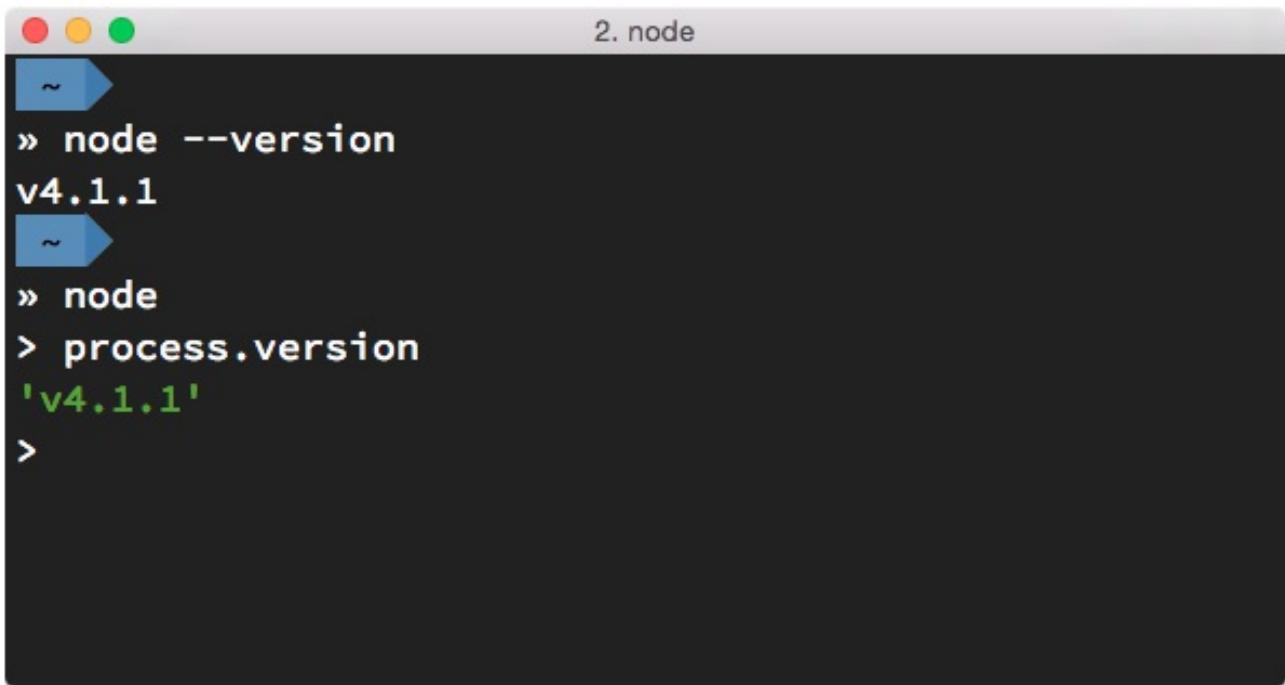


```
2. node
~ » node --version
v4.1.1
~ » node
> process.version
'v4.1.1'
>
```

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

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

Aisance avec l'invite de commande windows.



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

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

Manipulation 1

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

Manipulation 2

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

Manipulation 3

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

Environnement de développement. IDE et plug-ins.

Outils de développement les autres logiciels :

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

Nous utiliserons VS Code.

cf. pratique

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

Vérifier des installations :

cf. pratique

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

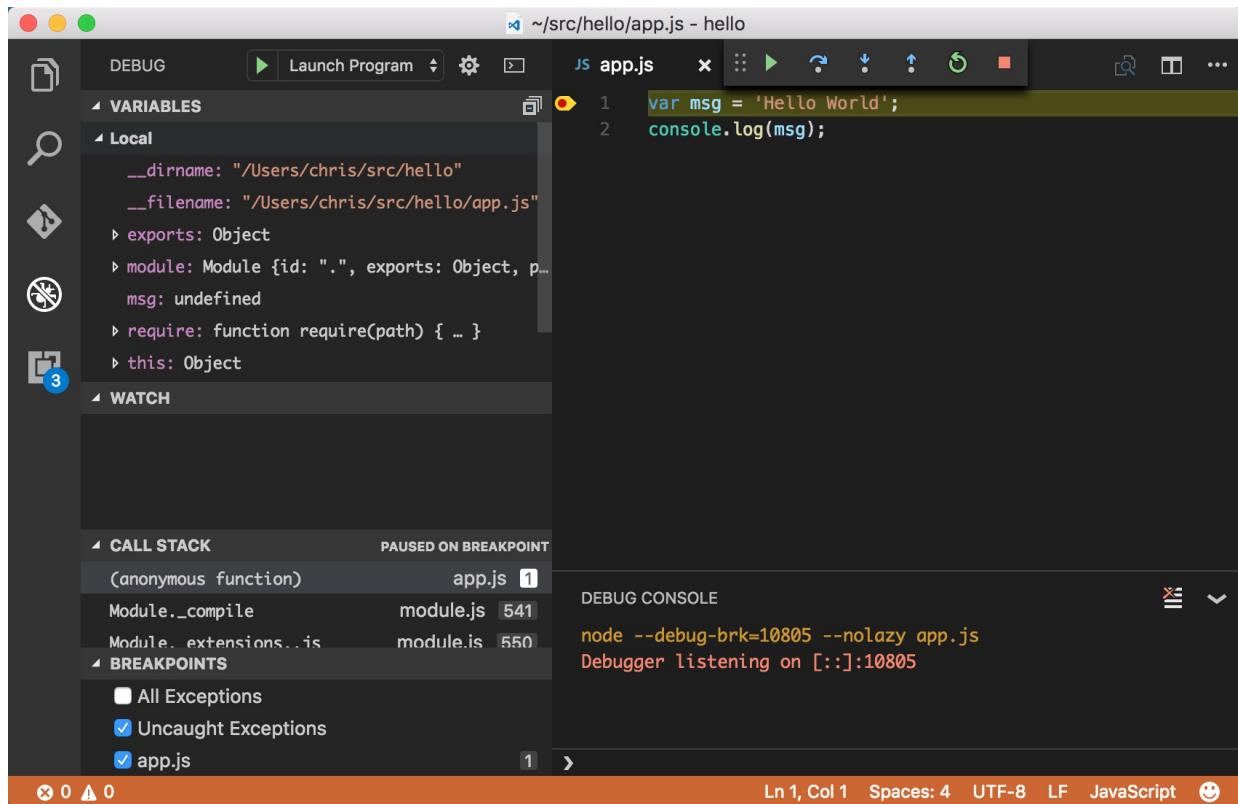
À noter Pour travailler avec les outils de l'ecosystem Angular2 il faut une version de **Node 4+ et NPM 3+**

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

✓ Installation réussie !

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

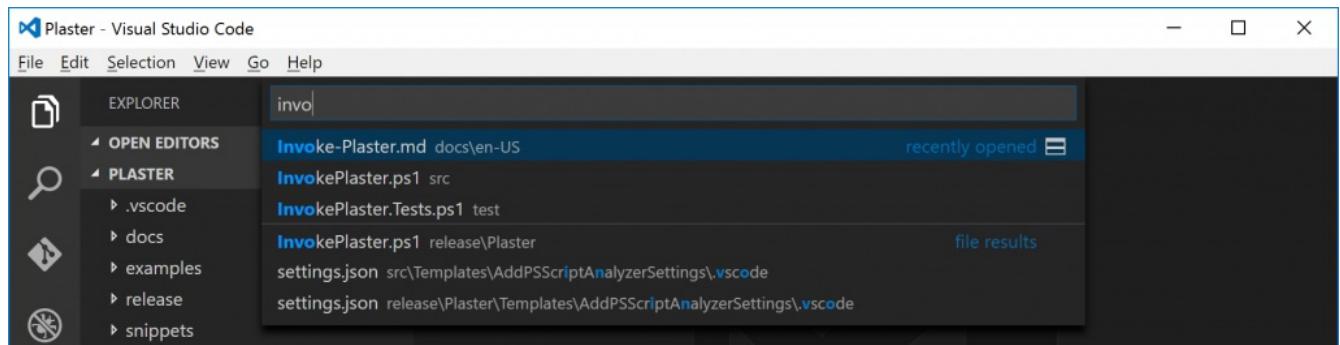
VS Code apporte toutes les fonctionnalités d'un éditeur moderne, sans nécessiter un investissement de formation.



Fonctionnalités principales:

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

La palette de commande



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

- Utilisation des commandes
- Saisie intuitive

Installer des modules tiers

Il existe deux modes d'installation avec npm :

global (machine)

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

local (dossier courant)

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

Installation des modules

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

Initialisation

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

`npm init`

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

La commande démarre une série de questions interactives.

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

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

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

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

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

Structure de projet

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

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

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* : artefacts 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
```

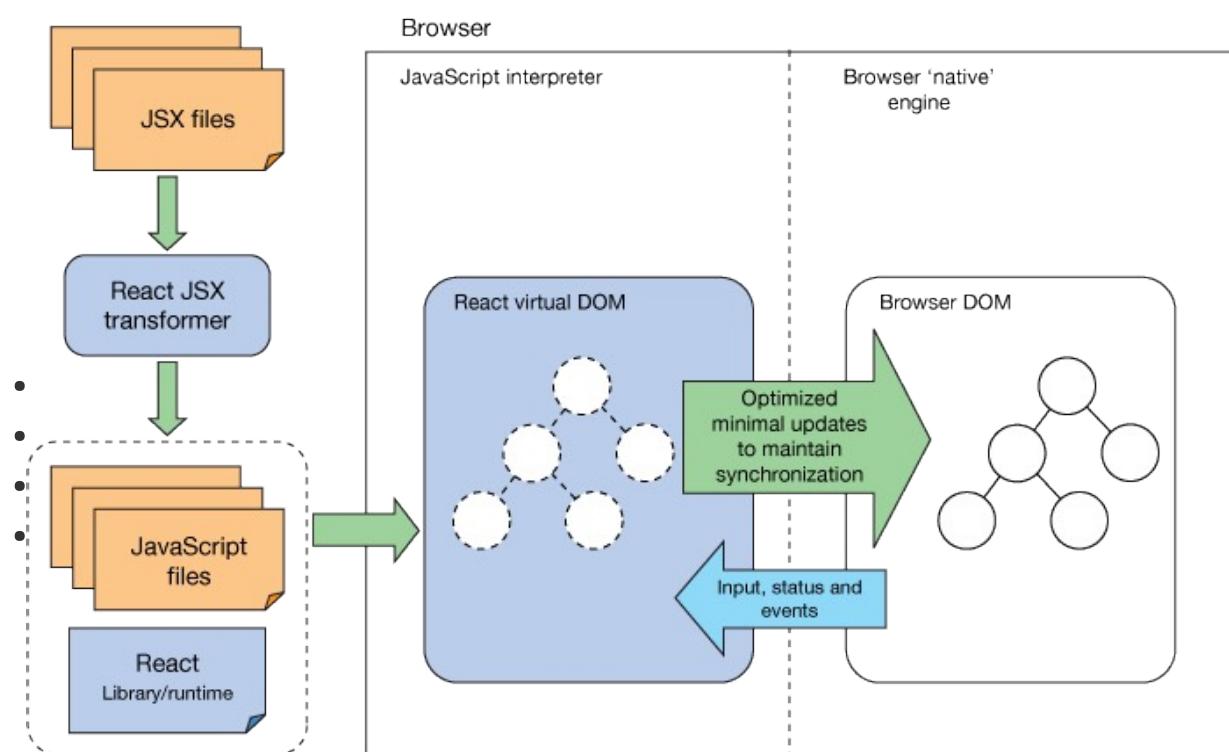
La commande précédente effectue plusieurs opérations :

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 renvoyée (équivalent à `npm view async version` et `npm view yargs version` — respectivement `0.9.0` et `1.3.1`) ;
3. téléchargement et décompression des paquets dans les répertoires `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`.

ReactJS.



Présentation.



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.

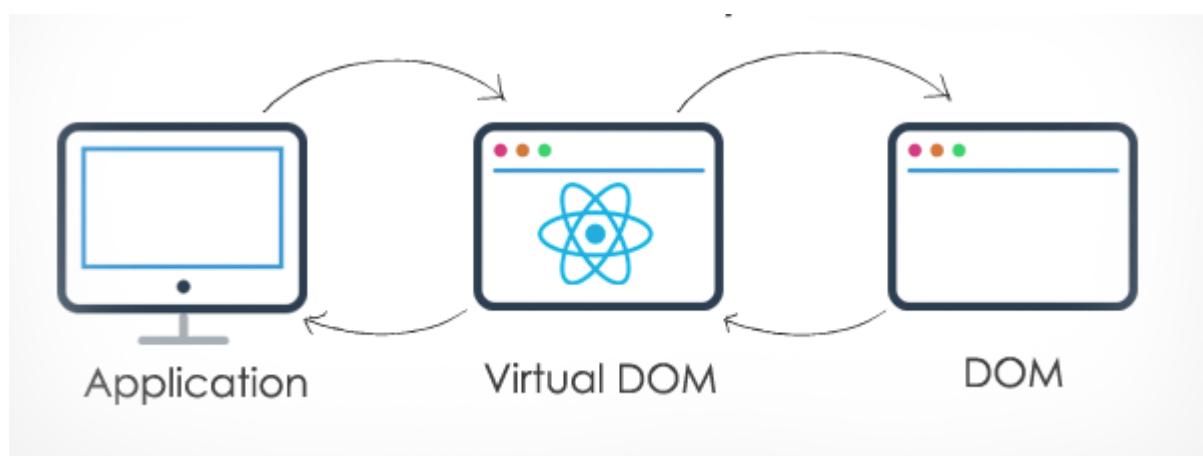
La logique *best of breed*

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 autre bibliothèque ou un framework (AngularJS).

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

ES6

Avant Propos



Développées par les [membres](http://tc39wiki.calculist.org/about/people/) (<http://tc39wiki.calculist.org/about/people/>) du groupe de travail [TC39](http://www.ecma-international.org/memento/TC39.htm) (<http://www.ecma-international.org/memento/TC39.htm>) ECMAScript 6 proposé en 2015 est une évolution significative du langage depuis ES5 (2009).

ES6 a atteint le statut de “proposal freeze” en mai 2011 : aucune nouvelle proposition importante ne peut être ajoutée, bien que les propositions peuvent encore être enlevées.

Les moteurs JavaScript mettent en œuvre des caractéristiques individuelles graduellement.

Autres références

- Site : (anglais) [outils pour ES6](https://github.com/addyosmani/es6-tools) (<https://github.com/addyosmani/es6-tools>)
- Site : (playground) [Tester ES6](http://www.es6fiddle.net/) (<http://www.es6fiddle.net/>)
- Site : (playground) [Learn Harmony](http://learnharmony.org/) (<http://learnharmony.org/>)
- Site : (playground) [ES6 ES5 Comparison](http://es6-features.org/) (<http://es6-features.org/>)
- Livre : (anglais) [JS.next: A Manager's Guide](http://chimera.labs.oreilly.com/books/1234000001623/index.html)
[\(<http://chimera.labs.oreilly.com/books/1234000001623/index.html>\)](http://chimera.labs.oreilly.com/books/1234000001623/index.html)
- Livre : (anglais) [You Don't Know JS: ES6 & Beyond](https://github.com/getify/You-Dont-Know-JS/blob/master/es6%20&%20beyond/README.md#you-dont-know-js-es6--beyond) (<https://github.com/getify/You-Dont-Know-JS/blob/master/es6%20&%20beyond/README.md#you-dont-know-js-es6--beyond>)

1/ Configuration du poste de développement.

Installer les logiciels suivants :

Vous trouverez les sources pour l'installation des logiciels dans le répertoire **ressources** fourni.

- Git
- node.js
- Sublime Text 3
- Chrome Canary

Attention à ajouter **git** au **PATH windows!**

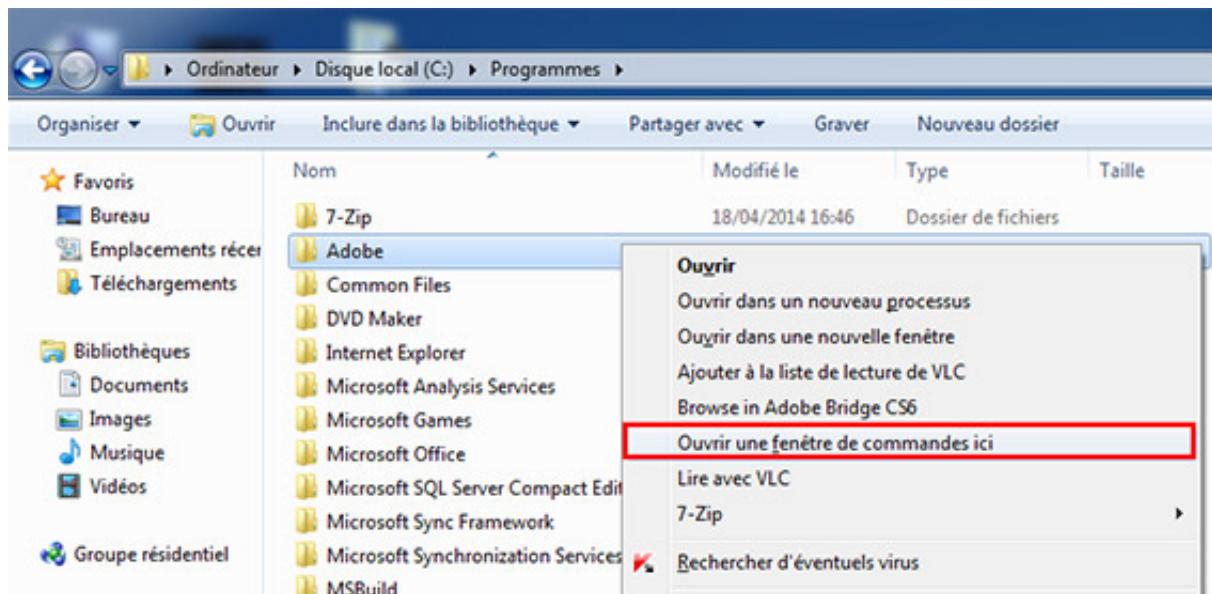
Vérification des installations :

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

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

✓ Installation réussie !

Aisance avec l'invite de commande windows.



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

Manipulation 1

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

Manipulation 2

A l'aide de l'explorateur windows, se placer dans le dossier 'workshops'

Saisir 'cmd' + ENTRÉE dans la barre d'adresse

Répertoire de travail

Créer un répertoire nommé **workshops** sur le bureau.

```
workshops/
    |
    -- concepts-es5/
    |
    -- concepts-es6/
    |
    -- workflow/
        |
        -- src/
        |
        -- dist/
    |
    -- application/
        |
        -- src/
        |
        -- dist/
```

{js}

node.js Est devenu un utilitaire indispensable pour le développeur JavaScript.

Prenez le temps de découvrir son écosystème.

Chrome Canary Est idéal pour tester ES6.

2/ Découverte de l'éditeur Sublime Text 3

[Sublime Text \(https://www.sublimetext.com/\)](https://www.sublimetext.com/) (version 3 en bêta) apporte toutes les fonctionnalités d'un éditeur moderne, sans nécessiter un investissement de formation.

Fonctionalités principales:

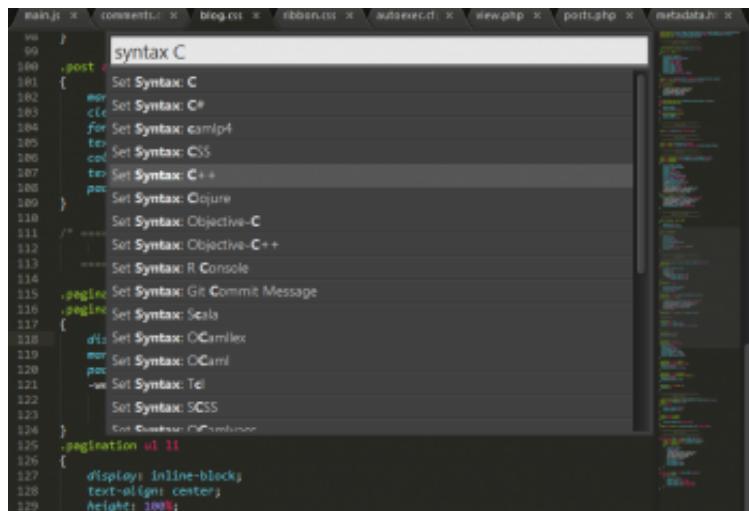
- Minimap
- Palette de commande
- Gestion des fichiers et des projets
- Coder en écran scindé
- “Snippets”
- Console
- Multiplate-forme

Raccourcis utiles :

[https://gist.github.com/spyl94/3963465 \(https://gist.github.com/spyl94/3963465\)](https://gist.github.com/spyl94/3963465)

- Ctrl+Maj+P: palette de commande
- Alt+Maj+(1,2,3,4,5,8,9): changer de mise en page
- Ctrl+D: sélectionner l'occurrence suivante dans le fichier
- Ctrl+Maj+C: montrer la console
- Ctrl+Maj+D: dupliquer la ligne courante
- Ctrl+Maj+/: commenter / décommenter

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
- Ajout de fonctionnalités (Package Control)
- Saisie intuitive

Manipulation 1

cf. pratique

Installer gestionnaire de plugin : Package Control

cf. [url \(<https://packagecontrol.io/installation>\)](https://packagecontrol.io/installation)

Afficher la console : **Menu > View > Show Console**

Coller le script suivant : (depuis le PDF ou en ligne)

```
import urllib.request,os,hashlib;
h = '2915d1851351e5ee549c20394736b442' + '8bc59f460fa1548d1514676163dafc88';
pf = 'Package Control.sublime-package'; ipp = sublime.installed_packages_path();
urllib.request.install_opener( urllib.request.build_opener( urllib.request.ProxyHandler() ) );
by = urllib.request.urlopen( 'http://packagecontrol.io/' + pf.replace(' ', '%20')).read();
dh = hashlib.sha256(by).hexdigest();
print('Error validating download (got %s instead of %s), please try manual install' % (dh, h))
```

Appuyer sur la touche : **ENTREE**

✓ Installation réussie !

Package Control

Ce plugin simplifie l'installation d'autres plugins depuis le serveur central.

- Ajout de fonctionnalités.
- Ajout de fragments de code (code snippet).

Manipulation 1

Installer le plugin **HTML-CSS-JS-Prettify**

- Ouvrir la palette de commande. **CTRL+MAJ+P**
- Saisir ‘ip’ comme ‘Install Package’
- Dans la nouvelle invite saisir ‘HCS’
- Sélectionner **HTML-CSS-JS-Prettify**
- Appuyer sur la touche : **ENTREE**
- ✓ Installation réussie !

Manipulation 2

Répéter autant de fois que nécessaire la manipulation 1

Installer les plugins :

- Bootstrap 3 Snippets
- Goto Bootstrap
- Emmet

Manipulation 3

Découverte et utilisation des packages

Les codes Snippets

L'enrichissement de l'IDE se fait simplement par la création de snippets: **menu > tools > new snippet.**

Les code snippets sont des **descriptions XML de fragments de code** à insérer au sein de votre code. Ce sont eux-mêmes des fichiers textes d'extension **.sublime-snippet**

Créer un nouveau “Code Snippet” et enregistrer le dans un dossier User/snippets/html

```
<snippet>
  <content>
    <!--[CDATA[
Hello, ${1:this} is a ${2:snippet}.
]]-->
  </content>
  <!-- Optional: Set a tabTrigger to define how to trigger the snippet -->
  <!-- <tabTrigger>hello</tabTrigger> -->
  <!-- Optional: Set a scope to limit where the snippet will trigger -->
  <!-- <scope>source.python</scope> -->
</snippet>
```

- **content** : contenu inséré par le snippet
- marqueurs **\$** : marquent la position du curseur de saisie par tabulation
- **tabTrigger** : séquence de lettres déclenchant le plugin
- **scope** : restriction selon le type de document (Ctrl+Maj+Alt+P)

Emmet

Le plugin **emmet** disponible pour de nombreux éditeurs de texte est un **outil indispensable pour le développeur WEB**

C'est un outil de productivité évaluant une expression CSS en structure HTML/XML.

Un premier essai :

Créer un nouveau document HTML et saisir l'expression suivante :

```
ul>li>a
```

Appuyer sur la touche **TAB** pour générer le code suivant :

```
<ul>
  <li><a href=""></a></li>
</ul>
```

Emmet supporte des combinaison de la syntaxe CSS commune comme décrit dans la [documentation \(http://docs.emmet.io/cheat-sheet/\)](http://docs.emmet.io/cheat-sheet/).

Attention ! Les expression emmet ne supportent pas les espaces.

Résumé :

```
<!-- Chaine de parenté -->
ul>li>a

<!-- Profondeur identique -->
header+p+footer

<!-- Classe CSS -->
button.btn.btn-primary

<!-- Identifiant -->
div#menu

<!-- Attribut -->
input[placeholder=Email required]
```

De plus Emmet étend la syntaxe de base afin de fournir des raccourcis utiles au développeur.

Essayer les raccourcis suivants :

```
<!-- Initialisation de document -->
html:5

<!-- Insertion de feuille de style -->
link

<!-- Insertion de script javascript -->
script:src

<!--Insertion d'un type particulier d'élément input -->
input:button

<!-- Insertion de 'faux texte' avec ou sans pondération -->
lorem
lorem10

<!-- Insertion de texte prédefini -->
header>h1>{Hello World}

<!-- Répétition -->
ul>li*10>a

<!-- Répétition avec récupération du numéro d'itération-->
ul>li#item$*10>a>{Lien numéro $$}

<!-- Insertion de groupe-->
form>(label[for=i$]+input#i$)*5
```

{js}

Sublime Text 3 Est éditeur très souple.

Choisissez un éditeur léger en complément de votre IDE traditionnel.

Emmet Est disponible pour toute les IDE majeures.

Créer une collection de snippet réutilisable pour entérinner vos bonnes pratiques.



ES6

De JavaScript ES5 à ES2015.

3/ De JavaScript ES5 à ES2015.

La norme JavaScript est appelée officiellement “**ECMAScript**” (en abrégé «ES»), et jusqu'à tout récemment a été versionné entièrement par le nombre ordinal (à savoir, “5” pour “5ème édition”).

Les premières versions, ES1 et ES2, ne sont pas largement connues ou mises en œuvre.

ES3 a été la première version généralisée de JavaScript, et constitue la norme JavaScript pour les navigateurs comme IE6-8 et plus les navigateurs mobiles Android 2.x.

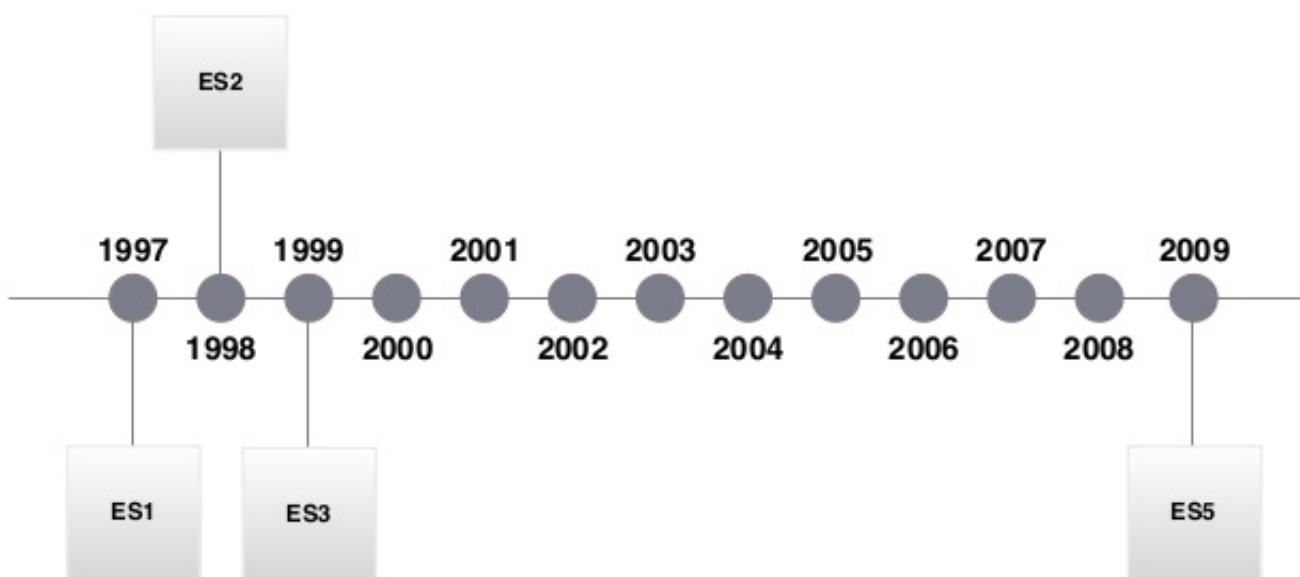
Pour des raisons politiques l’ES4 malheureux n'a jamais vu le jour.

En 2009, ES5 a été officiellement finalisé (plus tard ES5.1 en 2011).

L'évolution JavaScript est très dès qu'une idée commence à progresser, les navigateurs commencent donc intégration, et les adopteurs précoce commencer à expérimenter avec le code.

ECMASCRIPT PAST

Timeline



ES5, JS.next, ES6 / ES2015 évolution et “roadmap”.

TC39 a décidé de déplacer la date de publication officielle de la norme ECMAScript 6 à Juin à 2015.

Le temps supplémentaire est pour plus de rétroaction de mise en œuvre et plus ECMAScript 6 Spécification examen, des corrections de bugs , et le développement de test .

Le jeu de fonctionnalités ECMAScript 6 resteront gelés au cours de cette période de validation étendue .

ECMAScript 6 est essentiellement terminé: son ensemble de fonctionnalités est gelé.

Vous pouvez déjà programmer avec et le *compiler* vers ES6.

Des framework AngularJS et Ember.js se baseront sur ES6 pour leur futures version.

Les évolutions ECMAScript 6 apparaissent dans les navigateurs modernes à un [rythme soutenu](https://kangax.github.io/compat-table/es6/) (<https://kangax.github.io/compat-table/es6/>).

Le calendrier précédent se présente comme suit:

- Novembre 2013: examen final du projet
- Juillet 2014: éditorialement complète
- Décembre 2014: approbation Ecma (formelle de la date de publication)

Contrairement à ES5, ES6 est plus qu'un ajout de nouvelles API. Il intègre de nouvelles formes syntaxiques, dont certaines peuvent demander un peu de temps d'adaptation, en plus de nouvelles formes d'organisation et de nouvelles API.

ES5, rappel des bonnes pratiques. Etablir les règles de programmation.

Il est important d'établir un socle de compétences solides sur JavaScript dans sa norme courante, communément appelé **ES5** (techniquement ES 5.1) avant d'aborder la prochainne version **ES6**

- **Vous:** êtes nouveau à la programmation et JavaScript? Alors il vous faut commencer par les base ES5 : **ES6 est un enrichissement de ES5**

Quelques questions simples pour confirmer le socle de compétences.

- **Portée & Closure:** Saviez-vous que JavaScript utilise portée lexicale est basé sur le compilateur (pas l'interpréteur!)

Pouvez-vous expliquer :

L'ordre de résolution des variables ?

Leur principe de transmission ?

Pourquoi les *closures* sont le résultat direct de la portée lexicale et fonctions en tant que valeurs?

- **this & Prototypes d'objets:** Pouvez-vous donner les quatre règles simples d'initialisation du mot clé `this` est lié?
- **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 "callback enfer"?

ES5 “use strict” et méthodes moins connues.

Tout vos script devraient utiliser la directive ‘**use strict**’;

Le mode strict de ECMAScript 5 permet de choisir une variante restrictive de JavaScript. Les navigateurs ne supportant pas le mode strict exécuteront le code d'une façon légèrement différente de ceux le supportant.

```
// Tenter d'exécuter 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.

Voir la page Passer au mode strict pour plus de détails quant à la migration d'une base de code non-stricte vers une base de code compatible avec le mode strict.

Invoquer le mode strict

Le mode strict s'applique à des scripts entiers ou à des fonctions individuelles.

```
// Script entier en mode strict
"use strict";
var v = "Allo ! Je suis en mode strict !";
```

Attention : il n'est pas possible de concaténer du script en mode strict et du code en mode non-strict. Lors d'une phase de transition, il est donc recommandé de n'activer le mode strict que fonction par fonction.

Le mode strict pour les fonctions

Pour activer le mode strict pour une fonction, on placera l'instruction exacte “use strict”; (ou ‘use strict’;) dans le corps de la fonction avant toute autre déclaration.

```
function strict(){
    // Syntaxe en mode strict au niveau de la fonction
    'use strict';
    function nested() { return "Ho que oui, je le suis !"; }
    return "Allô ! Je suis une fonction en mode strict ! " + nested();
}
function notStrict() { return "Je ne suis pas strict."; }
```

En utilisant “**use strict**”; certaines instructions ou fragments de code lanceront une exception `SyntaxError` avant l’exécution du script :

- La syntaxe pour la base octale : `var n = 023;`
- L’instruction `with`
- L’instruction `delete` pour un nom de variable `delete maVariable;`
- L’utilisation de `eval()` ou `arguments` comme un nom de variable ou un nom d’argument
- L’utilisation d’un des **nouveaux mots-clés réservés** (en prévision d’ECMAScript 6) :
`implements, interface, let, package, private, protected, public, static, et yield`
- La déclaration de fonctions dans des blocs `if(a<b){ function f(){}}}`
- Les erreurs évidentes
- Déclarer deux fois le nom d’une propriété dans un littéral objet `{a: 1, b: 3, a: 7}`. Ceci n’est plus le cas pour ECMAScript 6 : bug 1041128
- Déclarer deux arguments de fonction avec le même nom `function f(a, b, b){}`

Ces erreurs sont bienvenues car elles révèlent des mauvaises pratiques et certaines erreurs claires. Elles apparaissent avant l’exécution du code.

Minimisez l’utilisation des éléments dont la sémantique pourrait changer :

`eval` : *n’utilisez cette fonction uniquement si vous êtes certains que c’est l’unique solution*

`arguments` : utilisez les arguments d’une fonction via leur nom ou faites une copie de l’objet en utilisant : ``var args = Array.prototype.slice.call(arguments)`

* **`this`** : n’utilisez `this` uniquement pour faire référence à un objet que vous avez créé

ES5 méthodes moins connues.

Les [fonctionnalités dépréciées](https://goo.gl/5ATj0X) (<https://goo.gl/5ATj0X>)

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. La méthode renvoie l'objet modifié.

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.defineProperties

La méthode Object.defineProperties() permet de définir ou de modifier les propriétés d'un objet directement sur celui-ci. La valeur renvoyée est l'objet modifié.

```

var obj = {};
Object.defineProperties(obj, {
  "propriété1": {
    value: true,
    writable: true
  },
  "propriété2": {
    value: "Coucou",
    writable: false
  }
  // etc.
});

```

Object.getPrototypeOf

La méthode `Object.getPrototypeOf()` renvoie le prototype d'un objet donné.

```
var proto = {};
var obj = Object.create(proto);
Object.getPrototypeOf(obj) === proto; // true
```

Object.keys

La méthode `Object.keys()` renvoie un tableau des propriétés propres à un objet (qui ne sont pas héritées via la chaîne de prototypes) et qui sont énumérables.

```
var arr = ["a", "b", "c"];
console.log(Object.keys(arr)); // affichera ['0', '1', '2']

// un objet semblable à un tableau
var obj = { 0 : "a", 1 : "b", 2 : "c"};
console.log(Object.keys(obj)); // affichera ['0', '1', '2']

// un objet semblable à un tableau avec un ordre de clé aléatoire
var an_obj = { 100: "a", 2: "b", 7: "c"};
console.log(Object.keys(an_obj)); // affichera [2, 7, 100]

// getName est une propriété non énumérable
var monObjet = Object.create({}, { getName : { value : function () { return this.name } } });
monObjet.name = 1;

console.log(Object.keys(monObjet)); // affichera ['name']
```

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)
    });
}
```

API issues de la communauté JavaScript.

La communauté des développeurs JavaScript apport parfois des convention ou librairie en réponse à une problématique données (DOM, Asynchronicité...) et certaines des ces solutions se voient parfois reprise dans les standards.

Getter/Setter : non standardisé mais très répandu.

```
$('.li').html() // getter : retourne La valeur  
$('.li').html('Hello World') // setter : affecte La valeur
```

Promise : standardisé.

On doit une des premières implémentations à [Kris Kowal \(<https://github.com/kriskowal>\)](https://github.com/kriskowal) via la librairie **q**

En substance une **promise** ou traitement déferré doit retourner un objet contenant une méthode **then()** recevant deux paramètres (**callbackSuccess, callbackError**) afin de représenter l'état du traitement (en attente, résolue, rejetée)

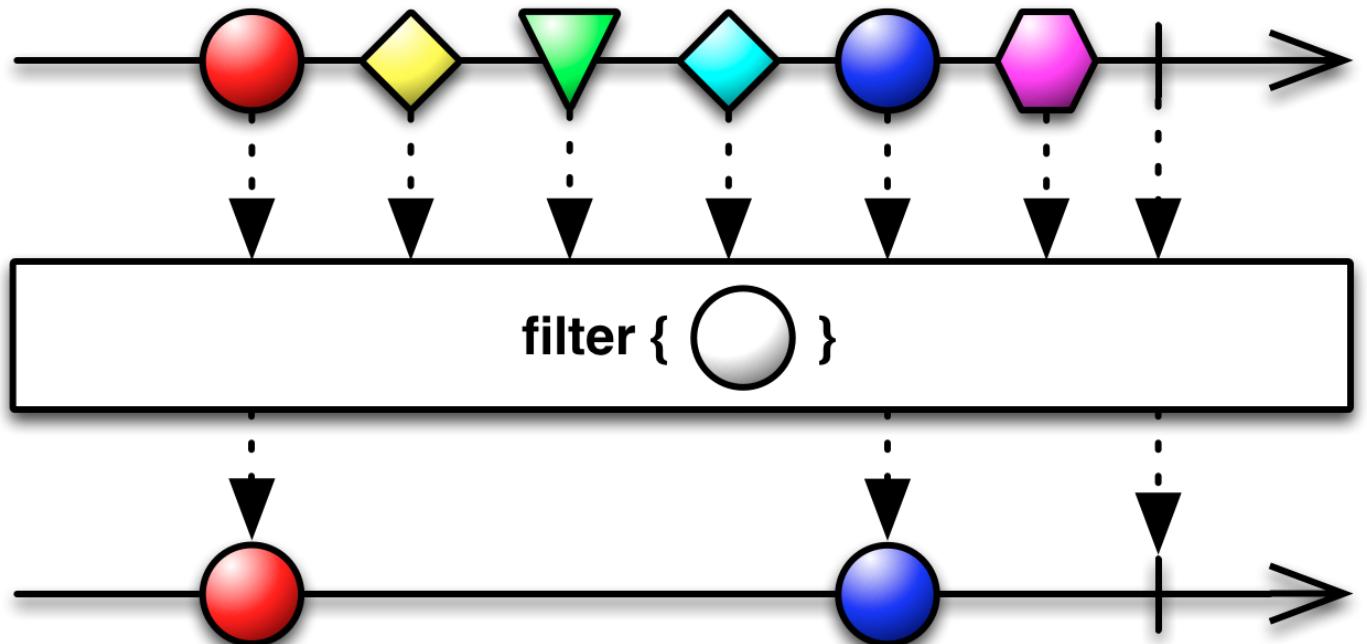
```
var defer = new Promise((resolve, reject) => { }); //Standard ES6  
  
//Avec Q  
Q.fcall(promisedFunction)  
.then(function (value) {  
})  
.catch(function (error) {  
})  
.done();
```

Standardisation de la manipulation du DOM basée sur les sélecteur CSS

Héritage de **jQuery**

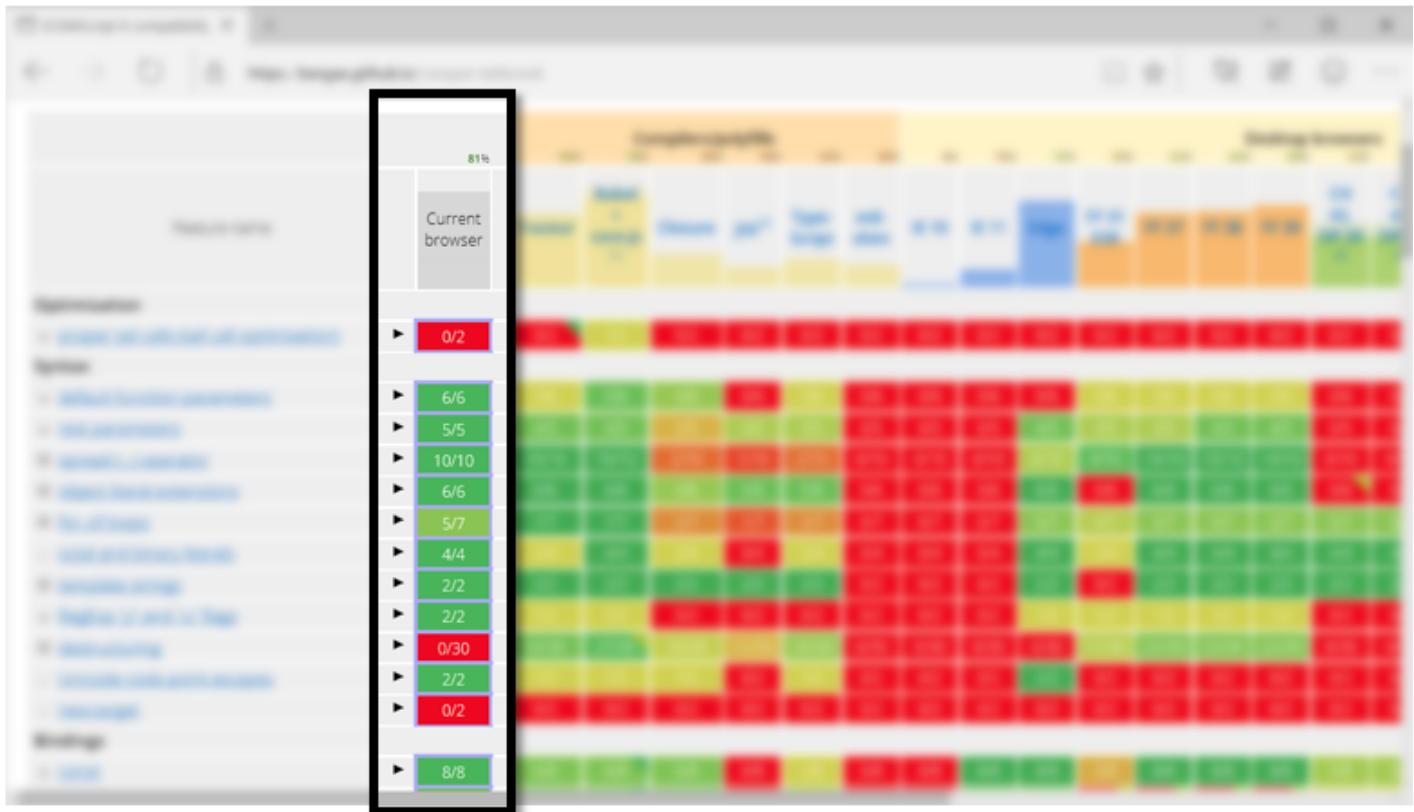
```
document.querySelector('body>.btn');  
document.querySelectorAll('body>.btn');
```

Des librairies telles que **reactivex.io** posent de nouveau paradigmes tels que la représentation d'évennement sous la forme de flux itérables.



Support courant pour ES6 : compilateurs, polifills, navigateurs serveurs.

La meilleure façon de suivre l'évolution du support ES6 sont les [tableaux de compatibilité ECMAScript](https://kangax.github.io/compat-table/es6/) (<https://kangax.github.io/compat-table/es6/>) maintenu par **Juriy Zaytsev** (@kangax)



Environnement et outils pour le développeur.

Transpilers

Avec l'évolution rapide des caractéristiques, un problème se pose pour les développeurs JavaScript qui désire utiliser les nouvelles fonctionnalités tout en maintenant ses applications pour les anciens navigateurs sans cette soutien.

Avant ES5 il était d'usage d'attendre que la plupart sinon tous les environnements javascript supporte ES5. En conséquence d'application 'utilise pas encore des options comme le strict , qui a pourtant plus de cinq ans.

Il est considéré néfaste pour l'avenir de l'écosystème JS d'attendre avant d'utiliser les fonctionnalités qui représentent de bonne pratiques

Comment anticiper le support natif ?

La réponse est l'outillage, en particulier une technique appelée **transpiling** (transformation + compilation).

L'idée est d'utiliser un outil spécial pour transformer votre code ES6 à un équivalent (ou presque!) ES5.

Les *transpilers* assure ce transformation comme part du **workflow** de la même manière que les *linter* ou les opérations de minification.

```
var foo = [1,2,3];

var obj = {
  foo // means `foo: foo`
};

obj.foo; // [1,2,3]
```

Exemple de *transpilation*:

```
var foo = [1,2,3];

var obj = {
  foo: foo
};

obj.foo; // [1,2,3]
```

Shims/Polyfills

Polyfill implémentation d'une fonctionnalité attendue mais non disponible.

```
if (!Object.is) {
  Object.is = function(v1, v2) {
    // test for `~-0~
    if (v1 === 0 && v2 === 0) {
      return 1 / v1 === 1 / v2;
    }
    // test for `NaN`
    if (v1 !== v1) {
      return v2 !== v2;
    }
    // everything else
    return v1 === v2;
  };
}
```

“[ES6 Shim](https://github.com/paulmillr/es6-shim/)” (<https://github.com/paulmillr/es6-shim/>) est une collection de polyfills que vous pouvez utiliser.

{js}

ES6 Est utilisable en production au moyen de transpiler.

Les nouvelles librairies et frameworks se baseront sur ES6.

ES6-Shim Est disponible.

Utilisez le mode strict dans votre code.

JavaScript est un language dynamique suivant des règles simples (code hoisting, scope...).

Object et Array possède des méthodes ES5 qui allégeront votre code.



ES6

Evolutions syntaxiques.

4/ Evolutions syntaxiques fondamentales.

L'optimisation des `tail calls` constituera à terme une optimisation significative de l'exécution récursive des fonctions pour ne pas bloquer le `stack` d'exécution et plus généralement l'invocation d'une fonction comme valeur de retour.

```
function factorial(x) {
    if (x <= 0) {
        return 1;
    } else {
        return x * factorial(x-1); // (A)
    }
}
```

Support de l'Unicode, de la nouvelle forme Unicode littérale dans les chaînes et le nouveau mode `u` pour les RegExp, ainsi que de nouvelles API pour traiter des chaînes. Ces ajouts favorise la création d'applications globalisées en JavaScript.

```
// ES5.1
"𠮷".length == 2

// Nouveau mode pour les RegExp
"𠮷".match(/./u)[0].length == 2

// Nouvelle forme littérale
"\u{20BB7}"=="𠮷"=="\uD842\uDFB7"

// Gestion des chaînes
"𠮷".codePointAt(0) == 0x20BB7

// for-of itération
for(var c of "𠮷") {
    console.log(c);
}
```

```
0b111110111 === 503 // true
0o767 === 503 // true
```

Reflect API l'inverse de l'API Proxy permettant de faire des appels correspondant aux mêmes méta-opérations.

Aperçu général des apports.

Promise <small>easy</small> basics <small>new</small> creation chaining `then()` the API	Array <small>easy</small> <code>Array.from()</code> <code>Array.of()</code> <code>[].fill()</code> <code>[].find()</code> <code>[].findIndex()</code> <code>[].entries()</code> <code>[].keys()</code> <code>[].values()</code>	Class <small>easy</small> creation accessors <small>easy</small> static <small>easy</small> extends more extends super in method super in constructor	Destructuring <small>easy</small> array <small>easy</small> string <small>easy</small> object <small>easy</small> defaults parameters assign	Generator creation iterator yield expressions send value to a generator send function to a generator `return` inside a generator function	Map <small>easy</small> Basics <code>map.get()</code> <code>map.set()</code> initialize <small>easy</small> <code>map.has()</code>
Reflect <small>easy</small> Basics <code>Reflect.apply()</code> <code>Reflect.getPrototypeOf()</code> <code>Reflect.construct()</code> <code>Reflect.defineProperty()</code>	Set <small>easy</small> basics <code>Set.add()</code> <small>easy</small> <code>Set.delete()</code> <small>easy</small> the API <small>easy</small> <code>Set.clear()</code>	Iterator <small>easy</small> array <small>easy</small> string <small>easy</small> protocol <small>easy</small> usage	Object literal <small>easy</small> basics computed properties <small>easy</small> getter <small>easy</small> setter	String <small>easy</small> <code>String.includes()</code> <small>easy</small> <code>String.repeat(count)</code> <small>easy</small> <code>String.startsWith()</code> <small>easy</small> <code>String.endsWith()</code>	
Template strings <small>easy</small> basics <small>easy</small> multiline tagged template strings `raw` property	Symbol <small>easy</small> basics <code>Symbol.for()</code> <code>Symbol.keyFor()</code>	Arrow functions <small>easy</small> basics <small>easy</small> function binding	Block scope <small>easy</small> `let` declaration <small>easy</small> `const` declaration	Rest operator as parameter with destructuring	Spread operator with arrays with strings
Default parameters <small>easy</small> Basics	Modules <small>easy</small> `import` statement	Number <small>easy</small> <code>Number.isInteger()</code>	Object <small>easy</small> <code>Object.is()</code>	Unicode in strings	

Constantes et variables de bloc. Assignation destructurée.

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, cela entraîne une exception `TypeError`. Contrairement au corps de fonction.

```
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`. La variable est placée dans une « zone morte temporaire » entre le début du bloc et le moment où la déclaration est traitée.

Zone morte temporaire (temporal dead zone / TDZ) concerne les erreurs liées à `let`

```
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);
};
```

Assignation destructurée.

L'affectation par décomposition (destructuring en anglais) est une expression JavaScript permettant d'extraire des données d'un tableau ou d'un objet d'après une comparaison de forme syntaxique.

```
[a, b] = [1, 2]
[a, b, ...c] = [1, 2, 3, 4, 5]
{a, b} = {a:1, b:2}
```

L'intérêt de l'assignation par décomposition est de pouvoir lire une structure entière en une seule instruction.

```
var myVar= ["un", "deux", "trois"];

// sans utiliser la décomposition
var un    = myVar[0];
var deux  = myVar[1];
var trois = myVar[2];

// en utilisant la décomposition
var [un, deux, trois] = myVar;
```

Échange de variables :

```
var a = 1;
var b = 3;

[a, b] = [b, a];
```

Renvoyer plusieurs valeurs:

Il était déjà possible de renvoyer un tableau/objet mais cela ajoute un nouveau degré de flexibilité.

```
function f() {
    return [1, 2];
}
var [a, b] = f();
var x = f();
console.log("A vaut " + a + " B vaut " + b, x);

var url = "http://www.orsys.fr/formation-Ecmascript-6.asp/";

var parsedURL = /^(w+)\:\/\//([^\/]+)\/(.*)$/.exec(url);
var [, protocol, fullhost, fullpath] = parsedURL;

console.log(protocol); // enregistre "http"
```

Ignorer certaines valeurs:

```
function f() {
    return [1, 2, 3];
}

var [a, , b] = f();
console.log("A vaut " + a + " B vaut " + b);
```

Décomposition d'objet:

```
var o = {a: 111, b: true};  
var {a, b} = o;  
  
console.log(a); // 111  
console.log(b); // true  
  
// Réasignation  
var {a: newA, b: newB} = o;  
  
console.log(newA); // 111  
console.log(newB); // true
```

Propriétés calculées:

Il est possible d'utiliser des noms de propriétés calculés, comme avec les littéraux objets, avec la décomposition.

```
let key = "a";  
let { [key]: myVar } = { a: "Hello World" };  
  
console.log(myVar); // "truc"
```

Fonction, paramètres par défaut, opérateurs “rest / spread”.

Paramètres par défaut.

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

```
//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
```

Les paramètres déjà rencontrés dans la définition peuvent être utilisés comme paramètres par défaut dans la suite de la définition :

```
function singulierAutoPluriel(singulier, pluriel = singulier+s", message = pluriel +
  return [singulier, pluriel, rallyingCry ];
}

function go() {
  return ":P"
}

function avecDéfaut(a, b = 5, c = b, d = go(), e = this, f = arguments, g = this.value
  return [a,b,c,d,e,f,g];
}
```

Auparavant, pour définir une valeur par défaut pour un paramètre, il fallait tester s'il valait undefined et lui affecter une valeur choisie le cas échéant.

L'argument par défaut est évalué à l'instant de l'appel. Un nouvel objet est créé à chaque appel de la fonction.

Paramètres “rest”.

La syntaxe des paramètres du reste 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(...[, , 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);
```

Décomposition de paramètres.

A noter La décomposition de paramètres est supposée apporter de la lisibilité !

Les règles de déstructuration vues ci-avant s'appliquent au paramètres de fonction pour l'assignation de valeur par défaut.

```
function foo( { x, y } ) {
  console.log( x, y );
}

foo( { y: 1, x: 2 } ); // 2 1
foo( { y: 42 } );    // undefined 42
foo( {} );          // undefined undefined
```

Avec valeurs par défaut :

```
function foo( { x = 10, y = 20 } ) {
  console.log( x, y );
}

foo( { y: 1, x: 2 } );    // 2 1
foo( { y: 42 } );        // 10 42
foo( {} );               // 10 20
foo( );                 //Error
```

En mappant des valeurs :

```
function foo( { name:n = 'Bob', age:a = 20 } = {name:'',age:0} ) {
  console.log( n, a );
}

foo( { name: 'Bill', age: 25 } );  // 10 25
foo( );                          // 10 20
```



Wagon Trés Fumant regardons en détail !

```
{ name:n = 'Bob', age:a = 20 } Définition des valeurs de clés d'objets (déstructuration).  
name:n Map la clé name sur l'argument n  
n = 'Bob' Fixe la valeur par défaut de n à 'Bob'  
age:a Map la clé age sur l'argument a  
a = 20 Fixe la valeur par défaut de a à 20
```

{name: '', age:0} Définition des valeurs par défaut de paramètres

Assigné un objet par défaut si celui n'est pas transmis.

Noter La résolution étant faite dans le cycle d'exécution le moteur a besoin d'un objet à déstructurer pour ne pas produire d'erreur.

Mapper sur un nouvel objet:

```
function foo( { name:n , age:a } = {name:'',age:0},obj = {innerName:n,innerAge:a} ) {
  console.log( obj );
}

foo( { name: 'Bill' , age: 25 } );
foo( );
```

Wapa Tout Feuillu ;) Lisibilité ?

L'évaluation des paramètres pendant l'exécution peut produire du code de moins bonne qualité.

```
function foo( { name:n , age:a } = {name:'',age:0},
  obj = {innerName:((s)=>s.toUpperCase())(n),innerAge:a} ) {
  console.log( obj );
}

foo( { name: 'Bill' , age: 25 } );
```

Variations possible de la syntaxe:

```
function f1([ x=2, y=3, z ]) { .. }
function f2([ x, y, ...z ], w) { .. }
function f3([ x, y, ...z ], ...w) { .. }

function f4({ x: X, y }) { .. }
function f5({ x: X = 10, y = 20 }) { .. }
function f6({ x = 10 } = {}, { y } = { y: 10 }) { .. }
```

Chaînes de caractères : multiligne, template, formatage.

Les **template string** (template literals ou template string) sont des gabarits de chaînes de caractères qui intègrent des expressions. Il est possible d'utiliser des chaînes de caractères sur plusieurs lignes ainsi que les fonctionnalités d'interpolation de chaînes.

Syntaxe : le délimiteur est le symbole ``

Les **template string** délimités par des accents graves seuls (backticks) (`) et non avec des doubles ou simples quotes.

```
`chaîne de texte`  
  
`chaîne ligne 1  
chaîne ligne 2`  
  
`texte ${expression} texte chaîne`  
  
function tag(){console.log(arguments)}  
tag `texte ${expression} texte chaîne`
```

Ils peuvent contenir des éléments de substitution (placeholders) indiqués par le signe dollar (\$) et des accolades : **`\${expression}`**.

Les expressions contenues dans les éléments de substitution et le texte sont ensuite passés à une fonction.

Si une expression précède le gabarit (par exemple : tag ci-avant), le gabarit appelé “étiqueté”. Dans ce cas, l’expression qui étiquette (généralement, ce sera un fonction) le gabarit est appelée pour traiter le gabarit.

Interpolation d'expression

- Le code desubstitution peut être toute expression JavaScript, donc des appels de fonction, l'arithmétique sont autorisés.
- Si l'une des valeurs est pas une chaîne, il sera converti en une chaîne en utilisant les règles habituelles. Par exemple, si l'action est un objet, sa méthode `.toString()` sera appelée.
- Si vous devez écrire un backtick dans une chaîne de modèle, vous devez échapper avec une barre oblique inverse: `\`` est le même que `" "`.
- Pour les deux caractères `$ {` vous pouvez échapper le caractère avec une barre oblique inverse: `\$` ou `\$`.

```
var a = 5;
var b = 10;
console.log(`Quinze est ${a + b} et n'est pas ${2 * a + b}.`);
// "Quinze est 15 et n'est pas 20."
```

Il est possible d'utiliser les gabarits de manière plus avancée grâce à l'étiquetage de gabarits.

Cette fonction d'étiquette (tag) permettra de modifier la chaîne résultante d'un gabarit.

Le premier argument de la fonction est un tableau de littéraux de chaînes de caractères. **Les arguments suivants** représentent les valeurs qui sont à traiter.

Finalement, la fonction renvoie la chaîne résultante du gabarit. Le nom utilisé pour la fonction peut être n'importe quel nom valide.

```
var a = 5;
var b = 10;

function tag(strings, ...values) {
  console.log(strings[0]); // "Hello "
  console.log(strings[1]); // " World "
  console.log(values[0]); // 15
  console.log(values[1]); // 50

  return "Bazinga !";
}

tag`Hello ${ a + b } World ${ a * b }`;
// "Bazinga !"
```

La propriété spéciale `raw, est une propriété du premier argument de la fonction d'étiquette vue ci-avant. Elle permet d'accéder aux chaînes brutes, telles qu'elles ont été entrées.

De plus, la méthode `String.raw()` permet de créer des chaînes de caractères brutes, de la même façon qu'avec la concaténation par défaut des gabarits :

```
function tag(strings, ...values) {
  console.log(strings.raw[0]);
}

tag `chaîne ligne 1 \n chaîne ligne 2`;
// affichera dans la console :
// "chaîne ligne 1 \\n chaîne Ligne 2"

String.raw`Salut\n${2+3}!`;
```

Recommandation : Dans la mesure du possible encasez vos gabarit ! Pour éviter les manipulations.

L'évaluation des chaînes de gabarits donne accès au fonction.

```
`${alert("this is",this)}`;
```

Les template strings sont une solution prometteuse pour l'internationalisation

```
i18n`Hello ${name}, you I come from ${country}.`
```

“Arrow Function” : portée lexicale. Usages.

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

Les fonctions ainsi créées sont syntaxiquement similaires 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 fonctions fléchées sont nécessairement anonymes.

Les **Arrows** sont un raccourci pour la création de fonction en utilisant le `=>`. Les fonctions ainsi créées sont syntaxiquement similaires à la fonction en C#, Java 8 et CoffeeScript.

Elles définissent le corps (bloc de code) et la valeur de retour.

Fonction itératrice : “iterator”.

Un des éléments ajoutés par ECMAScript 2015 (ES6) n'est ni une nouvelle syntaxe ni un nouvel objet natif mais un protocole.

Ce protocole peut être implémenté par n'importe quel objet qui respecte certaines conventions.

Deux composantes : les itérables, les itérateurs.

Le protocole « itérable » permet aux objets JavaScript de définir ou de personnaliser leur comportement lors d'une itération, par exemple la façon dont les valeurs seront parcourues avec une boucle `for..of`.

Certains types natifs tels que `Array` ou `Map` possèdent un comportement itératif par défaut, d'autres types, comme `Object` n'ont pas ce type de comportement.

itérable

Pour itérable, un objet doit implémenter la méthode `@@iterator`, l'objet doit avoir une propriété avec une clé `Symbol.iterator` :

`[Symbol.iterator]` Une fonction sans argument qui renvoie un objet conforme au protocole itérateur.

itérateur

Un objet est considéré comme un itérateur lorsqu'il implémente une méthode `next()` :

Une fonction sans argument qui renvoie un objet qui possède deux propriétés :

- **done (booléen)** true lorsque l'itérateur a fini la suite. **false** lorsque l'itérateur a pu produire la prochaine valeur de la suite.
Si on ne définit pas la propriété `done`, on aura ce comportement par défaut.
- **value (any)**, renvoyée par l'itérateur.
Cette propriété peut être absente lorsque `done` vaut `true`.

Exemple : Une String est un exemple d'objet natif itérable :

```
var msg = "Hello";
typeof msg[Symbol.iterator];// "function"

var iterator = msg[Symbol.iterator]();
console.log(iterator);    // "[object String Iterator]"

iterator.next(); // { value: "H", done: false }
iterator.next(); // { value: "e", done: false }
iterator.next(); // { value: "l", done: false }
iterator.next(); // { value: "l", done: false }
iterator.next(); // { value: "o", done: false }
iterator.next(); // { value: undefined, done: true }
```

Les itérables natifs

String, Array, TypedArray, Map et Set sont des itérables natifs car leurs prototypes possèdent une méthode `@@iterator`.

Il est possible de redéfinir le comportement par défaut en définissant soi-même le symbole `@@iterator`

```
var msg = new String("Hello"); // objet String explicite afin d'éviter la conversion

msg[Symbol.iterator] = function() {
  return { // L'objet itérateur qui renvoie un seul élément, la chaîne "bop"
    next: function() {
      if (this._first) {
        this._first = false;
        return { value: "World", done: false };
      } else {
        return { done: true };
      }
    },
    _first: true
  };
}
var iterator = msg[Symbol.iterator]();
iterator.next();
```

Exemple : fonction de création d'itérateur sur un tableau.

```
function setIterator(collection){
    var nextIndex = 0;

    return {
        next: function(){
            return nextIndex < collection.length ?
                {value: collection[nextIndex++], done: false} :
                {done: true};
        }
    }
}

var it = setIterator(['Hello', 'World']);

console.log(it.next().value); // 'Hello'
console.log(it.next().value); // 'World'
console.log(it.next().done); // true
```

Si une méthode @@iterator d'un objet itérable ne renvoie pas d'objet itérateur, on dira que cet objet est un itérable mal-formé.

Eléments de syntaxique utilisant des itérables

```
// Itération avec for of
for(let value of ["a", "b", "c"]){
    console.log(value);
}
// "a"
// "b"
// "c"

// rest, spread
[..."abc"]; // ["a", "b", "c"]

//Fonction génératrice
function* gen(){
    yield* ["a", "b", "c"];
}

gen().next(); // { value:"a", done:false }

// Décomposition
[a, b, c] = new Set(["a", "b", "c"]);
a; // "a"
```

Fonction génératrice : “yield” et “generator object”.

La déclaration **function*** (**le mot-clé function suivi par un astérisque**) permet de définir un générateur (objet Generator, aussi appelé une fonction génératrice).

Un générateur est un itérateur et un itérable.

Un générateur n'est pas invocable via l'opérateur **new**

Syntaxe:

```
function* nom([param1[, param2[, ... paramN]]]) {  
    //instructions  
}
```

nom: Le nom de la fonction.

paramN: Le nom d'un argument qu'on passe à la fonction. Une fonction peut avoir jusqu'à 255 arguments.

Les générateurs sont des fonctions qu'il est possible de quitter puis de reprendre. Le contexte d'un générateur (les variables) est sauvegardé entre les reprises successives.

Lorsqu'on appelle une fonction génératrice, son corps n'est pas exécuté immédiatement, c'est un itérateur qui est renvoyé pour la fonction. Lorsque la méthode `next()` de l'itérateur est appelée, le corps de la fonction génératrice est utilisé **jusqu'à ce que la première expression yield soit trouvée**.

```
function* creerID(){  
    var index = 0;  
    while(index < 3){  
        yield index++;  
    }  
  
    var gen = creerID();  
  
    console.log(gen.next().value); // 0  
    console.log(gen.next().value); // 1  
    console.log(gen.next().value); // 2  
    console.log(gen.next().value); // undefined  
    console.log(gen.next().done); // true
```

`yield*`, permet de déléguer la génération des valeurs à une autre fonction génératrice.

```
function* autreGenerateur(i) {
  yield i + 1;
  yield i + 2;
  yield i + 3;
}
function* generateur(i){
  yield i;
  yield* autreGenerateur(i);
  yield i + 10;
}

var gen = generateur(10);

console.log(gen.next().value); // 10
console.log(gen.next().value); // 11
console.log(gen.next().value); // 12
console.log(gen.next().value); // 13
console.log(gen.next().value); // 20
```

Communication.

Il est possible de passer un paramètre à la méthode `next()`

```
function* say(message){
  let msg = yield console.log(message);
  yield console.log(msg + '!');
}

var speaker = say("Hello");

console.log(speaker.next());
console.log(speaker.next("World"));
console.log(speaker.next().done);
```

Gestion de l'asynchronicité.

Les générateur simplifie l'expression de l'asynchronicité.

- 1) La variable **speaker** est créée et affectée à l'objet générateur.
- 2) **next()** est appelée sur **speaker** pour lancer le générateur.
- 3) Parce que **next()** est seulement appelé une fois que le setTimeout a terminé, nous pouvons nous garantir que la variable var un sera correctement affectée à la ligne suivante

```
const async = (msg) => {
  setTimeout(()=>{
    console.info(msg + '!');
    speaker.next(msg+'!');
  }
  ,1000);
  return msg+'!';
}

function* say(message,constainer){
  let a = yield async(message);
  let b = yield async(a);
  let c = yield async(b);
  console.warn(c);
  constainer['result'] = c;
}

let container = {};
var speaker = say("Hello World",container);
speaker.next();
console.log(container);
```

Exemple avec Ajax

```
// Implémentation très simples
const get = (url) => {
  return function (callback) {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.onload = () => callback(null, xhr.responseText);
    xhr.send();
  };
};

/* getTweets (Generator) */

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

})();

// Initialisation et différentes consommation

console.log([...getTweets])
// ou console.Log(Array.from(getTweets))

for (let result of getTweets) {
  result((err, res) => console.warn(res));
}

/*
Let result = getTweets.next();

result.value((err, res) => {
  if (err) console.Log('Error', err);
  console.warn(res);
});

*/
```

Objet littéral : évolution.

Les objets littéraux sont étendus pour :

- supporter la définition du prototype à la construction.
- simplifier les affectations de propriété/méthodes.
- permettre avec `super` des appels au modèle prototypal.
- calculer des noms de propriété avec des expressions.

```
const handler = () => true;

var obj = {
    //Définition du prototype : __proto__
    __proto__: {toString:()=>'Hello World !'},
    // Raccourci pour 'handler: handler'
    handler,
    // Simplification syntaxique
    desc() {
        // Appel avec super
        return "Got " + super.toString();
    },
    // propriété calculée
    [ 'code' + (() => 'FR')(): 'France',
    [ 'code' + 'UK' ]: 'England',
};

console.log(obj);
obj.handler();
obj.desc();
```

Définir un accesseur avec l'opérateur get

La syntaxe `get` permet de lier une propriété d'un objet à une fonction qui sera appelée lorsqu'on accédera à la propriété.

A noter:

peut être identifié par un nombre ou une chaîne de caractères

ne doit pas posséder de paramètre

* l'identifiant ne doit pas être utilisé avec autre propriété

Un accesseur peut être supprimé grâce à l'opérateur `delete`.

```
var o = {
  get dernier() {
    if (this.journal.length > 0) {
      return this.journal[this.journal.length - 1];
    }
    else {
      return null;
    }
  },
  journal: []
}

o.dernier()

delete o.dernier;
```

Utiliser un nom de propriété calculé

```
var expr = "hello";

var obj = {
  get [expr]() { return "Hello World"; }
};

console.log(obj.hello); // "Hello World"
```

Une technique supplémentaire pour optimiser ou retarder le calcul d'une valeur d'une propriété est de la mettre en cache pour les accès ultérieurs en utilisant des accesseurs « mémoïsés ».

- Si le calcul de la valeur est coûteux.
- Si la valeur est utilisée plus tard ou, dans certains cas, n'est pas utilisée du tout.
- Si elle est utilisée plusieurs fois, il n'est pas nécessaire de la recalculer car sa valeur ne changera pas.

```
...
// Cette exemple retarde la recherche dans le DOM à son utilisation réelle
get notifier() {
    delete this.notifier;
    return this.notifier = document.getElementById("bookmarked-notification-anchor");
},
...
```

Définir un mutateur avec l'opérateur set

La syntaxe `set` permet de lier une propriété d'un objet à une fonction qui sera appelée à chaque tentative de modification de cette propriété.

Il n'est pas possible d'avoir à la fois un mutateur et une valeur donnée pour une même propriété.

A noter:

peut être identifié par un nombre ou une chaîne de caractères

doit avoir exactement un paramètre

* l'identifiant ne doit pas être utilisé avec autre propriété

Pour retirer un mutateur, on peut utiliser l'opérateur delete :

```
var o = {
  set courant (str) {
    this.log= str + ' World';
  },
};

o.courant = 'Hello';
console.log(o);

delete o.courant;
console.log(o);
```

{js}

ES6 apporte une syntaxe fondamentalement nouvelle.

Préservez la lisibilité du code

Les nouveaux types **Set et Map** ne sont pas convertible au moyen de **JSON.parse**.

Les symbols est un méta-programmation.

L'itération est un méta-comportement sur les objet.

Les “const” assure juste la non réassigant de l'identifiant.

Définissez un chemin de migration.



ES6

POO, nouveautés.

5/ POO, nouveautés pour la conception objet.

Modèles de classe et héritage. Méthodes statiques.

Les classes ont été introduites dans JavaScript avec ECMAScript 6 et sont un **sucré 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 occurrences 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 membres d'une classe.

get/set Permet l'utilisation de méthodes sous la forme de propriété.

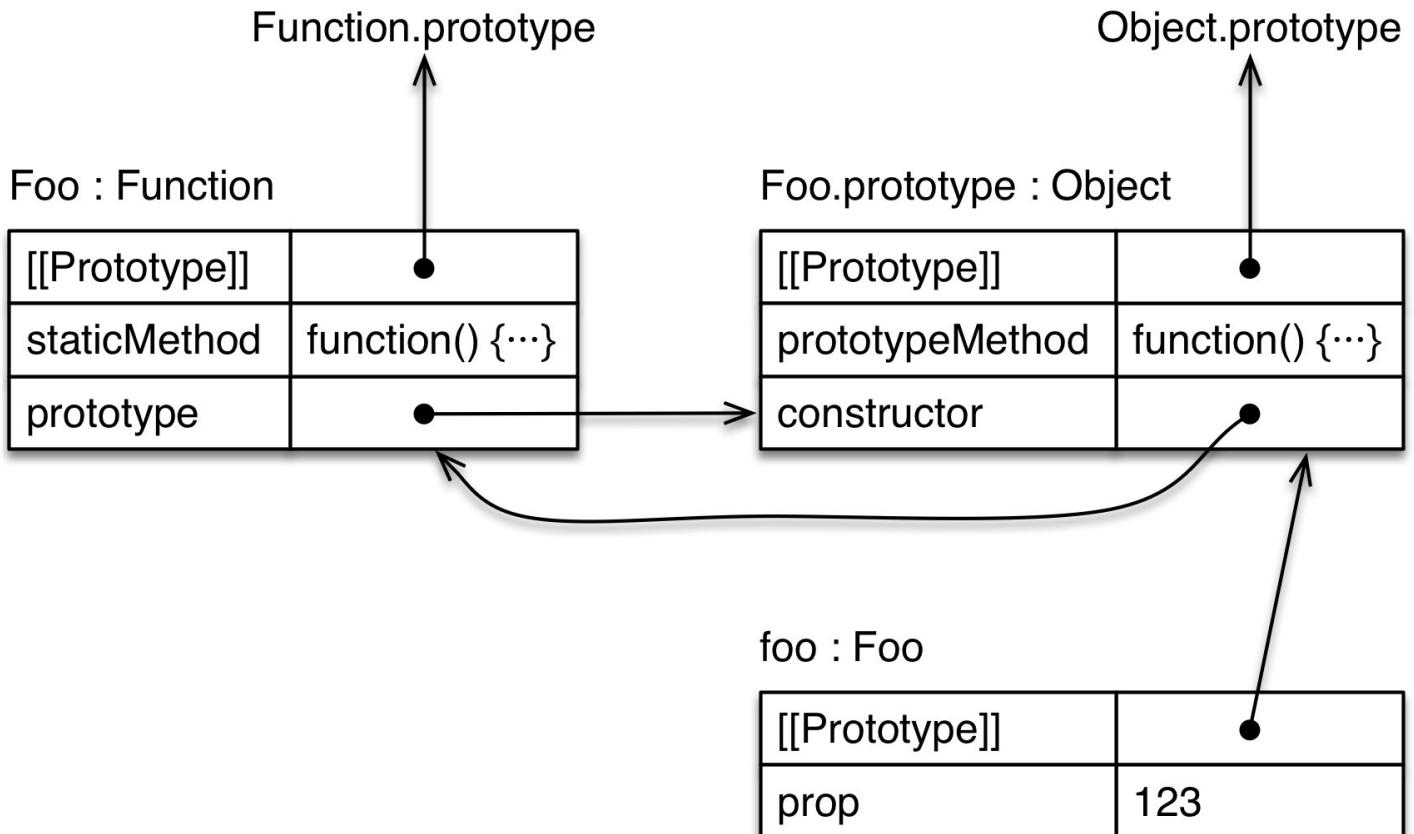
Méthodes statiques

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

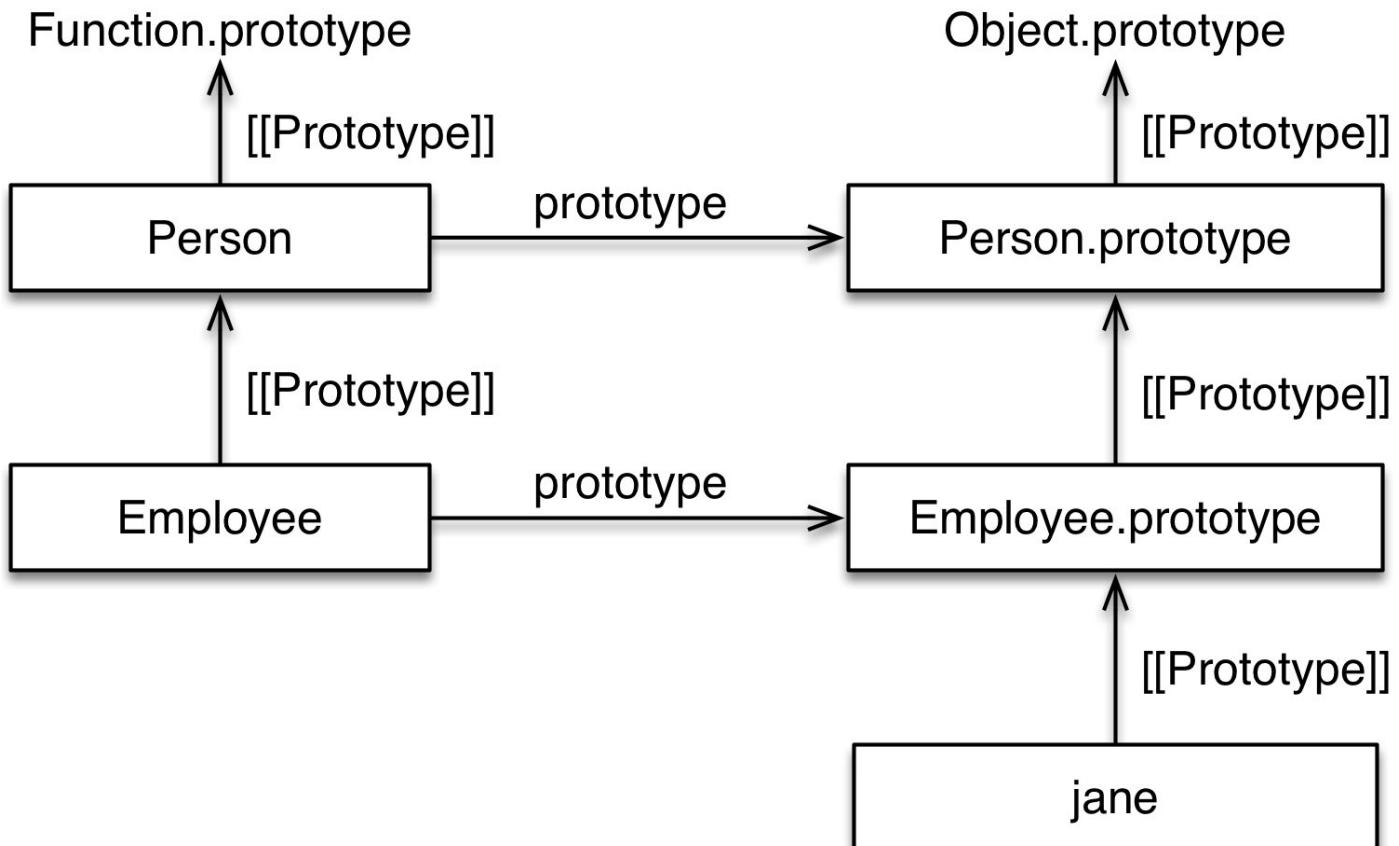
Les méthodes statiques sont appelées par rapport à la classe entière et non par rapport à une instance donnée
Ces méthodes sont généralement utilisées sous formes d'utilitaires.

```
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



Modèle prototypal



```
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)
```

Prorité privées

Quatre approches permettent la création de propriété privées

Restriction au scope du `constructor`

```
class Person {  
    constructor(name) {  
        var name = name;  
        this.getName = () => name;  
    }  
}
```

Utiliser une convention de nommage (préfixe `_underscore`)

```
// Pas privé : convention  
class Person {  
    constructor(name) {  
        this._name = name;  
        this._action = action;  
    }  
    get name(){  
        return this._name;  
    }  
}
```

Utiliser des WeakMaps

```
let _name = new WeakMap();  
class Person {  
    constructor(name) {  
        _name.set(this, name);  
    }  
    get name(){  
        return _name.get(this);  
    }  
}
```

Utiliser des symbols en tant que clés

```
// Pas réellement privé : getOwnPropertySymbols
const _name = Symbol WeakMap('name');
class Person {
    constructor(name) {
        this[_name] = name;
    }
    get name(){
        return this[_name];
    }
}
```

Combiner avec les pattern ES5

```
//Immediate Function Execution
(function(context){

    let _secret;

    //Zone de déclaration privée
    class Person {

        constructor(name,secret) {
            this.name = name;
            _secret = secret;
            Object.freeze(this); // Option : instance immuable
        }
        tell(){
            return _secret;
        }
    }

    //Option prévient écrasement
    Object.defineProperty(context, "Person", {get : function(){return Person; }});

})(this)
```

Héritage de classes

Le mot-clé `extends`, utilisé dans les déclarations ou les expressions de classes, permet de créer une classe qui hérite d'une autre classe (on parle aussi de « sous-classe » ou de « classe-fille »).

Le mot-clé `super` fait référence de la classe parente.

```
class Animal {  
    constructor(nom) {  
        this.nom = nom;  
    }  
  
    parle() {  
        console.log(this.nom + ' fait du bruit.');//  
    }  
}  
  
class Chien extends Animal {  
    parle() {  
        super.parle();  
        console.log(this.nom + ' aboie.');//  
    }  
}
```

Une classe ECMAScript ne peut avoir qu'une seule classe parente.

Une fonction peut prendre une classe parente en entrée et renvoyer une classe fille qui étend cette classe parente. Cela peut permettre d'émuler **la composition par mix-ins avec ECMAScript**.

```
var MixinDecoratorA = Base => class MixinDecoratorA extends Base {  
    methodA() { }  
};  
  
var MixinDecoratorB = Base => class MixinDecoratorB extends Base {  
    methoB() { }  
};  
  
class Ancestor { }  
class PolyParent extends MixinDecoratorA(MixinDecoratorB(Ancestor)) { }  
  
let o = new PolyParent()  
console.log(o.constructor.name,o.constructor.prototype);
```

Créer une classe statique

```
class Manager {  
    constructor() {  
        return Manager  
    }  
  
    static create(value) {  
        Manager.collection.push(value);  
    }  
  
    static remove(value) {  
        Manager.collection.splice(Manager.collection.indexOf(value), 1);  
    }  
}  
Manager.collection = []  
  
Manager.create();
```

Création de “proxy”. Design pattern.

L'objet `Proxy` permet de redéfinir un comportement sur sur un objet cible (par exemple, l'accès aux propriétés, les affectations, les énumérations, les appels de fonctions, etc.).

Terminologie.

`handler`

Un objet qui contient les trappes qui intercepteront les opérations.

`traps`

Les méthodes qui fournissent l'accès aux propriétés.

`target`

L'objet virtualisé par le proxy. Il est souvent utilisé comme objet de stockage.

Syntaxe.

```
var p = new Proxy(target, handler);
```

Paramètres

`target`

Un objet cible (qui peut être n'importe quel objet, y compris un tableau, une fonction ou voire même un autre proxy) ou une fonction qu'on souhaite envelopper dans un Proxy.

`handler`

Un objet dont les propriétés sont des fonctions qui définissent le comportement du proxy.

```
var handler = {
  get: function(cible, nom){
    return nom in cible?
      cible[nom] :
      37;
  }
};

var p = new Proxy({}, handler);
p.a = 1;
p.b = undefined;

console.log(p.a, p.b); // 1, undefined
console.log('c' in p, p.c); // false, 37
```

```
// Proxy « invisible »

var cible = {};
var p = new Proxy(cible, {});

p.a = 37; // L'opération est transmise à la cible par le proxy

console.log(cible.a); // 37. L'opération a bien été transmise
```

En utilisant un Proxy, il devient simple de valider les valeurs passées à un objet.

```
let validateur = {
  set: function(obj, prop, valeur) {
    if (prop === 'age') {
      if (!Number.isInteger(valeur)) {
        throw new TypeError('Cet age n\'est pas un entier.');
      }
      if (valeur > 200) {
        throw new RangeError('Cet age semble invalide.');
      }
    }
    // Le comportement par défaut : enregistrer la valeur
    obj[prop] = valeur;
  }
};

let personne = new Proxy({}, validateur);

personne.age = 100;
console.log(personne.age); // 100
personne.age = 'jeune';    // Lève une exception
personne.age = 300;        // Lève une exception
```

Un exemple avec toutes les traps

```
var obj = new Proxy({}, {
  "get": function (oTarget, sKey) {
    return oTarget[sKey] || undefined;
  },
  "set": function (oTarget, sKey, vValue) {
    return oTarget[sKey] = vValue;
  },
  "deleteProperty": function (oTarget, sKey) {
    delete oTarget[sKey];
  },
  "enumerate": function (oTarget, sKey) {
    return oTarget.keys();
  },
  "ownKeys": function (oTarget, sKey) {
    return oTarget.keys();
  },
  "has": function (oTarget, sKey) {
    return sKey in oTarget;
  },
  "defineProperty": function (oTarget, sKey, oDesc) {
    return oTarget;
  },
  "getOwnPropertyDescriptor": function (oTarget, sKey) {
    var vValue = oTarget[sKey];
    return vValue ? {
      "value": vValue,
      "writable": true,
      "enumerable": true,
      "configurable": true
    } : undefined;
  },
});
console.log(obj.val1 = "Première valeur");
console.log(obj.val1);

console.log(obj.val1 = "Valeur modifiée");
console.log(obj.val1);
```

Nouveaux types : Set, Weakset, Map, Weakmap.

Set, Weakset

L'objet Set (Ensemble en français) permet de stocker des valeurs uniques, de n'importe quel type, que ce soit des valeurs d'un type primitif ou des objets.

```
new Set([itérable]);
```

itérable

Paramètre optionnel. Si un objet itérable est donné comme argument, l'ensemble de ses éléments sera ajouté au nouvel objet Set. null sera traité comme undefined.

Il est possible d'itérer sur les éléments contenus dans l'objet Set dans leur ordre d'insertion. Une **valeur** donnée ne peut apparaître qu'**une seule fois par Set**.

Propriétés

Set.length La valeur de la propriété length est 0.

get Set[@@species] Le constructeur utilisé pour créer des objets dérivés.

Set.prototype Représente le prototype du constructeur Set. Cela permet d'ajouter des propriétés à tous les objets Set.

Toutes les instances de Set héritent de Set.prototype.

Méthodes par l'exemple :

```
var monSet = new Set();

monSet.add(1);
monSet.add(5);
monSet.add("du texte");

monSet.has(1); // true
monSet.has(3); // false, 3 n'a pas été ajouté à l'ensemble
monSet.has(5); // true
monSet.has(Math.sqrt(25)); // true
monSet.has("Du Texte".toLowerCase()); // true

monSet.size; // 3

monSet.delete(5); // retire 5 du set
monSet.has(5); // false, 5 a été retiré de l'ensemble

monSet.size; // 2, on a retiré une valeur de l'ensemble
```

Itérer sur des ensembles (Set)

```
// On itère sur les différents éléments de l'ensemble
for (let item of monSet) console.log(item);

// On affiche les clés de l'ensemble : 1, "du texte"
for (let item of monSet.keys()) console.log(item);

// On affiche les valeurs de l'ensemble : 1, "du texte"
for (let item of monSet.values()) console.log(item);

// ici on affiche les clés et valeurs de l'ensemble
for (let [clé, valeur] of monSet.entries()) console.log(clé,valeur);

// on convertit l'ensemble en un tableau Array
var monTableau = Array.from(monSet);

// Cela fonctionnera également dans un document HTML
monSet.add(document.body);
monSet.has(document.querySelector("body")); // true
monSet.has(document.delete("body")); // true
monSet.clear();

// convertir un tableau (Array) en ensemble (Set) et vice versa
monSet2 = new Set([1,2,3,4]);
monSet2.size; // 4
[...monSet2]; // [1,2,3,4]

// On peut itérer sur les entrées d'un ensemble avec forEach
mySet.forEach(function(value) {
    console.log(value);
});
```

WeakSet Les principales différences avec l'objet Set sont les suivantes :

Contrairement aux Sets, les WeakSets sont des ensembles **uniquement constitués d'objets** et ne peuvent pas contenir des valeurs de n'importe quel type.

L'objet WeakSet est faible : Les références vers les objets de l'ensemble sont des références faibles. **Si aucune autre référence vers l'objet n'est présente en dehors du WeakSet, l'objet pourra alors être nettoyé** par le ramasse-miette. Cela signifie également qu'on ne peut pas lister les objets contenus à un instant donné dans l'ensemble.

Les objets WeakSets ne sont pas énumérables.

Utiliser l'objet WeakSet

```
var ws = new WeakSet();
var obj = {};  
  
ws.add(window);  
  
ws.has(window); // true
ws.has(obj);    // false, obj n'a pas été ajouté à l'ensemble
ws.add(obj);  
  
ws.delete(window); // window est retiré de l'ensemble
ws.has(window); // false, window a été enlevé
```

Map, Weakmap

L'objet Map représente un dictionnaire, autrement dit une carte de clés/valeurs. N'importe quelle valeur valable en JavaScript (que ce soit les objets ou les valeurs de types primitifs) peut être utilisée comme clé ou comme valeur.

```
new Set([itérable]);
```

itérable

Un tableau Array ou tout autre objet itérable dont les éléments sont des paires clé/valeur (des objets Arrays à deux éléments). Chaque paire clé/valeur sera ajoutée au nouvel objet Map. null est traité comme undefined.

Un objet Map permet de retrouver ses éléments dans leur ordre d'insertion.

Comparaison entre objets et maps

Un Object possède un prototype, il peut donc y avoir certaines clés par défaut.

On peut contourner cela en utilisant map = Object.create(null).

Les clés d'un Object sont des chaînes de caractères, alors que pour une Map ça peut être n'importe quelle valeur.

Il est possible d'obtenir facilement la taille d'une Map. En revanche, pour un Object il faudra compter « manuellement ».

Il est conseillé d'utiliser des Maps plutôt que des objets quand les clés sont inconnues au moment de l'exécution et lorsque toutes les clés sont du même type et que toutes les valeurs sont du même type.

- Est-ce que les clés sont inconnues avant l'exécution ?
- Doit-on les parcourir dynamiquement ?
- Est-ce que toutes les valeurs ont le même type ?
- Est-ce que les clés ne sont pas nécessairement des chaînes ?
- Est-ce que des paires clés-valeurs sont fréquemment ajoutées ou retirées ?
- Faut-il pouvoir utiliser un nombre arbitraire de clés-valeurs ?
- Itère-t-on sur cette collection ?

Utiliser un objet Map

```
var maMap = new Map();

var objetKey = {},
fonctionKey = function () {},
chaineKey = "une chaîne";

// définir les valeurs
maMap.set(chaineKey, "valeur associée à 'une chaîne'");
maMap.set(objetKey, "valeur associée à objetKey");
maMap.set(fonctionKey, "valeur associée à fonctionKey");

maMap.size; // 3

// getting the values
maMap.get(chaineKey);    // "valeur associée à 'une chaîne'"
maMap.get(objetKey);     // "valeur associée à objetKey"
maMap.get(fonctionKey);  // "valeur associée à fonctionKey"

maMap.get("une chaîne"); // "valeur associée à 'une chaîne'"
                        // car chaineClé === 'une chaîne'
maMap.get({});          // indéfini car objetClé !== {}
maMap.get(function() {}); // indéfini car fonctionClé !== function () {}

for (var [key, valeur] of maMap.entries()) {
  console.log(key + " = " + valeur);
}

var tableauKeyValue = [["Key1", "valeur1"], ["Key2", "valeur2"]];
var maMap = new Map(tableauKeyValue);
maMap.get("Key1");
```

Symbols

Un Symbole est un type de donnée primitif dont les instances sont uniques et immuables. Dans certains langages de programmation, ils sont également appelés atomes.

L'objet Symbol est un conteneur objet implicite pour le type de données primitif symbole.

`Symbol([description])`

description (Facultatif)

Une chaîne de caractères optionnelle. Correspond à une description du symbole, elle peut être utile pour déboguer (mais pas pour accéder au symbole).

```
var sym1 = Symbol();
var sym2 = Symbol("Hello");
var sym3 = Symbol("World");
var sym3 = Symbol("World");

//Chaque symbole est unique
Symbol("World") === Symbol("World"); // false

var sym = new Symbol(); // TypeError
```

Utilisant l'opérateur `new`, entraînera une exception `TypeError`.

Symboles partagés et registre global des symboles

Pour créer des symboles qui soient disponibles pour différents fichiers et appartenant à l'environnement global, il faut utiliser les méthodes **Symbol.for()** et **Symbol.keyFor()** afin de définir et de récupérer les symboles listés dans le registre global.

La méthode **Symbol.keyFor(sym)** permet de récupérer la clé d'un symbole donné qui est partagé via le registre global des symboles.

Afin d'éviter des conflits entre les clés des symboles globaux liés à votre application il peut être judicieux de les préfixer :

```
Symbol.for("Hello"); // crée un nouveau symbole global
Symbol.for("Hello"); // renvoie le symbole déjà existant

// Globalement on a un symbole par clé, localement non
Symbol.for("Hello") === Symbol.for("Hello"); // true
Symbol("Hello") === Symbol("Hello"); // false

// La clé sert également de description
var sym = Symbol.for("Hello");
sym.toString(); // "Symbol(mario)"

var symboleGlobal = Symbol.for("World"); // on crée un symbole global
Symbol.keyFor(symboleGlobal); // "t"
```

Symboles

- Un moyen supplémentaire de représenter les états.
- Accessible via : `Object.getOwnPropertySymbols()` ou `Object.getOwnSymbols()`
- Incorecible
- Copiable `Object.assign(newObject, objectWithSymbols)`

Quels usages ?

Valeur unique

```
log.levels = {
  DEBUG: Symbol('debug'),
  INFO: Symbol('info'),
  WARN: Symbol('warn'),
};
```

Meta Données

```
var size = Symbol('size');
class Collection {
    constructor() {
        this[size] = 0;
    }

    add(item) {
        this[this[size]] = item;
        this[size]++;
    }

    static sizeOf(instance) {
        return instance[size];
    }
}

var x = new Collection();
Collection.sizeOf(x);
x.add('foo');
Collection.sizeOf(x);
```

Afin d'éviter des conflits entre les clés des symboles globaux liés à votre application il peut être judicieux de les préfixer :

```
Symbol.for("mdn.Hello");
Symbol.for("google.Hello");
```

Symboles connus

En plus des symboles que vous pouvez créer, JavaScript possède certains symboles natifs représentant des aspects internes du langage qui, pour ECMAScript 5 et les versions précédentes, n'étaient pas exposées aux développeurs. Il est possible d'accéder à ces symboles en utilisant les propriétés suivantes :

Symbol.iterator

Une méthode qui renvoie l'itérateur par défaut d'un objet. Ce symbole est utilisé par la boucle `for...of`.

Symboles liés aux expressions rationnelles

Symbol.match

Une méthode qui fait correspondre une expression rationnelle avec une chaîne de caractères. Elle est aussi utilisée pour déterminer si un objet peut être utilisé comme une expression rationnelle.

Symbol.replace

Une méthode qui remplace les sous-chaînes correspondantes dans une chaîne de caractères. Utilisée par `String.prototype.replace()`.

Symbol.search

Une méthode qui renvoie l'indice d'une chaîne de caractères pour lequel on a une correspondance avec une expression rationnelle. Utilisée par `String.prototype.search()`.

Symbol.split

Une méthode qui découpe la chaîne à l'indice donné par la correspondance avec une expression rationnelle. Utilisée par `String.prototype.split()`.

Autres symboles

Symbol.hasInstance

Une méthode qui permet de déterminer si un constructeur reconnaît un objet comme son instance. Utilisé par `instanceof`.

Symbol.isConcatSpreadable

Une valeur booléenne qui indique si un objet devrait être réduit à la concaténation des éléments de son tableau via `Array.prototype.concat()`.

Symbol.species

Un constructeur utilisé pour construire des objets dérivés.

Symbol.toPrimitive

Spécifié comme `@@toPrimitive`. Une méthode qui convertit un objet en sa valeur primitive.

Symbol.toStringTag

Spécifié comme @@toStringTag. Une chaîne de caractères utilisée pour la description d'un objet. Ce symbole est utilisé par Object.prototype.toString().

Les Symbole connus constituent la possibilité pour le développeur de surcharger les comportement par défaut du moteur pour un objet spécifique.

```
//Héritage
class MonArray extends Array {
    // On surcharge species avec le constructeur parent Array
    static get [Symbol.species]() { return Array; }
}
var a = new MonArray(1,2,3);
var mapped = a.map(x => x * x);

console.log(mapped instanceof MonArray); // false
console.log(mapped instanceof Array);    // true

//Coercion
var obj2 = {
    [Symbol.toPrimitive](hint) {
        if (hint === "number") {
            return 10;
        }
        if (hint === "string") {
            return "coucou";
        }
        return true;
    }
};
console.log(+obj2);      // 10      -- hint vaut "number"
console.log(` ${obj2}`); // "coucou" -- hint vaut "string"
console.log(obj2 + "");

// Itération
var monItérable = {}
monItérable[Symbol.iterator] = function* () {
    yield 1;
    yield 2;
    yield 3;
};
[...monItérable] // [1, 2, 3]
```

Objets natifs héritables.

Avec ES6, les objets natifs tels que Array, Date et DOMElements peuvent être étendus.

La construction d'objet se décompose en deux phases :

1. Appel du constructeur pour allouer l'objet, l'initialisation d'un comportement spécial
2. Invocation du constructeur sur une nouvelle instance pour initialiser

Les types natifs exposent leur constructeur explicitement.

```
// Pseudo-code of Array
class Array {
    constructor(...args) { /* ... */ }
    static [Symbol.create]() {
        // Initialisation[[DefineOwnProperty]]
    }
}

// Héritage de Array
class MyArray extends Array {
    constructor(...args) { super(...args); }
}

// Deux phase 'new':
// 1) Appel @@create pour allouer l'objet
// 2) Création de l'instance
var arr = new MyArray();
arr[1] = 12;
arr.length == 2
```

{js}

class est juste un mot clé donnant plus de lisibilité.

N'attendez pas et n'essayez pas de reproduire le comportement d'un langage tel que JAVA.

L'Héritage est simplifié, est-il nécessaire ?.

Set , Map et Symbol permettront une optimisation de la mémoire.



ES6

Nouvelles API JavaScript avec ES6.

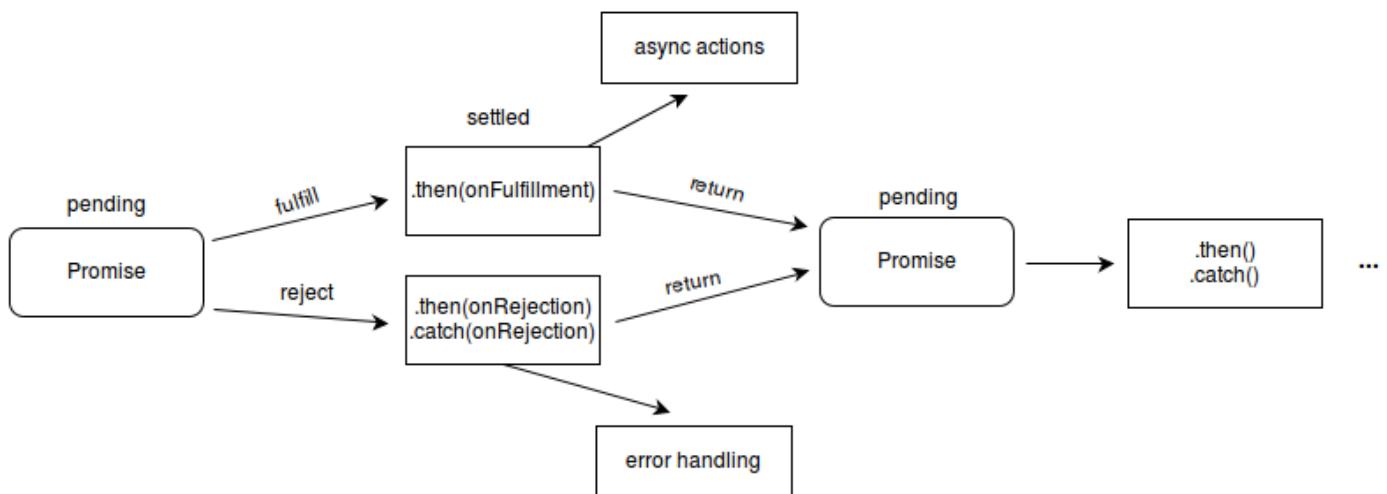
6/ Nouvelles API JavaScript avec ES6.

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 asynchrones

```
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))
```

Object API : revisiter les méthodes.

Les Constructeur natif connus sont enrichi de nouvelle méthodes **static** ou **prototypal**

Object : méthodes statiques

Object.assign

La méthode Object.assign() est utilisée afin de copier les valeurs de toutes les propriétés directes (non héritées) d'un objet qui sont énumérables sur un autre objet cible. Cette méthode renvoie l'objet cible.

```
var o1 = { a: 1 };
var o2 = { [Symbol('Hello')]: 2 };
var o3 = { c: 3 };

var obj = Object.assign(o1, o2, o3);
console.log(obj); // { a: 1, c: 3, [Symbol("Hello")]: 2 }
console.log(o1);
```

Object.is

La méthode Object.is() permet de déterminer si deux valeurs sont les mêmes.

```
Object.is("Hello", "Hello");      // true
Object.is(window, window);       // true

Object.is("Hello", "World");     // false
Object.is([], []);              // false

var test = {a: 1};
Object.is(test, test);          // true

Object.is(null, null);          // true
```

Object.getOwnPropertySymbols

La méthode `Object.getOwnPropertySymbols()` renvoie un tableau contenant tous les symboles des propriétés trouvées directement sur un objet donné.

```
var obj = {};
var a = Symbol("a");
var b = Symbol.for("b");

obj[a] = "symboleLocal";
obj[b] = "symboleGlobal";

var objectSymboles = Object.getOwnPropertySymbols(obj);
```

Object.setPrototypeOf

La méthode `Object.setPrototypeOf()` définit le prototype (autrement dit la propriété interne [Prototype \(/Prototype/\)](#)) d'un objet donné avec un autre objet ou null.

A éviter pour de sraison de performance

String : méthodes statiques

String.raw

La méthode statique String.raw() est une fonction d'étiquettage (tag function) pour les gabarits de chaînes de caractères . Cette fonction permet d'obtenir la chaîne brute pour un gabarit.

```
let nom = "Bob";
String.raw`Hi\n${nom}!`;
// "Hi\nBob!", Les remplacements sont effectués.
```

String.fromCharCode

La méthode statique String.fromCharCode() renvoie une chaîne de caractères créée à partir d'un suite de codets.

```
String.fromCharCode(42);      // "*"
String.fromCharCode(65, 90);   // "AZ"
String.fromCharCode(0x404);    // "\u0404"
String.fromCharCode(0x2F804);  // "\uD87E\uDC04"
String.fromCharCode(194564);   // "\uD87E\uDC04"
String.fromCharCode(0x1D306, 0x61, 0x1D307)
```

String : méthodes sur le prototype

String.prototype.charCodeAt

La méthode codePointAt() renvoie un entier positif qui correspond au code Unicode du caractère de la chaîne à la position donnée.

```
'ABC'.codePointAt(1);        // 66
'\uD800\uDC00'.codePointAt(0); // 65536
```

String.prototype.normalize

La méthode **normalize(form)** permet de renvoyer la forme normalisée Unicode d'une chaîne de caractères (si la valeur n'est pas une chaîne de caractères, elle sera convertie).

- NFC - Défaut - Normalization Form Canonical Composition.
- NFD - Normalization Form Canonical Decomposition.
- NFKC - Normalization Form Compatibility Composition.
- NFKD - Normalization Form Compatibility Decomposition.

```
var a= '\u00F6'
var b= 'o\u0308'

console.log(a,b,a == b )

var a= '\u00F6'.normalize()
var b= 'o\u0308'.normalize()

console.log(a,b,a == b )
```

String.prototype.repeat

La méthode `repeat()` construit et renvoie une nouvelle chaîne de caractères qui contient le nombre de copies demandée de la chaîne de caractères sur laquelle la méthode a été appelée, concaténées les unes aux autres.

```
"abc".repeat(-1)      // RangeError
"abc".repeat(0)        // ""
"abc".repeat(1)        // "abc"
"abc".repeat(2)        // "abcabc"
"abc".repeat(3.5)      // "abcabcabc" (Le compteur est converti en un nombre entier)
"abc".repeat(1/0)      // RangeError
```

String.prototype.startsWith

La méthode `startsWith()` renvoie un booléen indiquant si la chaîne de caractères commence par la deuxième chaîne de caractères fournie en argument.

```
var str = "Être, ou ne pas être : telle est la question.";  
  
console.log(str.startsWith("Être"));           // true  
console.log(str.startsWith("pas être"));        // false  
console.log(str.startsWith("pas être", 12));    // true
```

String.prototype.endsWith

La méthode `endsWith()` renvoie un booléen indiquant si la chaîne de caractères se termine par la deuxième chaîne de caractères fournie en argument.

```
var str = "Être, ou ne pas être : telle est la question.";  
  
console.log(str.endsWith("question."));         // true  
console.log(str.endsWith("pas être"));          // false  
console.log(str.endsWith("pas être", 20));       // true
```

String.prototype.includes

La méthode `includes()` détermine si une chaîne de caractères est contenue dans une autre et renvoie true ou false selon le cas de figure.

```
var str = "Être ou ne pas être, telle est la question.";  
  
console.log(str.includes("Être"));           // true  
console.log(str.includes("question"));        // true  
console.log(str.includes("pléonasme"));       // false  
console.log(str.includes("Être", 1));          // false  
console.log(str.includes("ÊTRE"));            // false
```

Array : méthodes statiques

Array.from

La méthode **Array.from()** permet de créer une nouvelle instance d'Array à partir d'un objet itérable ou semblable à un tableau.

Avec ES6, la syntaxe de classe permet d'avoir des sous-classes pour les objets natifs comme pour les objets définis par l'utilisateur. Ainsi, les méthodes statiques de classe comme `Array.from` sont héritées par les sous-classes d'Array et créent de nouvelles instances de la sous-classe d'Array.

Paramètres

arrayLike Un objet semblable à un tableau ou bien un objet itérable dont on souhaite créer un tableau, instance d'Array.

fonctionMap Argument optionnel, une fonction à appliquer à chacun des éléments du tableau.

thisArg Argument optionnel. La valeur à utiliser pour `this` lors de l'exécution de la fonction `fonctionMap`.

```
// créer une instance d'Array à partir de l'objet arguments qui est semblable à un tableau
function foo() {
  return Array.from(arguments);
}

foo(1, 2, 3); // [1, 2, 3]

// Ça fonctionne avec tous les objets itérables...
var s = new Set(["toto", window]);
Array.from(s); // ["toto", window]

var m = new Map([[1, 2], [2, 4], [4, 8]]);
Array.from(m); // [[1, 2], [2, 4], [4, 8]]

Array.from("toto"); // ["t", "o", "t", "o"]

// En utilisant une fonction fléchée pour remplacer map
Array.from([1, 2, 3], x => x + x); // [2, 4, 6]

// Pour générer une séquence de nombres
Array.from({length: 5}, (v, k) => k); // [0, 1, 2, 3, 4]
```

Array.of

La méthode Array.of() permet de créer une nouvelle instance d'objet Array avec un nombre variable d'argument, quels que soient leur nombre ou leur type.

La différence avec la méthode Array.of() et le constructeur Array se situe dans la façon de gérer les arguments entiers. Ainsi Array.of(42) créera un tableau avec un seul élément (42) alors que Array(42) créera un tableau contenant 42 éléments, chacun valant

```
Array.of(1);          // [1]
Array.of(1, 2, 3);    // [1, 2, 3]
Array.of(undefined); // [undefined]
```

Array : méthodes sur le prototype

Array.prototype.copyWithin

La méthode copyWithin() permet de copier une suite d'éléments à l'intérieur d'un tableau à partir d'un indice cible. L'argument fin est facultatif et sa valeur par défaut correspond à la taille du tableau.

copyWithin est une méthode qui modifie l'objet courant. En effet, elle modifiera l'objet this avant de le renvoyer et ne créera pas un tableau à part pour le résultat

```
[1, 2, 3, 4, 5].copyWithin(0, 3); // [4, 5, 3, 4, 5]
[1, 2, 3, 4, 5].copyWithin(0, 3, 4); // [4, 2, 3, 4, 5]
[1, 2, 3, 4, 5].copyWithin(0, -2, -1); // [4, 2, 3, 4, 5]
[].copyWithin.call({length: 5, 3: 1}, 0, 3); // {0: 1, 3: 1, Length: 5}
```

Array.prototype.find

La méthode `find()` renvoie une valeur contenue dans le tableau si un élément du tableau respecte une condition donnée par la fonction de test passée en argument. Sinon, la valeur `undefined` est renvoyée.

La méthode `find` exécute la fonction `callback` une fois pour chaque élément présent dans le tableau jusqu'à ce qu'elle retourne une valeur vraie (qui peut être convertie en `true`). Si un élément est trouvé, `find` retourne immédiatement la valeur de l'élément.

find ne modifie pas le tableau à partir duquel elle est appelée.

Array.prototype.findIndex

La méthode `findIndex()` renvoie l'indice d'un élément du tableau qui satisfait une condition donnée par une fonction. Si la fonction renvoie faux pour tous les éléments du tableau, le résultat vaut `-1`.

```
var inventaire = [
  {nom: 'pommes', quantité: 2},
  {nom: 'bananes', quantité: 0},
  {nom: 'cerises', quantité: 5},
];

function trouveCerises(fruit) {
  return fruit.nom === 'cerises';
}

console.log(inventaire.find(trouveCerises));
console.log(inventaire.findIndex(trouveCerises));
```

Array.prototype.fill

La méthode `fill()` remplit tout les éléments d'un tableau entre deux index avec une valeur statique.

```
arr.fill(valeur[, début = 0[, fin = this.length]])
```

```
[1, 2, 3].fill(4);           // [4, 4, 4]
[1, 2, 3].fill(4, 1);       // [1, 4, 4]
[1, 2, 3].fill(4, 1, 2);    // [1, 4, 3]
[1, 2, 3].fill(4, 1, 1);    // [1, 2, 3]
[1, 2, 3].fill(4, -3, -2);  // [4, 2, 3]
[1, 2, 3].fill(4, NaN, NaN); // [1, 2, 3]
Array(3).fill(4);           // [4, 4, 4]
[].fill.call({length: 3}, 4); // {0: 4, 1: 4, 2: 4, length: 3}
```

Array.prototype.keys

La méthode `keys()` renvoie un nouveau **Array Iterator** qui contient les clefs pour chaque indice du tableau.

```
var arr = ["a", "b", "c"];
var itérateur = arr.keys();

console.log(itérateur.next()); // { value: 0, done: false }
console.log(itérateur.next()); // { value: 1, done: false }
console.log(itérateur.next()); // { value: 2, done: false }
console.log(itérateur.next()); // { value: undefined, done: true }

var arr = ["a", , "c"];
var clésCreuses = Object.keys(arr);
var clésDenses = [...arr.keys()];
console.log(clésCreuses); // ["0", "2"]
console.log(clésDenses); // [0, 1, 2]
```

Array.prototype.values

La méthode **values()** renvoie un nouvel objet **Array Iterator** qui contient les valeurs pour chaque indice du tableau.

```
var arr = ['w', 'y', 'k', 'o', 'p'];
var eArr = arr.values();
// votre navigateur doit supporter les boucles for..of
// et les variables définies avec let
for (let lettre of eArr) {
  console.log(lettre);
}

var arr = ['w', 'y', 'k', 'o', 'p'];
var eArr = arr.values();
console.log(eArr.next().value); // w
console.log(eArr.next().value); // y
console.log(eArr.next().value); // k
console.log(eArr.next().value); // o
console.log(eArr.next().value); // p
```

Array.prototype.entries

La méthode **entries()** renvoie un nouvel objet de type **Array Iterator** qui contient le couple clef/valeur pour chaque éléments du tableau.

```
var arr = ["a", "b", "c"];
var eArr = arr.entries();

console.log(eArr.next().value); // [0, "a"]
console.log(eArr.next().value); // [1, "b"]
console.log(eArr.next().value); // [2, "c"]
```

Number : méthodes sur le prototype

Number.isFinite

La méthode Number.isFinite() permet de déterminer si la valeur fournie est un nombre fini.

```
Number.isFinite(Infinity); // false
Number.isFinite(NaN); // false
Number.isFinite(-Infinity); // false

Number.isFinite(0); // true
Number.isFinite(2e64); // true

Number.isFinite("0"); // false

Number.isFinite(null); // false
```

Number.isInteger

La méthode Number.isInteger() permet de déterminer si l'argument est un nombre entier.

```
Number.isInteger(0.1); // false
Number.isInteger(1); // true
Number.isInteger(-100000); // true
Number.isInteger(Math.PI); // false
Number.isInteger(-Infinity); // false
Number.isInteger(true); // false
Number.isInteger(NaN); // false
Number.isInteger(0); // true
Number.isInteger("10"); // false
```

Number.isSafeInteger

La méthode `Number.isSafeInteger()` permet de déterminer si la valeur, passée en argument, est un entier représentable correctement en JavaScript (c'est-à-dire un nombre compris entre $-(2^{53}-1)$ et $2^{53}-1$).

```
Number.isSafeInteger(3);          // true
Number.isSafeInteger(Math.pow(2, 53)) // false
Number.isSafeInteger(Math.pow(2, 53) - 1) // true
Number.isSafeInteger(NaN);        // false
Number.isSafeInteger(Infinity);   // false
Number.isSafeInteger("3");        // false
Number.isSafeInteger(3.1);        // false
Number.isSafeInteger(3.0);        // true
```

Number.isNaN

La méthode Number.isNaN() permet de déterminer si la valeur passée en argument est NaN.

Cette version est plus robuste que la méthode de l'objet global isNaN.

```
Number.isNaN(NaN); // true
Number.isNaN(Number.NaN); // true
Number.isNaN(0 / 0) // true

// tout le reste renverra : false
Number.isNaN(undefined);
Number.isNaN({});

Number.isNaN(true);
Number.isNaN(null);
Number.isNaN(37);

Number.isNaN("37");
Number.isNaN("37.37");
Number.isNaN("");
Number.isNaN(" ");
Number.isNaN("NaN");
Number.isNaN("blabla");
```

Math est également enrichi de nombreuses méthodes.

{js}

Les Promises apporte une solution à l'organisation de code asynchrone.

Les nouvelles Méthodes de Array et Object sont optimisées.



ES6

Modules JavaScript.

7/ Modules JavaScript.

Un module est simplement un fichier JavaScript exportant des valeurs, qui peuvent ensuite être importés par d'autres modules.

Un module loader permet de charger dynamiquement des modules, et assure également le suivi de tous les modules chargés dans un registre de module.

Asynchronous Module Definition ou CommonJS.

Le besoin de modularisation et d'encapsulation logique à trouver plusieurs réponses technique, bien avant la proposition standardisée en ES6.

On distingue

Global Module global.

CJS Common JavaScript Module : implémenté par node.js

AMD Asynchronous Module Définition JavaScript Module : implémenté par node.js

System La proposition ES6

UMD Universal Module Définition

Module global

Exposition d'une API dans l'espace global.

```
(function IIFE(context){  
    context.API = {  
        value:true  
        logic:logic  
    }  
    function logic(){  
    }  
})(this);
```

Common JavaScript Module

Implémenté par node.js. Synchrone. Portable côté navigateur avec **browserify**
Export nommés.

[CommonJS \(http://wiki.commonjs.org/wiki/CommonJS\)](http://wiki.commonjs.org/wiki/CommonJS)

S'appuie sur l'implémentation de l'objet **module.exports** et de la méthode **require()**

Export :

```
module.exports.foo = function(){
    console.log('foo')
}
```

Consommation :

```
var foo = require('foo')

foo();
```

Asynchronous Module Définition

Reprend la logique de modules CJS en ajoutant la dimension asynchrone (réseau)

Implémentation connue: requirejs

[AMD \(https://github.com/amdjs/amdjs-api/wiki/AMD\)](https://github.com/amdjs/amdjs-api/wiki/AMD)

S'appuie sur méthode de définition **define()** et de résolution **require()** de dépendances exécuter par un **loader**

Export :

```
define('myModule', ['dep1', 'dep2'], function (dep1, dep2) {  
  
    //Export logic  
    return function () {};  
});
```

Consommation :

```
// main.js  
requirejs(['jquery', 'myModule'],  
    function  ($,myModule,) {  
        //jQuery, canvas and the app/sub module are all  
        //loaded and can be used here now.  
    }  
);
```

La librairie require.js (ou le loader choisit) résoud les chargement.

Après son propre chargement **require.js** charge le point d'entrée définit par son attribut **data-main**

```
<script data-main="scripts/main.js" src="scripts/require.js"></script>
```

Malheureusement cette solution demande beaucoup de configuration.

Universal Module Définition

JavaScript Pattern visant permettre la compatibilité entre les différentes définition de modules. (AMD,CJS...)

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD
        define(['jquery'], factory);
    } else if (typeof exports === 'object') {
        // Node, CommonJS-Like
        module.exports = factory(require('jquery'));
    } else {
        // Browser globals (root is window)
        root.returnExports = factory(root.jQuery);
    }
}(this, function ($) {
    // methods
    function myFunc(){};

    // exposed public method
    return myFunc;
}));
```

Inconvénient ajoute beaucoup de **overhead**

[SystemJS \(<https://github.com/systemjs/systemjs>\)](https://github.com/systemjs/systemjs) est une élégante solution de chargement supportant les Modules ES6.

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 variable ou fonction est réutilisé comme nom d'export.

Mais il est possible de redéfinir 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 jQue

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

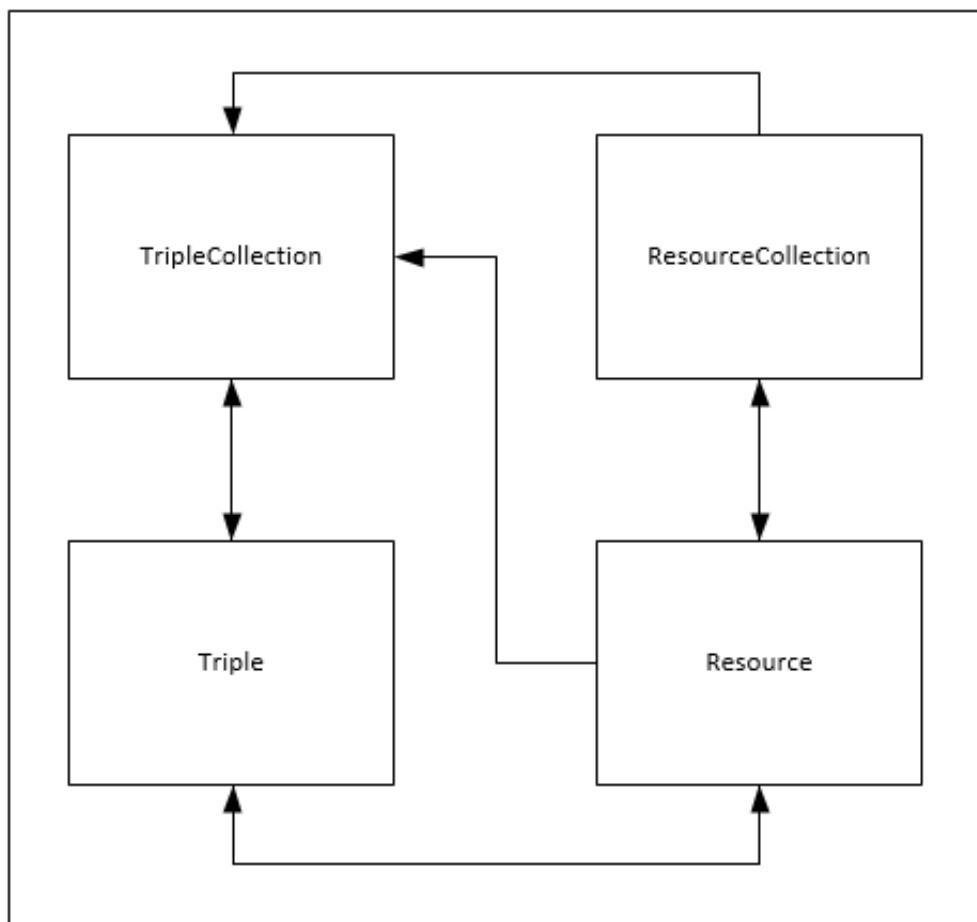
export default 42;                            // exporte par défaut de variable
export default function foo() {};              // exporte par défaut de function

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
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
```

Gestion et résolution des dépendances.

AMD, CommonJS et ES6 traitent les dépendances circulaires différemment. Cela implique l'analyse de l'arbre de dépendance et des couches alternées de modules ES6 / non-ES6 avec des références circulaires dans chaque couche pour lier. Les couches sont ensuite reliées individuellement, avec le traitement de la référence circulaire appropriée se fait dans chaque couche.

Ceci permet à des références CommonJS circulaires d'interagir avec des références circulaires ES6.



Chargement dynamique.

Le Module Loader ES6 va chercher la source, déterminer les dépendances, et attendre jusqu'à ce que ces dépendances soient chargées avant d'exécuter le module.

Le chargement CommonJS via un module de ES6, s'appuie sur l'analyse statique les déclarations, et est seulement exécuté une fois les dépendances chargées.

Cette cohabitation exclue les import dynamique CJS

```
if (condition) require('some' + 'name') ;  
var mod = (condition)? require('moduleA') : require('moduleB') ;
```

ES6 imports are declarative and meant for static analysis. They cannot be dynamic

{js}

Choisissez un système de module cohérent : CJS/ES6.

Utilisez un wrapper (browserify) plutôt qu'une implémentation UMD manuelle.

JSPM est un très bon “Module Loader” (à date 03/2016).

Définissez un stratégie : bundle, modulaire ou mixte.



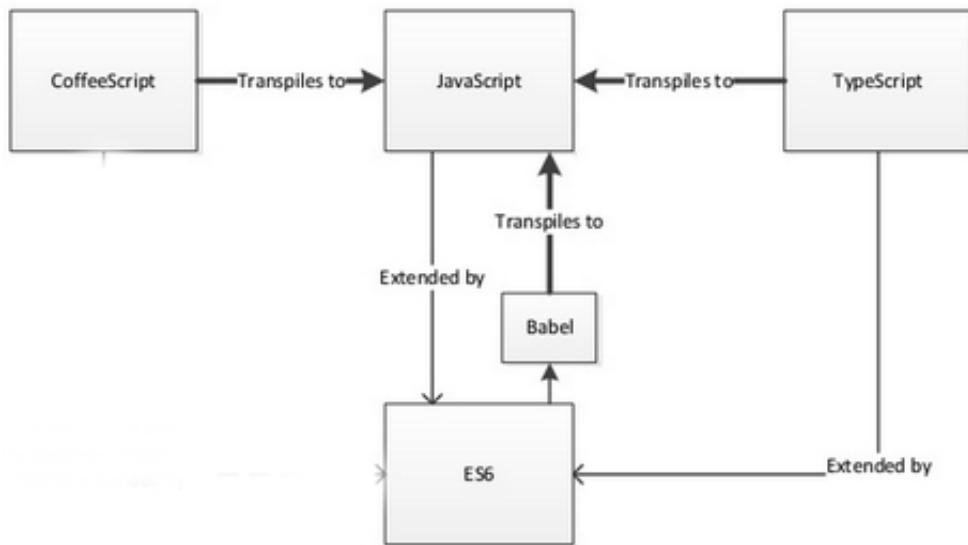
ES6

Mise en production.

8/ Mise en production.

A date (mars 2016) le system natif de chargement n'est pas implémenté dans les navigateur.

De plus les navigateurs ne supporte pas uniformément la syntaxe ES6.



Pour anticiper la migration de code et bénéficier de la syntaxe ES6 il faut choisir un “transpiler” et un système de chargement de module.

[es6-module-loader](https://github.com/ModuleLoader/es6-module-loader) (<https://github.com/ModuleLoader/es6-module-loader>) est un polyfill utilisable en production.

Stratégie : “transpileur”, “package manager” et système de module.

On peut distinguer deux approches :

Le **bundle** basé sur une phase de préparation de ressources concaténés en un seul fichier.

Avantage : garde le développement modulaire tout en diminuant le nombre de requête nécessaire.

Le **runtime** basé sur un “module loader” assurant le chargement de fichier.

Avantage : préserve la modularité en production anticipe HTTP2.

Un **package manager** est un outil permettant de préparer votre code pour sa diffusion et sa réutilisation.



Choix du “transpiler” : présentation des solutions.

Il existent beaucoup de “transpiler” capable de convertir du code ES6 en ES5

Référence

[Babel \(<https://github.com/babel/babel>\)](https://github.com/babel/babel)

[Traceur compiler \(<https://github.com/google/traceur-compiler>\)](https://github.com/google/traceur-compiler)

* [TypeScript \(<http://www.typescriptlang.org/>\)](http://www.typescriptlang.org/)

A noter Ces trois références principales existent sous la forme de tâche pour gulp et grunt

Avec browserify

- [es6ify \(<https://github.com/thlorenz/es6ify>\)](https://github.com/thlorenz/es6ify)
- [babelify \(<https://github.com/babel/babelify>\)](https://github.com/babel/babelify)
- [es6-transpiler \(<https://github.com/termi/es6-transpiler>\)](https://github.com/termi/es6-transpiler)

Modules

- Square's [es6-module-transpiler \(<https://github.com/esnext/es6-module-transpiler>\)](https://github.com/esnext/es6-module-transpiler)

Autres

Facebook's [regenerator \(<https://github.com/facebook/regenerator>\)](https://github.com/facebook/regenerator)

Facebook's [jstransform \(<https://github.com/facebookarchive/jstransform>\)](https://github.com/facebookarchive/jstransform)

[Et encore bien d'autre \(<https://github.com/addyosmani/es6-tools>\)](https://github.com/addyosmani/es6-tools)

Mise en oeuvre de TypeScript, Traceur, Babel.

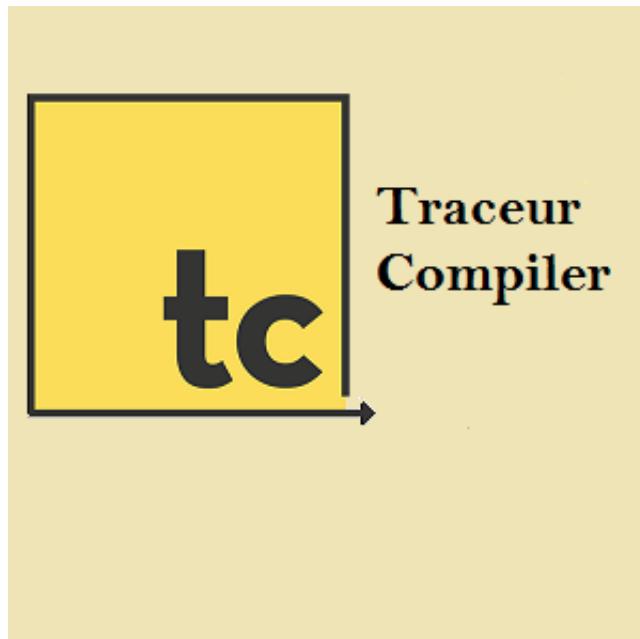
Installation

```
$> npm init  
$> npm install --global typescript traceur jspm yo gulp-cli  
$> npm i -g generator-traceur-gulp generator-babel generator-jspm-es6-quick generator-
```

A noter babel peut fonctionner avec beaucoup d'[environnements](#)
(<https://babeljs.io/docs/setup/>)

Traceur.

Niveau de support 57%



Mise en oeuvre : avec le generator-traceur-gulp

```
yo traceur-gulp
```

L'avenir de traceur ne semble pas certain...

TypeScript.

Niveau de support 60%



Mise en oeuvre

```
tsc file.ts  
tsc --watch
```

TypeScript : offre en plus de la syntaxe ES6, le support de type, d'interface...
[référence \(http://www.typescriptlang.org/docs/handbook/\)](http://www.typescriptlang.org/docs/handbook/)

Configuration tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "out": "../built/local/tsc.js",
    "sourceMap": true
  },
  "exclude": [
    "node_modules",
    "wwwroot"
  ]
}
```

Babel.

Niveau de support 74%



Mise en oeuvre : avec le generator-babel

```
$> yo babel
```

Babel présente le plus fort niveau de support ES6

Mise en oeuvre : avec jspm

```
$> cd my-project  
$> npm install jspm --save-dev  
$> jspm init
```

Faire des installation :

```
$> jspm install npm:lodash-node  
$> jspm install github:components/jquery  
$> jspm install jquery
```

Bootstrap :

```
<!doctype html>
<script src="jspm_packages/system.js"></script>
<script src="config.js"></script>
<script>
  System.import('lib/main.js');
</script>
```

Bundle :

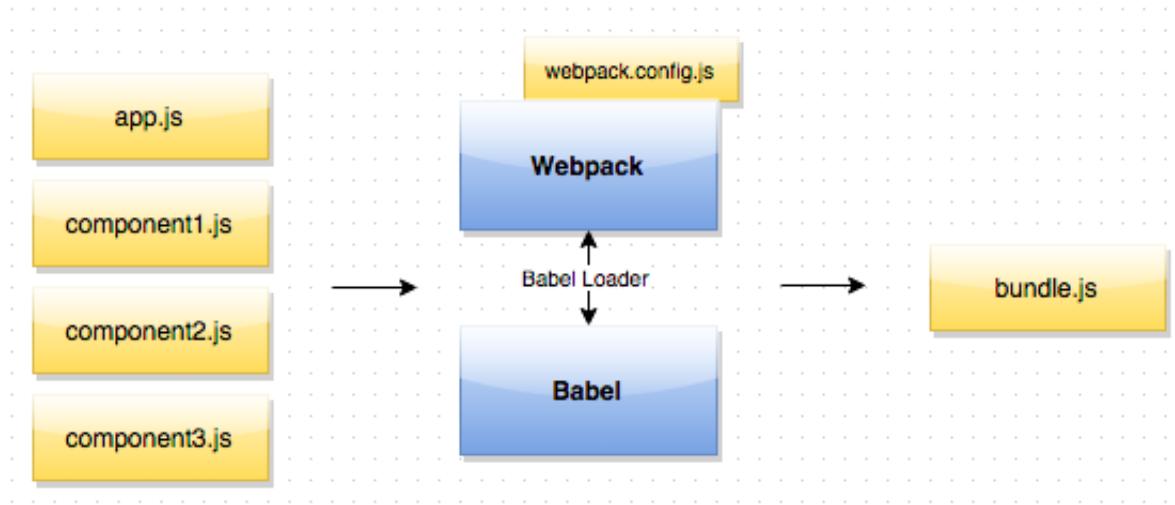
```
$> jspm bundle lib/main --inject
```

Pourquoi “packager” son code ? Avantages et solutions.

Packger son code permet de distribuer simplement l'ensemble des ressources sous la forme d'un fichier(parfois plusieurs).

Les avantages évidents sont :

- Distribution facilité
- Référencement sur les plateforme connues
- Maintenance et versioning faciliter



Solution

- [Rollup \(<http://rollupjs.org/>\)](http://rollupjs.org/)
- [webpack \(<https://webpack.github.io/>\)](https://webpack.github.io/)
- [jspm \(<http://jspm.io/>\)](http://jspm.io/)
- [npm \(<https://www.npmjs.com/>\)](https://www.npmjs.com/)
- [bower \(<http://bower.io/>\)](http://bower.io/)
- [browserify \(<http://browserify.org/>\)](http://browserify.org/)

Création de package avec npm, bower et jspm.

Chaque système demande de suivre une procédure différente :

npm [url \(<https://docs.npmjs.com/getting-started/publishing-npm-packages>\)](https://docs.npmjs.com/getting-started/publishing-npm-packages)

bower [url \(<http://bower.io/docs/creating-packages/>\)](http://bower.io/docs/creating-packages/)

jspm [url \(<http://jspm.io/docs/publishing-packages.html>\)](http://jspm.io/docs/publishing-packages.html)

Choisir entre RequireJS, Browserify, WebPack et SystemJS.

RequireJS ne constitue plus une solution.

Bowserify est un bon outil de bundle mais il conviendra d'utiliser **es6fy** ou **babelify**

```
npm i -g browserify  
npm install --save-dev babelify  
browserify script.js -o bundle.js -t [ babelify --presets [ es2015 ] ]
```

WebPack est un utilitaire très riche mais demandera un temps d'adaptation

[url \(<https://webpack.github.io/docs/cli.html>\)](https://webpack.github.io/docs/cli.html)

```
npm install webpack -g  
npm init  
npm install webpack --save-dev  
webpack <entry> <output>
```

Jspm et SystemsJS fonctionne ensemble:

Création de module autonome.

```
$> jspm bundle-sfx lib/main --inject
```

Interactions avec les autres outils (linting, test...).

En résumé vous avez le choix de composer le workflow qui vous correspondra au mieux à votre habitude de travail. Certain outil favorise une phase de configuration.

Tout ses outils (mis à part les modules loader) interagissent ensemble par le biais de contributions.

Essayez d'installer : **generator-modern-web-dev**

{js}

Prenez le temps d'expérimenter différent build système sur votre chaîne de production.

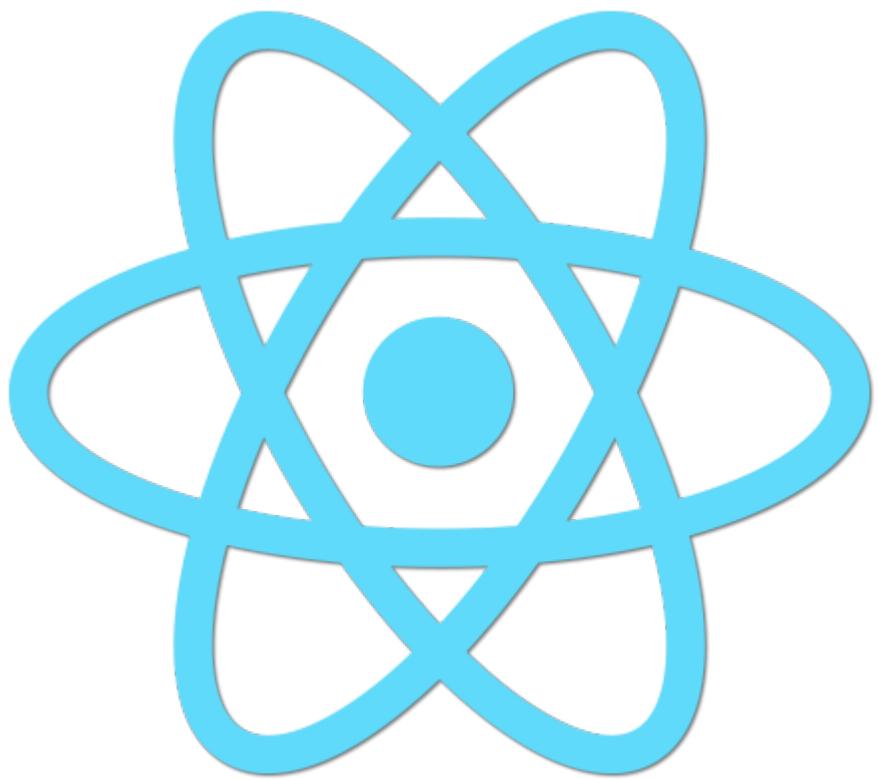
Favoriser la convention pour rester indépendant.

L'écosystème est npm capricieux : contrôler vos versions d'utilitaires.

start:rm * -r' ?. -> nodesecurity.io

Maintenez votre propre version de vos utilitaires.

ES6



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 compatibilité JavaScript

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

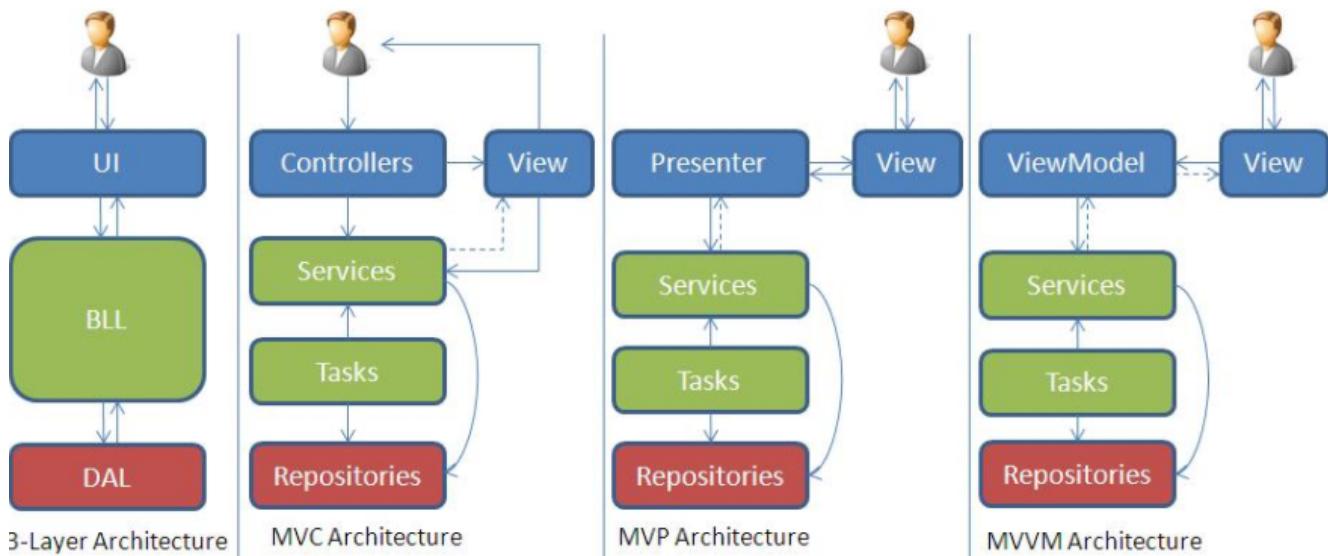
Référence JavaScript

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

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

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

Design patterns applicatifs classiques. Limitations des applications JavaScript.



Architecture 3 tiers

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

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

C'est une extension du modèle client-serveur.

Architecture MVC (Modèle-Vue-Contrôleur)

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

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

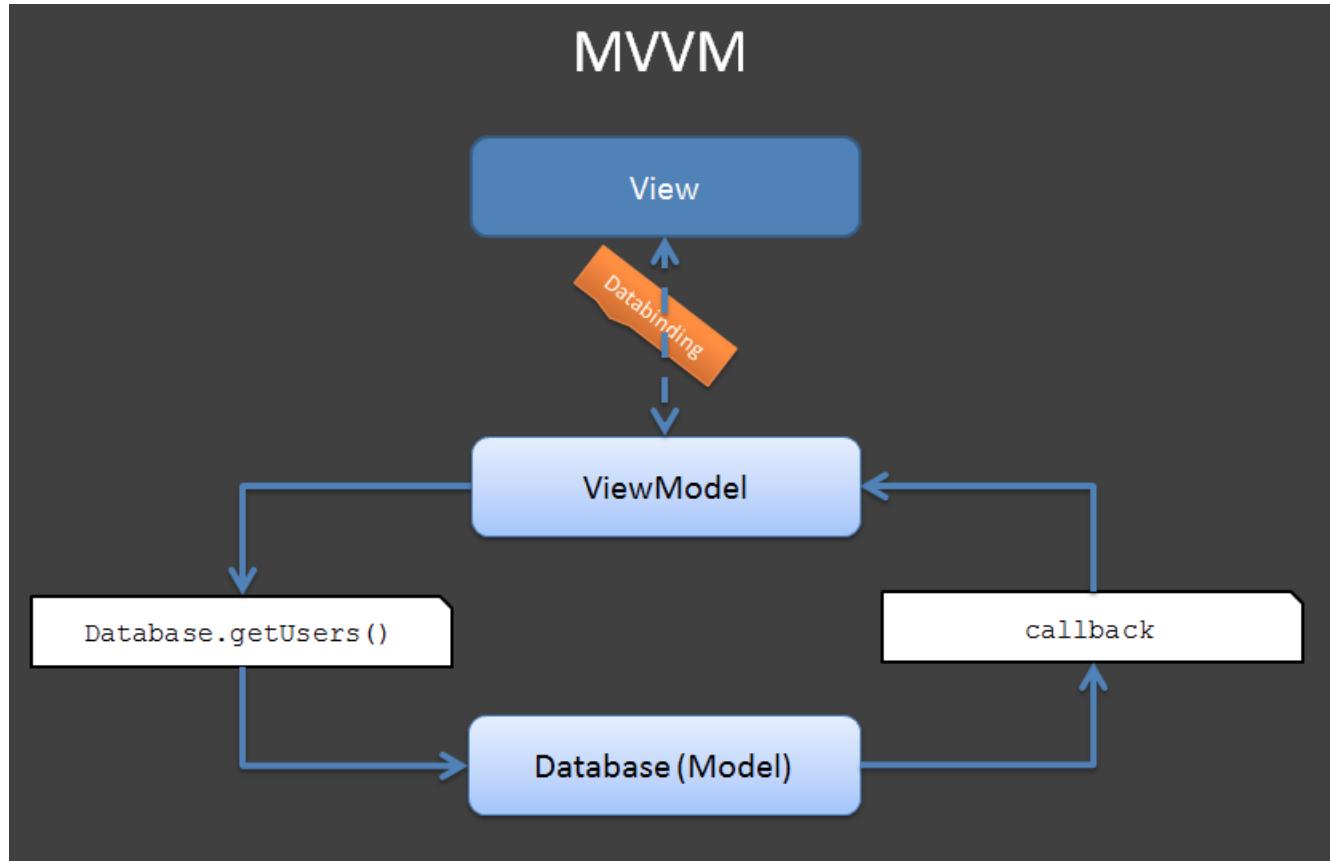
Architecture MVP (Model View Presenter)

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

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

Architecture MVVM (Model View ViewModel)

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

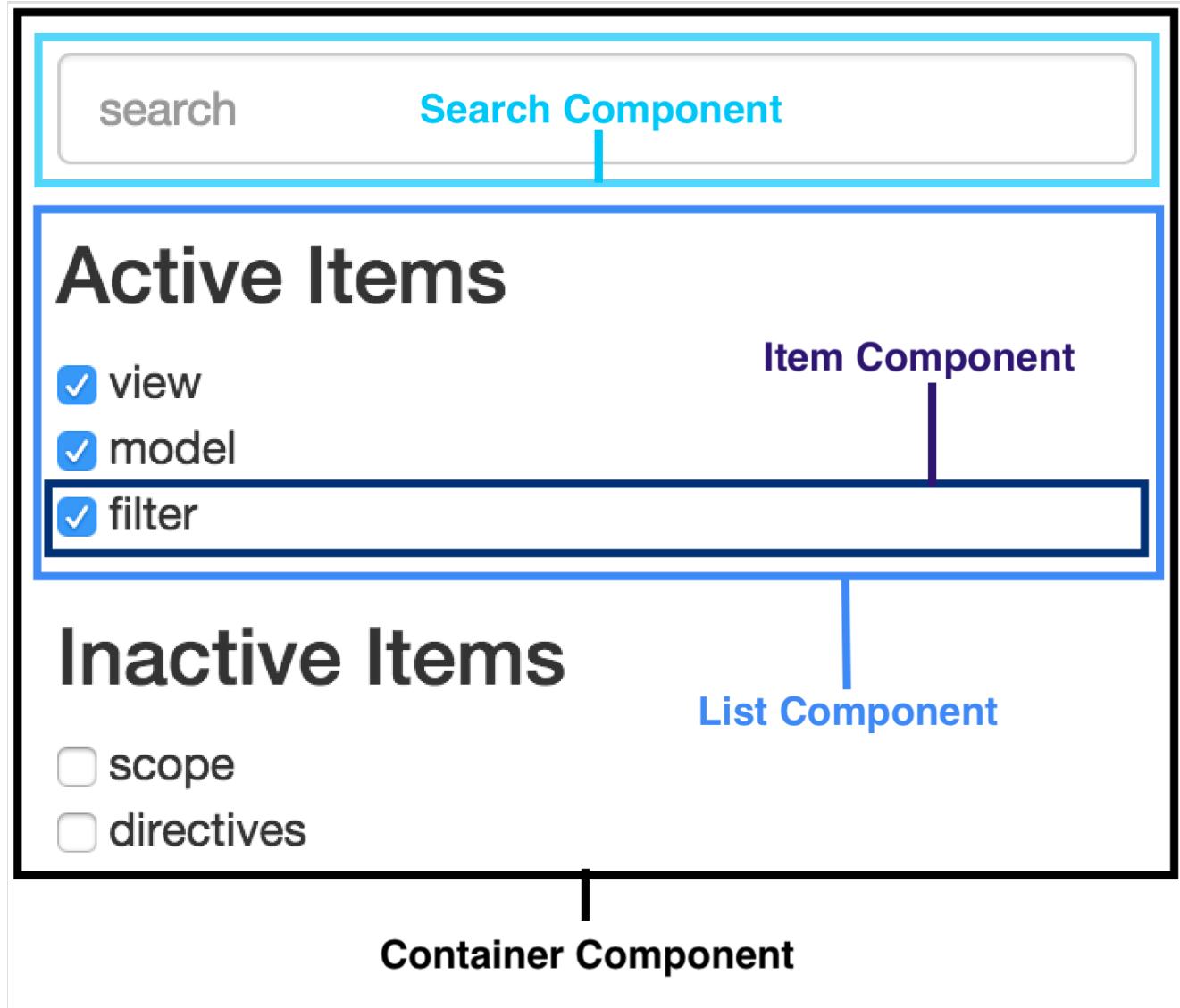


Cette méthode permet, tel le modèle MVC (modèle-vue-contrôleur), de séparer la vue de la logique et de l'accès aux données en **accentuant les principes de binding et d'événements**.

Component-based Architecture

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

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



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

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

A propos des Web Components



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

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

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

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

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

Une démonstration minimalist dans chrome

Cet exemple utilise la syntaxe ES6

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

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

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

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

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

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

Un modèle générique `HTMLElement`.

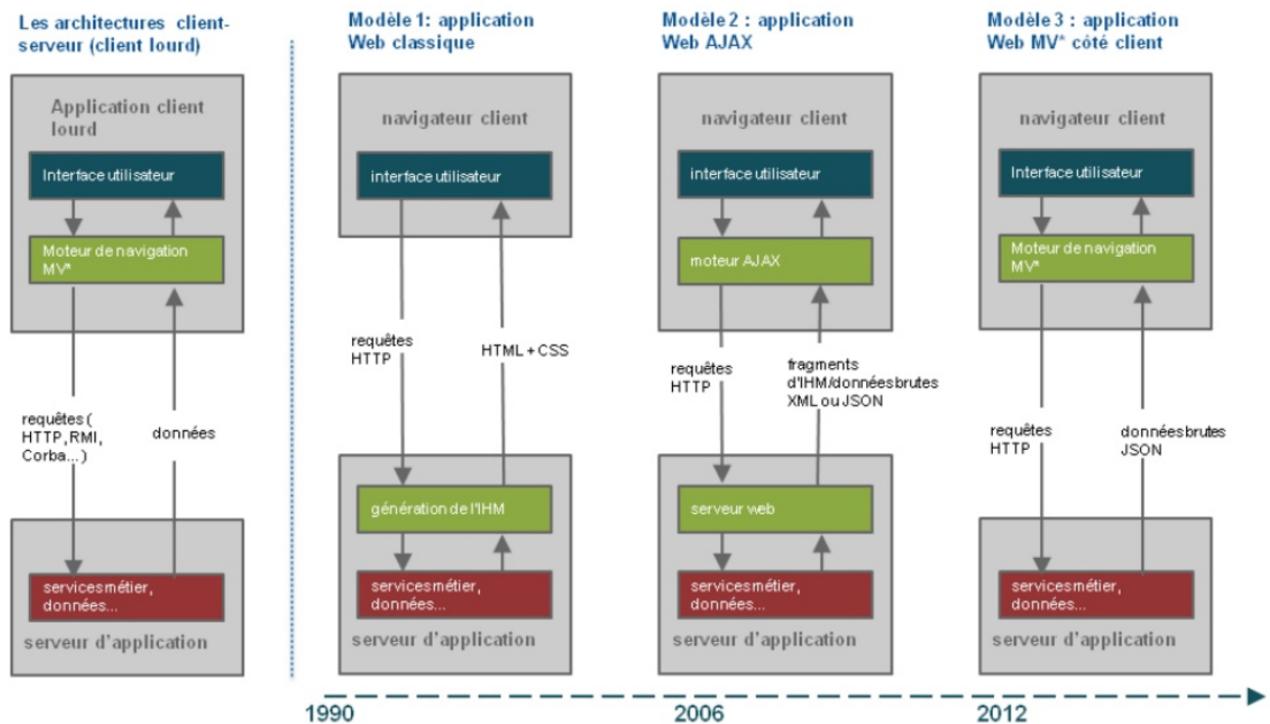
Un bloc logique `class`.

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

Un cycle de vie exposé par le système de rendu.

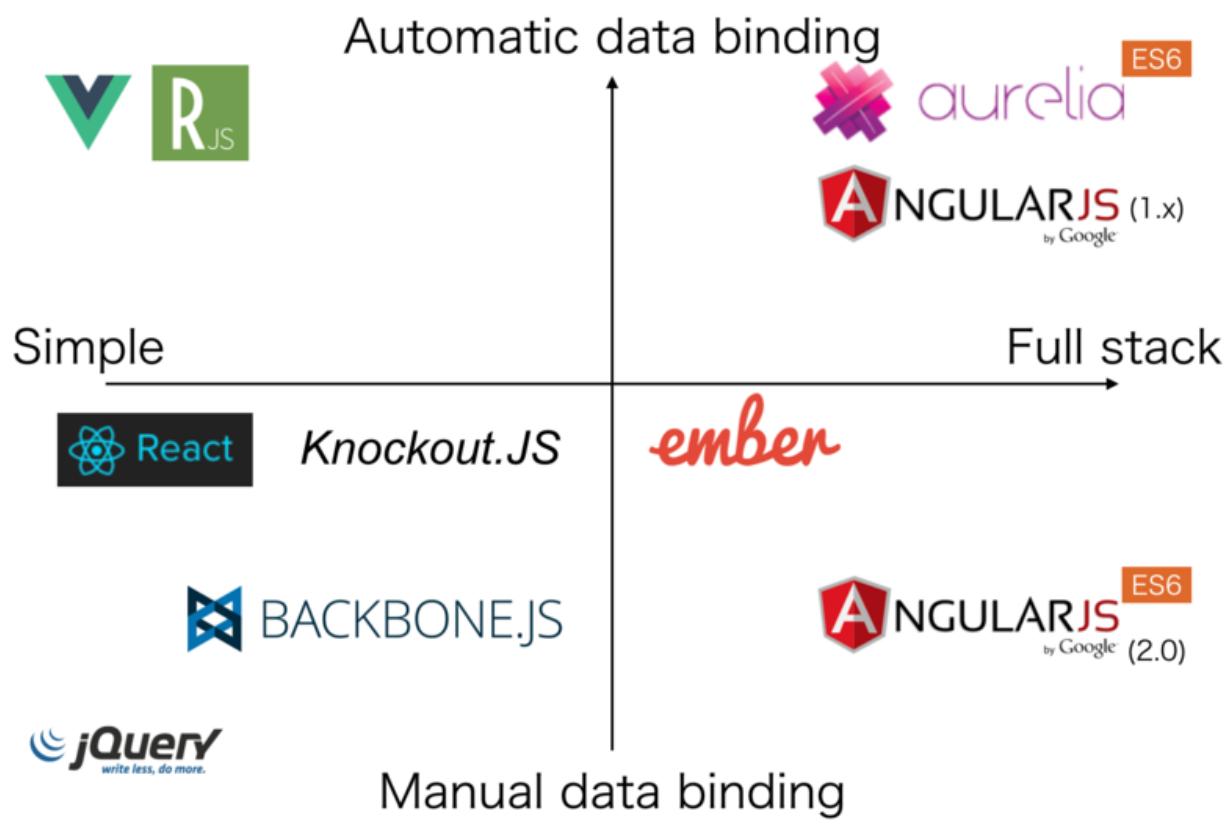
Ecosystème des frameworks JavaScript.

Evolution des développements Front End.

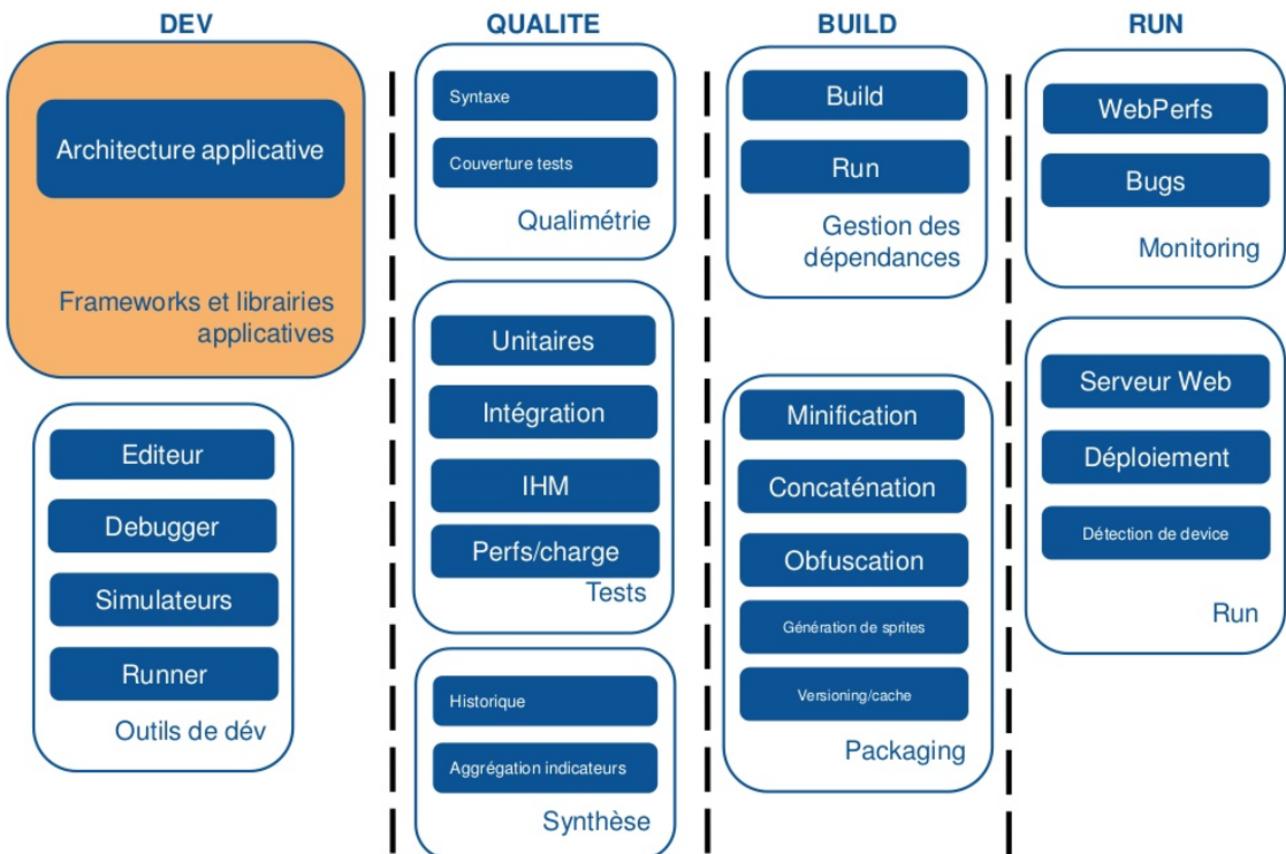


L'actuel "[marché](#)" des Frameworks UI/MVC JavaScript est très riche.

Chaque solution représente un choix d'implémentation particulier.

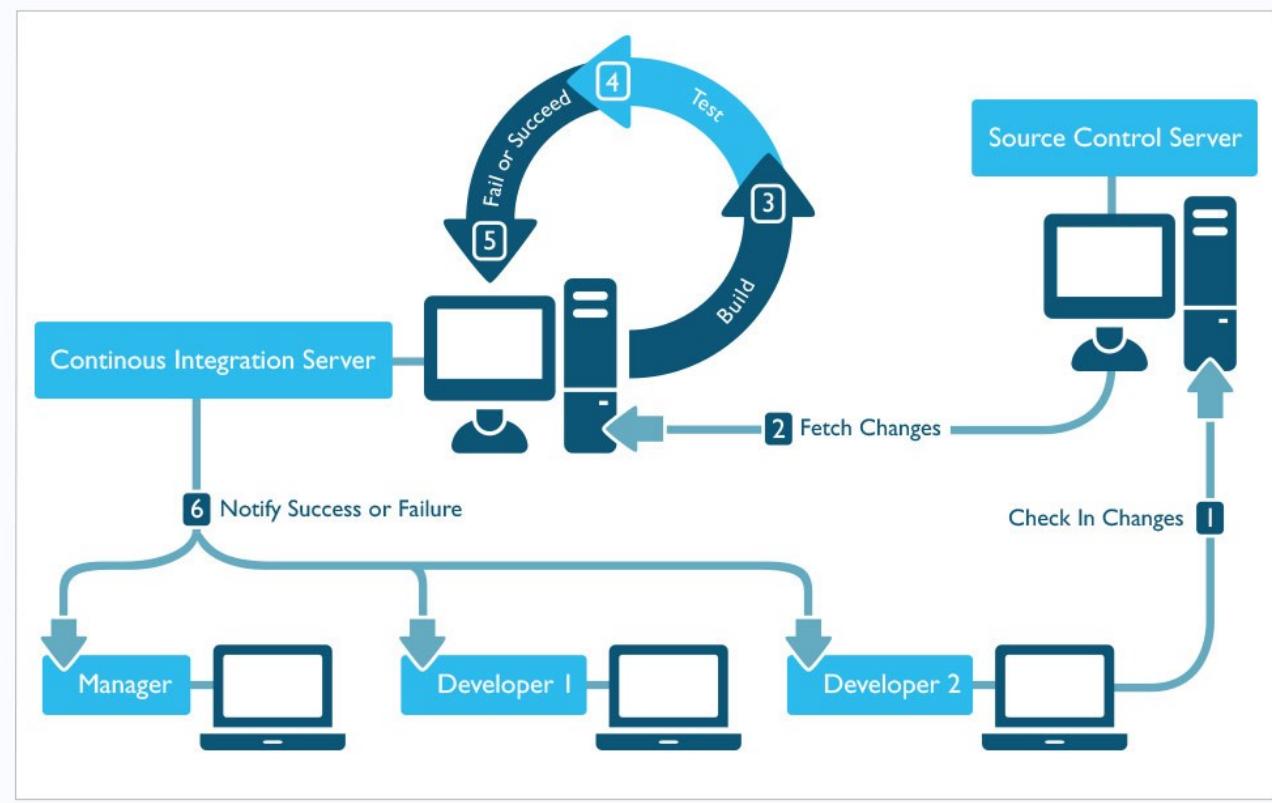


Un grand nombre d'utilitaires permettent d'automatiser les tâches découlant de l'évolution de ces pratiques de développement.



Industrialisation de la production.

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



Outils indispensables pour le développeur Front End.

Il est possible de classer les outils selon trois catégories :

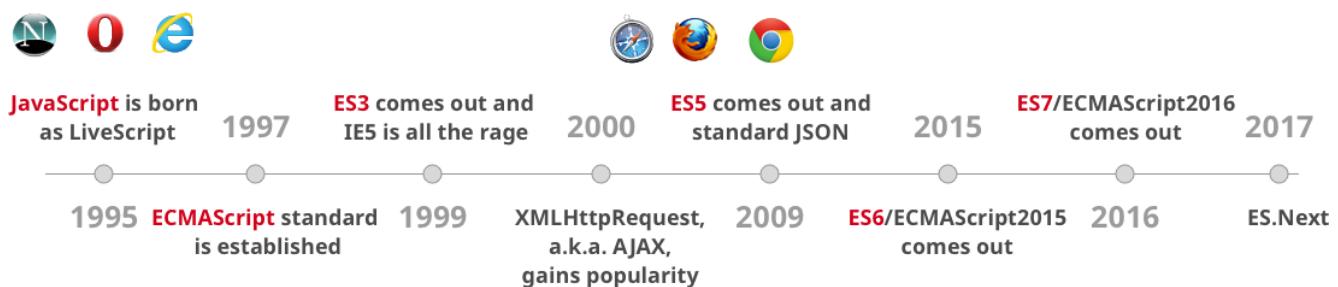
- **Logique** dépendances tiers ,telles que les différents frameworks.
- **Technique** ayant un impact sur la plate-forme de développement pour l'automatisation et le contrôle des tâches par exemple.
- **Productivité** toute autre solution non indispensable permettant d'accélérer le développement, allant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

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

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

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

Prendre en considération l'évolution du langage

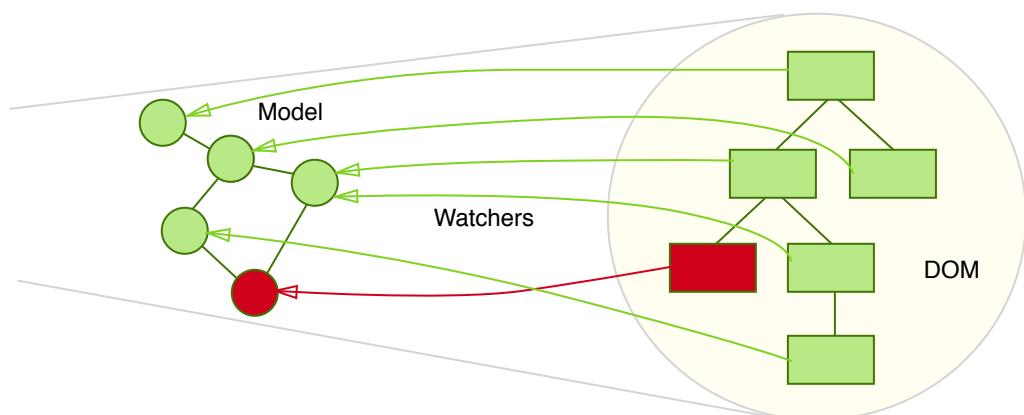


Par anticipation et les autres frameworks JavaScript se rapprochent de la future pattern de développement qu'apporteront les [Web Components](#)

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

Dirty-checking (vérifications massives)

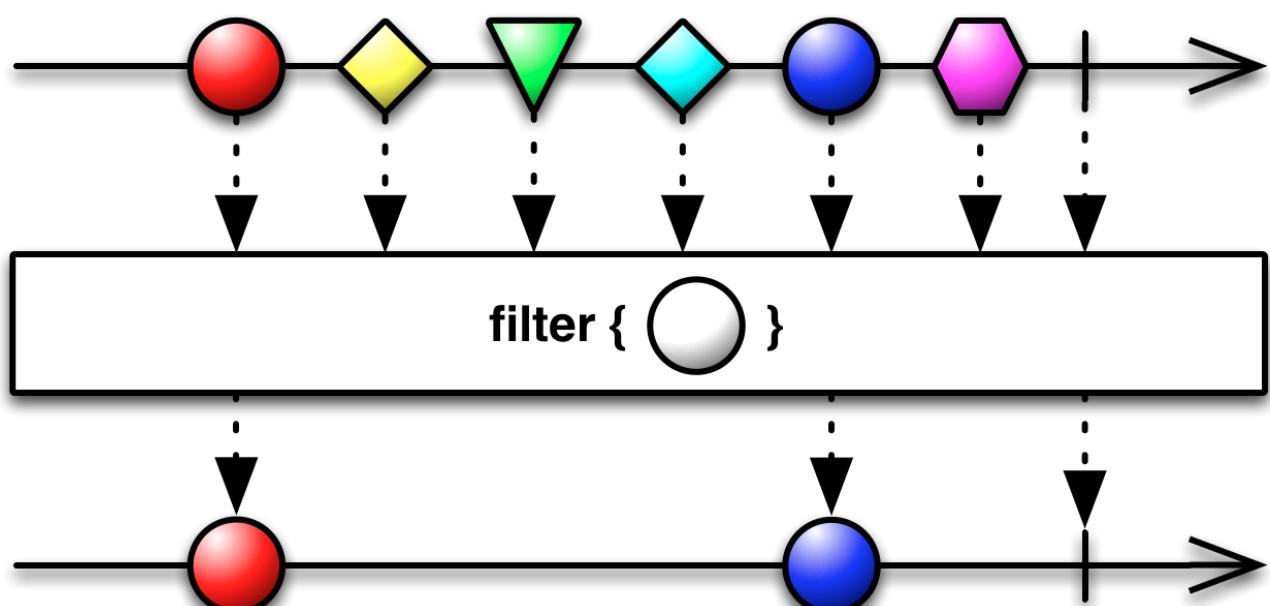
L'idée de base du dirty-checking est que dès qu'une modification des données peut avoir eu lieu, la bibliothèque va examiner l'intégralité du modèle de données pour déterminer s'il a changé, en utilisant un cycle de vie basé sur des sommes de contrôle ou sur les valeurs brutes.



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 Observé. On le mettra sous observation avec la méthode `subscribe` et cette observation sera exécutée par un *Observer*.



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

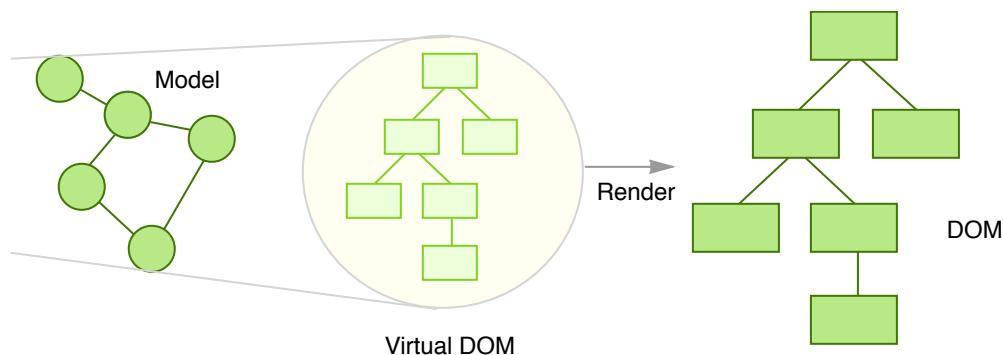
Virtual-Dom

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

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

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

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



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

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 JavaScript.

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 JavaScript !

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

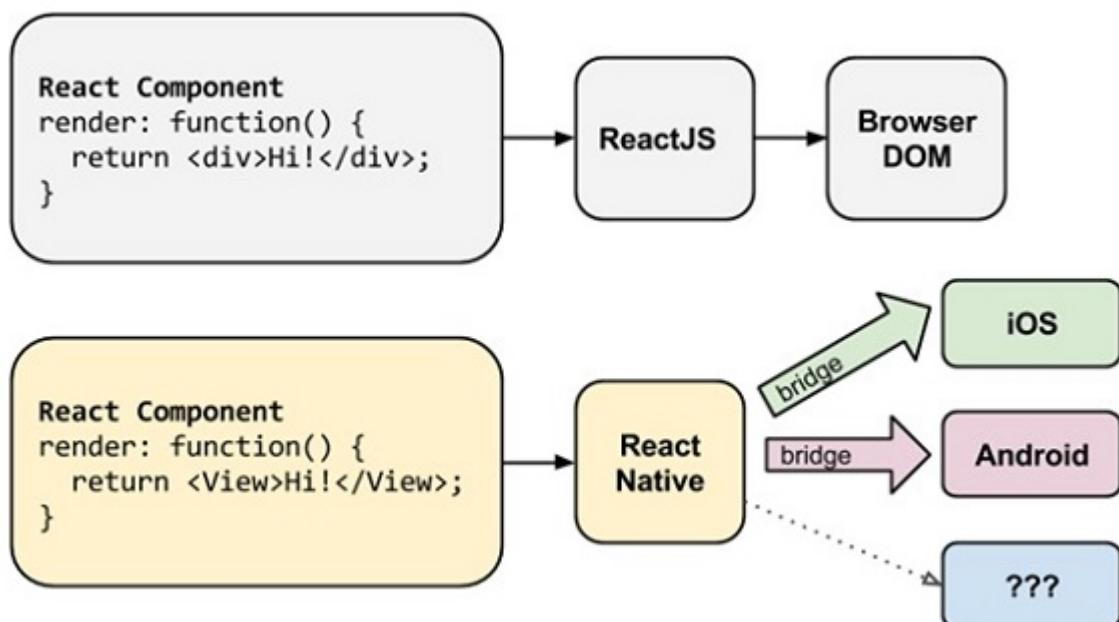
JSX, présentation. Mise en oeuvre “Transpilers”.

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.



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 ?**

Quelques questions simples pour confirmer le socle de compétences.

- **Portée & Closure:** Saviez-vous que JavaScript utilise portée lexicale est basé sur le compilateur (pas l'interpréteur!)

Pouvez-vous expliquer :

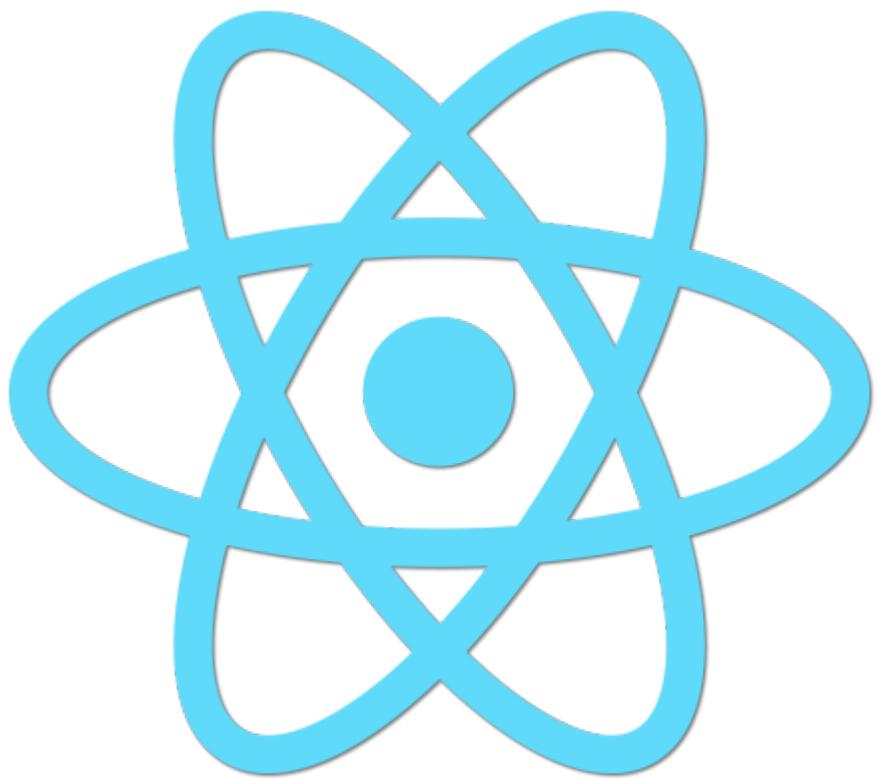
L'ordre de résolution des variables ?

Leur principe de transmission ?

Pourquoi les *closures* sont le résultat direct de la portée lexicale et fonctions en tant que valeurs?

- **this & Prototypes d'objets:** Pouvez-vous donner les quatre règles simples d'initialisation du mot clé `this` est lié?
- **Types & Syntaxe:** Connaissez-vous les types **natifs de JS** ? Comment vous sentez-vous avec les nuances syntaxiques 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”?



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.

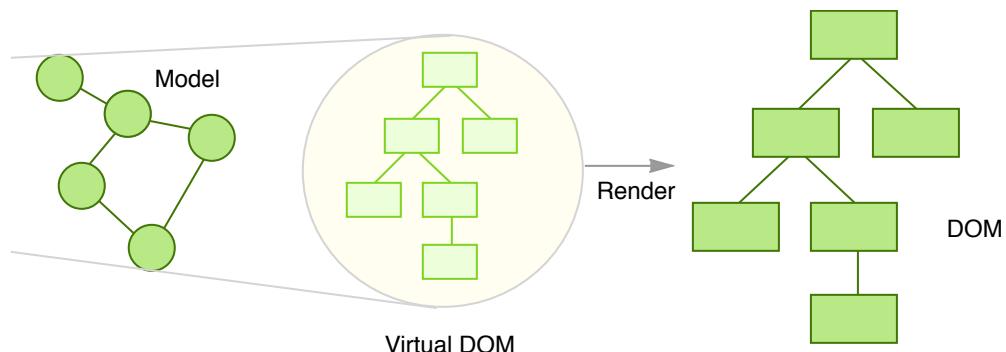
Il est plus facilement testable et vous permet d'avoir une bonne couverture de tests unitaires. **Il suffit de tester le comportement du composant en faisant abstraction de son contexte.**

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

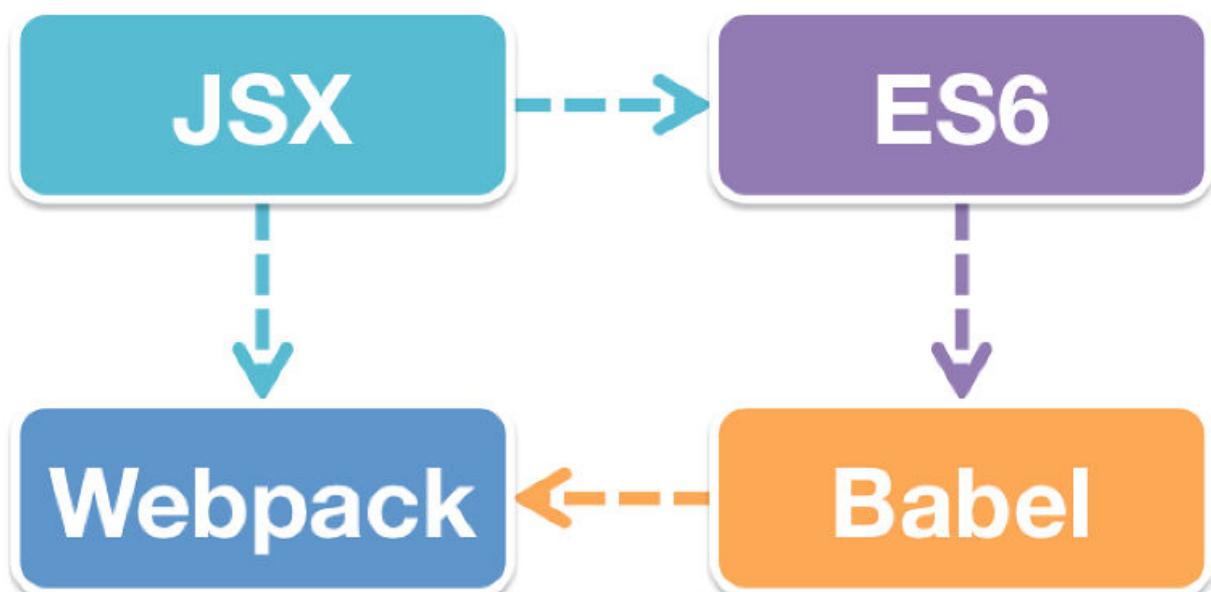
Approche : MVC et Virtual Dom, un choix de performance.

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

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



Utiliser JavaScript ou JSX.



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

Le code JSX:

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

Devient

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

Comprendre JSX en détail.

Déclaration de composants.

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

Les types en majuscules indiquent que la balise JSX fait référence à un composant

React. Ces balises sont compilées en une référence directe à la variable nommée, donc si vous utilisez l'expression JSX `<Foo />`, Foo doit être dans la portée.

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

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

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

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

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

```
import React from 'react';

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

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

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

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

```
import React from 'react';

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

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

Choix du type à l'exécution

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

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

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

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

Initialiser les propriétés de composants.

Il existe plusieurs façons de spécifier les propriétés en JSX.

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

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

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

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

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

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

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

Composants descendants (enfants).

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

Il existe plusieurs façons de transmettre des enfants:

- String Literals
- JavaScript Expressions
- JSX Children
- function

Booleans, Null, and Undefined seront ignorés.

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

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

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

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

Template conditionnel basé sur les valeurs ignorées

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

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

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

Rendu des éléments.

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

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

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

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

Note:

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

Rendu d'un élément dans le DOM

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

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

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

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

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

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

[Essayez sur CodePen.](#)

Mise à jour de l'élément rendu

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

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

Considérez cet exemple d'horloge tic-tac:

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

setInterval(tick, 1000);
```

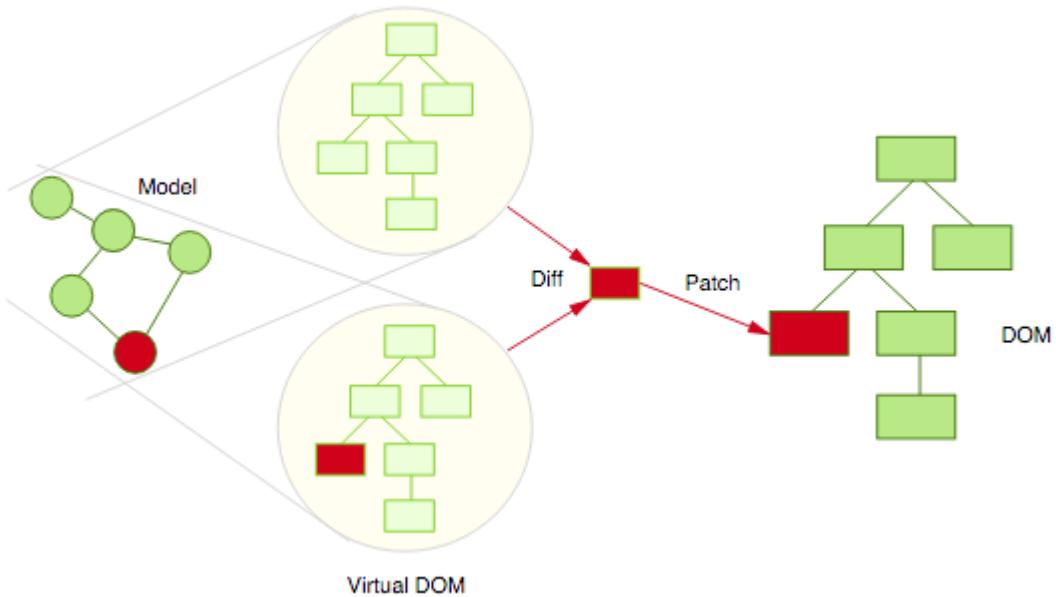
[Essayez sur CodePen.](#)

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

Note:

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

React ne met à jour qui est nécessaire



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

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

Hello, world!

It is 12:26:46 PM.

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

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

Méthodes principales de l'API.

L'API React est simple et concise.

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

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

Composants

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

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

Composants fonctionnels et de classe

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

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

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

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

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

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

Rendu d'un composant

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

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

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

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

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

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

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

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

[Essayez sur CodePen.](#)

Reprendons ce qui se passe dans cet exemple:

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

Attention:

Prefixez toujours les noms des composants avec une majuscule.

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

Composants

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

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

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

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

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

[Essayez sur CodePen.](#)

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

Attention:

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

Extraction de composants

Diviser les composants en composants plus petits.

Par exemple, considérez ce composant `Comment` :

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

[Essayez sur CodePen.](#)

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

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

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

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

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

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

Nous pouvons maintenant simplifier `Comment` un petit peu:

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

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

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

Cela nous permet de simplifier encore `Comment`:

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

[Essayez sur CodePen.](#)

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

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

Les props sont en lecture seule.

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

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

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

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

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

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

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

Création de composant de vues. Cycle de vie.

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

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

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

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

setInterval(tick, 1000);
```

[Essayez sur CodePen.](#)

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

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

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

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

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

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

Conversion d'une fonction en classe

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

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

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

[Essayez sur CodePen.](#)

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

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

Ajout d'un état local à une classe

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

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

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

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

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

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

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

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

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

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

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

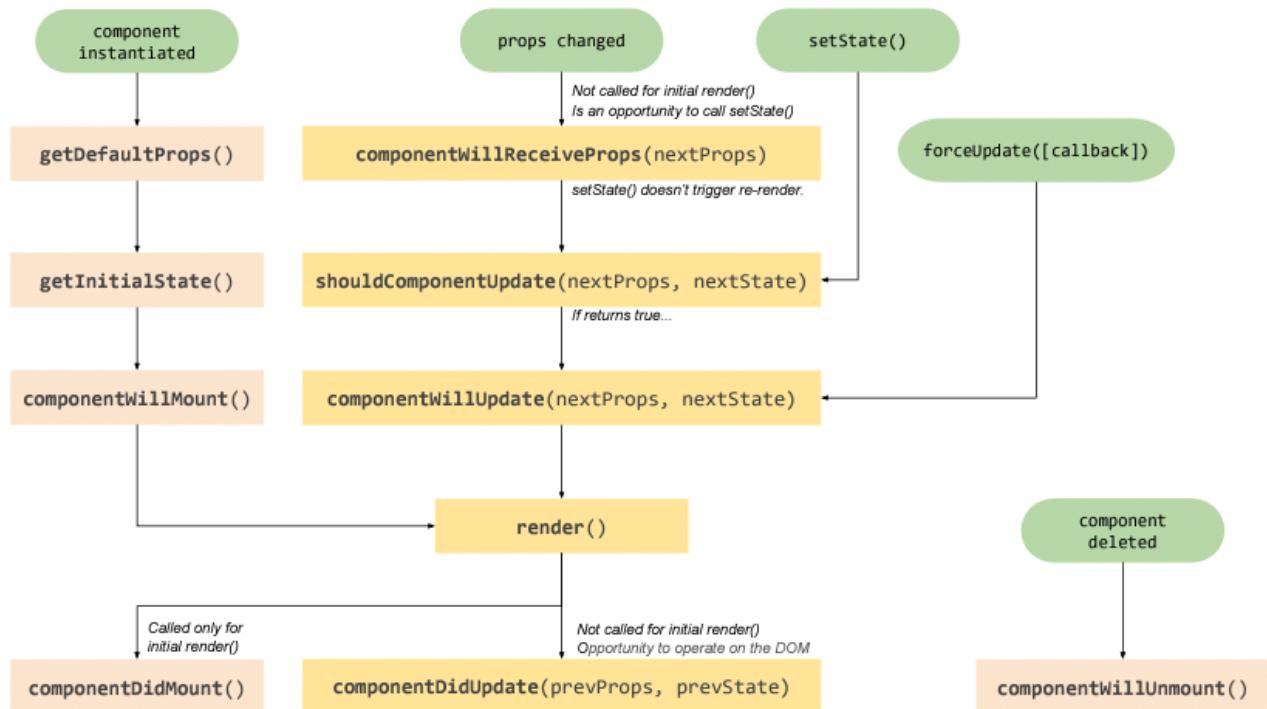
Nous ajouterons ensuite le code DU timer au composant lui-même.

Le résultat est le suivant:

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

[Essayez sur CodePen.](#)

Ajout de méthodes de cycle de vie à une classe



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

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

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

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

```

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

  componentDidMount() {

  }

  componentWillUnmount() {

  }

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

```

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

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

```

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

[Essayez sur CodePen.](#)

Récapitulif:

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

Utilisation correcte de l'état

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

Ne pas modifier l'état directement

Par exemple, cela ne rendra pas un composant:

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

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

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

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

Les mises à jour d'état peuvent être asynchrones.

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

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

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

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

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

ou

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

Mises à jour d'état sont fusionnées

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

Par exemple, votre état peut contenir plusieurs variables indépendantes:

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

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

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

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

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

Flux de données descendants.

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

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

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

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

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

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

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

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

[Essayez sur CodePen.](#)

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

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

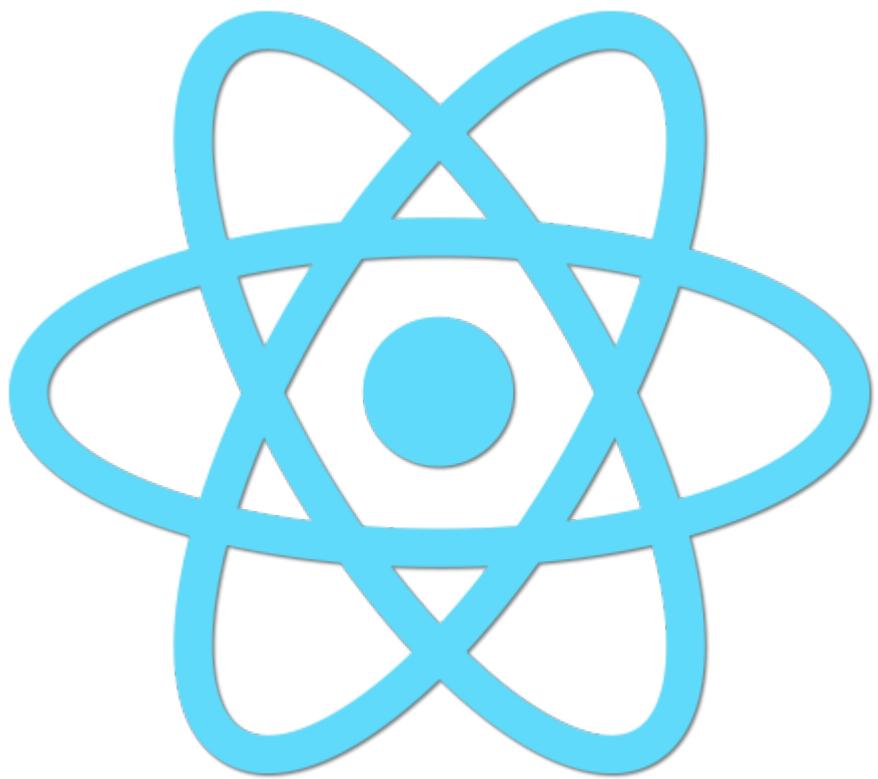
Pour montrer que tous les composants sont vraiment isolés, nous pouvons créer un composant `App` qui rend trois `<Clock>`s:

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

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

[Essayez sur CodePen.](#)

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



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 la méthode `handleCelsiusChange` de `Calculator` , et `onChange` de Fahrenheit `TemperatureInput` est la méthode `handleFahrenheitChange` .
- 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 appelle 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.

Enter temperature in Celsius:

Enter temperature in Fahrenheit:

The water would not boil.

The screenshot shows the React Developer Tools interface. At the top, there are tabs: Elements, Console, Sources, Network, Timeline, Profiles, Application, and Security. Below the tabs, there are checkboxes for Trace React Updates, Highlight Search, and Use Regular Expressions. On the right side, there is a sidebar with the title <Calculator>. The sidebar shows the component tree under <Calculator>: <Calculator> -> <div> -> <TemperatureInput scale="c">, <TemperatureInput scale="f">, <BoilingVerdict celsius=null>. To the right of the tree, there are sections for Props (empty object) and State (scale: "c", value: "").

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 $", approvisionné: true, le nom: "Footb  
  {Category: "Articles de sport", prix: "$ 9.99", approvisionné: true, le nom: "Baseba  
  {Category: "Articles de sport", prix: "29,99 $", approvisionné: false, le nom: "Basi  
  {Catégorie: "Electronique", prix: "99,99 $", approvisionné: true, le nom: "iPod Tou  
  {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”.

<input type="text" value="Search..."/>	
<input type="checkbox"/> Only show products in stock	
Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

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 définiront 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 vos applications.

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 la propriété `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 :

```
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 identifiants 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 géré 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 (soumission).

Composants contrôlés

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

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

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

```

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

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

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

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

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

```

[Essayez-le sur CodePen.](#)

La balise textarea

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

```

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

```

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

```

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

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

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

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

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}

```

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

La balise select

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

```

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

```

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

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

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

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

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

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

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

[Essayez-le sur CodePen.](#)

Manipulation Entrées multiples

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

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

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

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

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

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

[Essayez-le sur CodePen.](#)

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

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

Il est équivalent à ce code ES5:

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

Alternatives aux composants contrôlés

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

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

```
class NameForm extends React.Component {  
  constructor(props) {  
    super(props);  
    this.handleSubmit = this.handleSubmit.bind(this);  
  }  
  
  handleSubmit(event) {  
    alert('A name was submitted: ' + this.input.value);  
    event.preventDefault();  
  }  
  
  render() {  
    return (  
      <form onSubmit={this.handleSubmit}>  
        <label>  
          Name:  
          <input type="text"  
            defaultValue="Bob" //specifies un default value  
            ref={(input) => this.input = input} />  
        </label>  
        <input type="submit" value="Submit" />  
      </form>  
    );  
  }  
}
```

Manipulation du DOM.

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

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

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

Différences dans les attributs

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

Checked

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

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

className

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

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

dangerouslySetInnerHTML

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

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

```

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

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

```

htmlFor

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

style

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

```

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

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

```

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

```

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

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

```

suppressContentEditableWarning

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

value

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

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

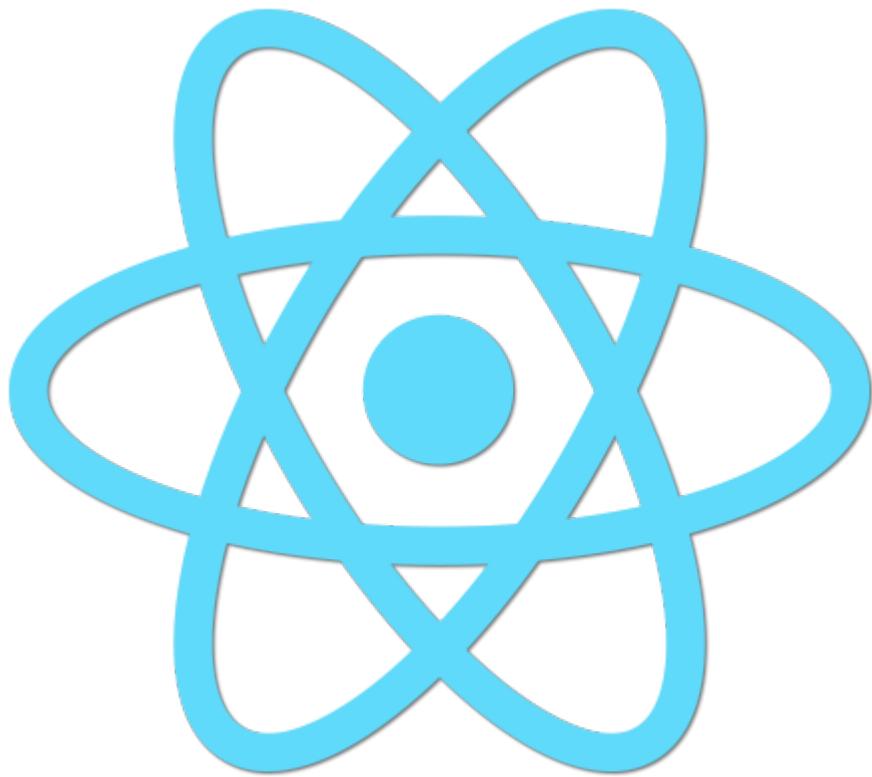
Tous les attributs HTML pris en charge

React supporte les attributs `data-*` et `aria-*`, [ainsi que ces attributs](#)

En outre, les attributs non standard suivants sont supportés:

- `autoCapitalize` `autoCorrect` Mobile Safari.
- `color` pour `<link rel="mask-icon"/>` dans Safari.
- `itemProp` `itemScope` `itemType` `itemRef` `itemID` pour les [micro data](#).
- `security` pour les anciennes versions d'Internet Explorer.
- `unselectable` pour Internet Explorer.
- `results` `autoSave` pour les champs de type `search` de WebKit/Blink.

Tous les attributs SVG pris en charge



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.

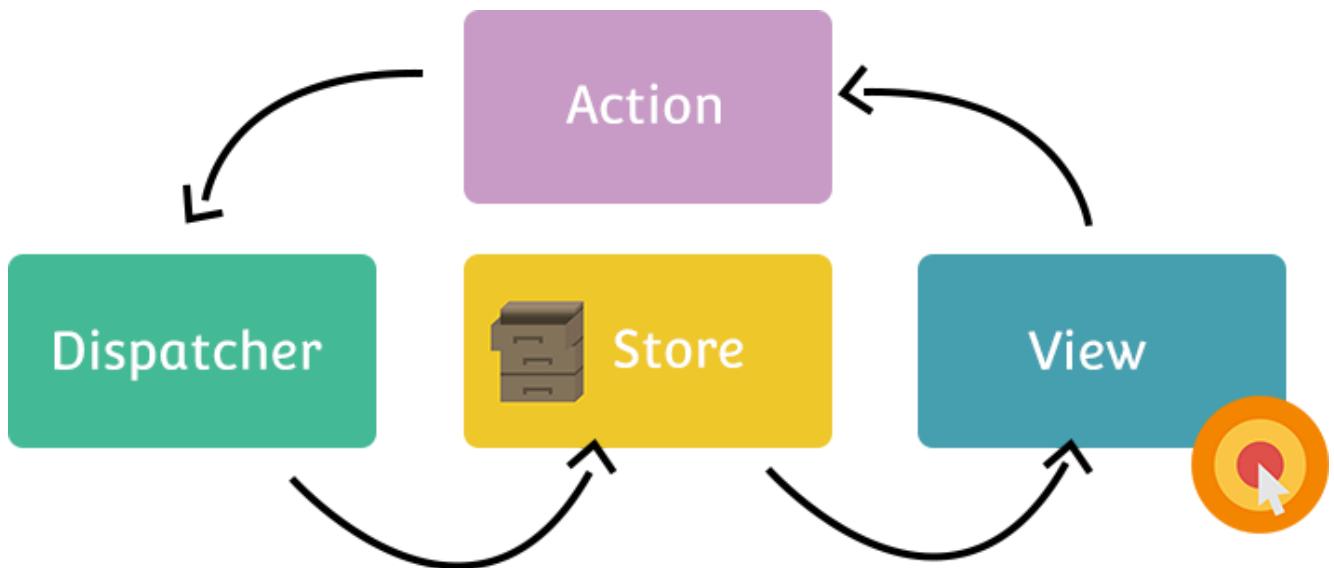
Pour surmonter cette difficulté, Facebook a créé une « nouvelle » architecture appelée **Flux**. Contrairement au MVC, celle-ci est une architecture garantissant (en théorie) un **flux unidirectionnel**, permettant à un développeur d'identifier rapidement le chemin critique d'un événement et ses conséquences.

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 pré-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.

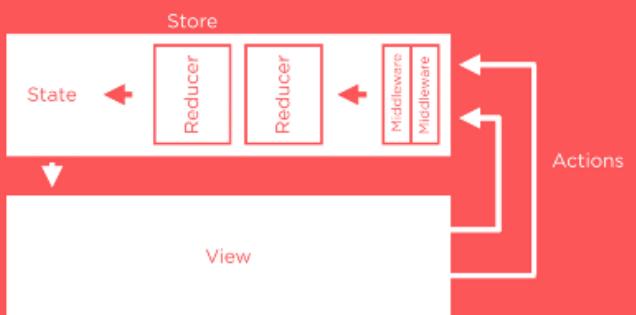
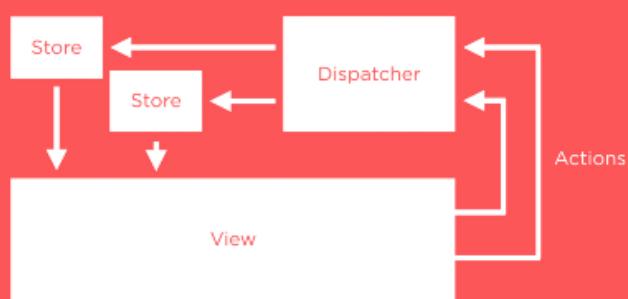
Comparaison des architectures.

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

Redux



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 transmises.

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 “Dispatcher” dans Flux pour les actions.

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

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

Les “Stores”, gestionnaire d’états logique dans Flux.

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

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

Un `store` se comportent comme un modèle observable (il dispose de `getters` et d'une méthode `addListener`) à la différence près qu'il n'a pas de `setter`.

Seul le `store` peut mettre à jour ses propres données.

```

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

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

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

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

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

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

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

//c'est ici que la magie s'opère. On enregistre un callback qui sera appelé dès que qu
appDispatcher.register(function (payload) {
    //dans le payload on a tous les détails de l'évènement (type et données qui va ave
    var action = payload.actionType,
        data = payload.data;

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

        //quand un todo est créé sur le serveur on l'ajoute dans notre
        //liste de todos
        case events.TODO_ADDED:
            todos.push(data.todo); //on ajoute la nouvelle todo dans la liste
    }
});

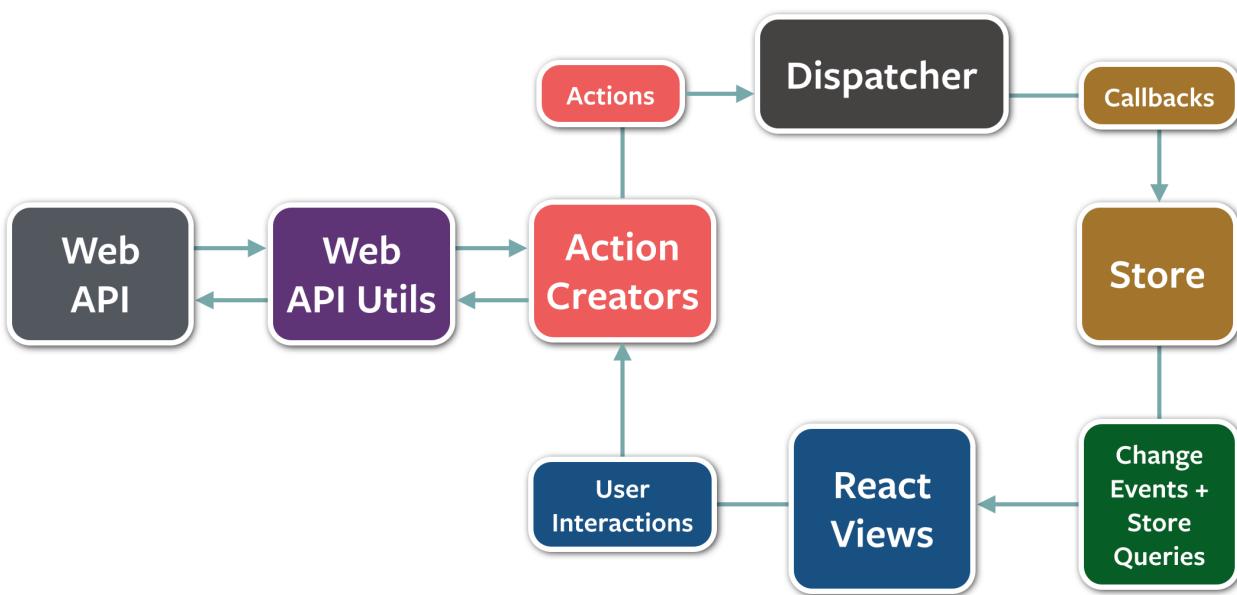
```

```
this.emitChange(); //stay tuned, on parle de ça bientôt
break;

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

module.exports = todosStore;
```

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 Functional Programming.

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

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

Approche avec Redux. Le “Reducer”.

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

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

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

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

Utilitaire Redux

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

Redux en exemples

```

import { createStore } from 'redux'

/**
 * Le reducer est une fonction dite "pure" ayant (state, action) => state comme signa
 * Il va décrire comment une action transforme le state (l'état) de l'application
 * en un nouvel état.
 *
 * L'implémentation de l'état de l'application dépend totalement de votre y * use case
 * une structure de données
 * immutable (basé sur Immutable.js par exemple).
 * La seule chose à retenir est que cette partie ne DOIT PAS modifier
 * l'objet correspondant à l'état de l'application lorsque l'état change.
 * Dans cet exemple, on utilise un switch et des strings, mais on pourra * très bien
 */
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

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

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

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

```

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

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

DIFFÉRENCES AVEC FLUX ?

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

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

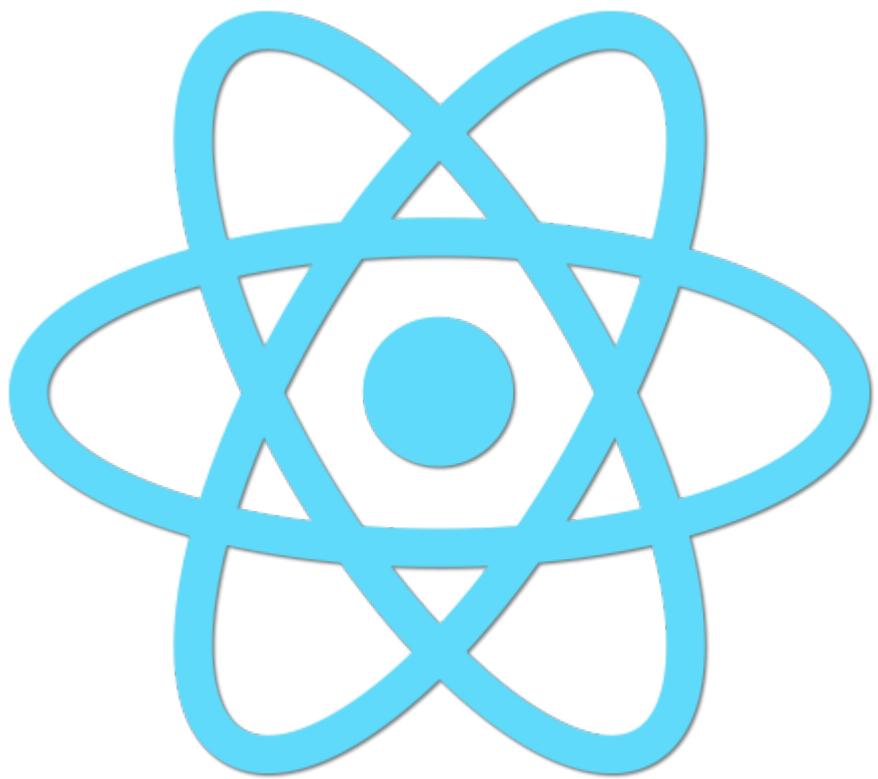
Extension pour ReactJS : “hot-loader”.

[React Hot Loader](#) permet la modification à chaud d'un composant React.

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

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

[React Hot Loader](#)



Application isomorphe

Application isomorphe

La performance

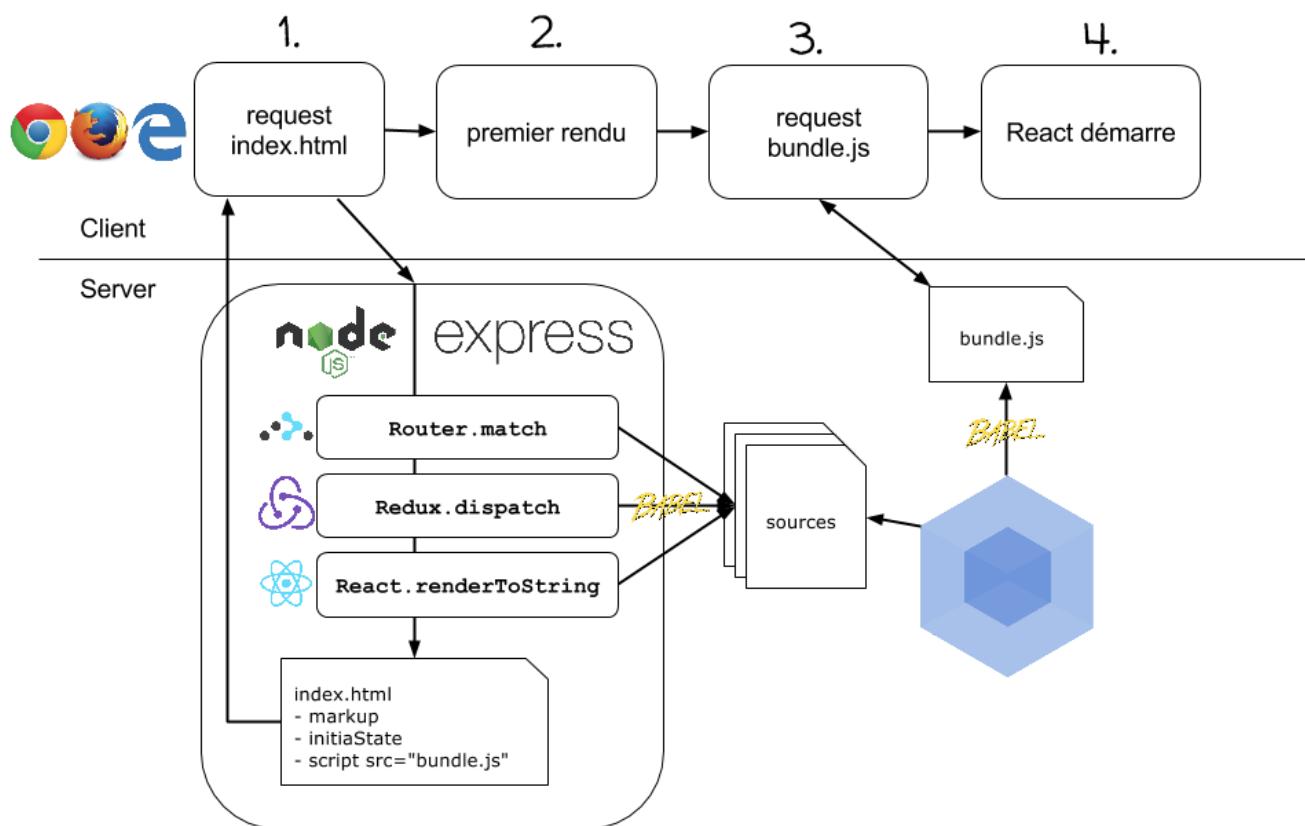
Cycle de d'initialisation d'une SPA coté client :

1. Chargement du fichier HTML

Chargement des différents Assets (Css, image, scripts JS application et librairies/frameworks)

Délai : Parsing / Execution.

* Délai : Démarrage et configuration de l'état.



Principe et bénéfices du développement isomorphe.

Le mot *isomorphisme* vient des racines grecques *isos* pour égal et *morph* pour forme.

L'isomorphisme désigne donc **deux entités de même forme dans un contexte différent**.

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

Concrètement, les avantages de l'isomorphisme sont multiples:

- Un seul code, pour toute une application (serveur et client)
- Les moteurs de recherche voient le contenu (plus totalement vrai: Google interprète le JavaSscript)
- Rapide, plus nécessaire d'attendre le téléchargement du JS pour voir la page.
- Plus facile à maintenir (une seule codebase)
- Un état identique (partagé) entre client et serveur = un debug plus simple.

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

React et l'isomorphisme

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

Il est possible d'utiliser 3 notations bien distinctes.

```

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

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

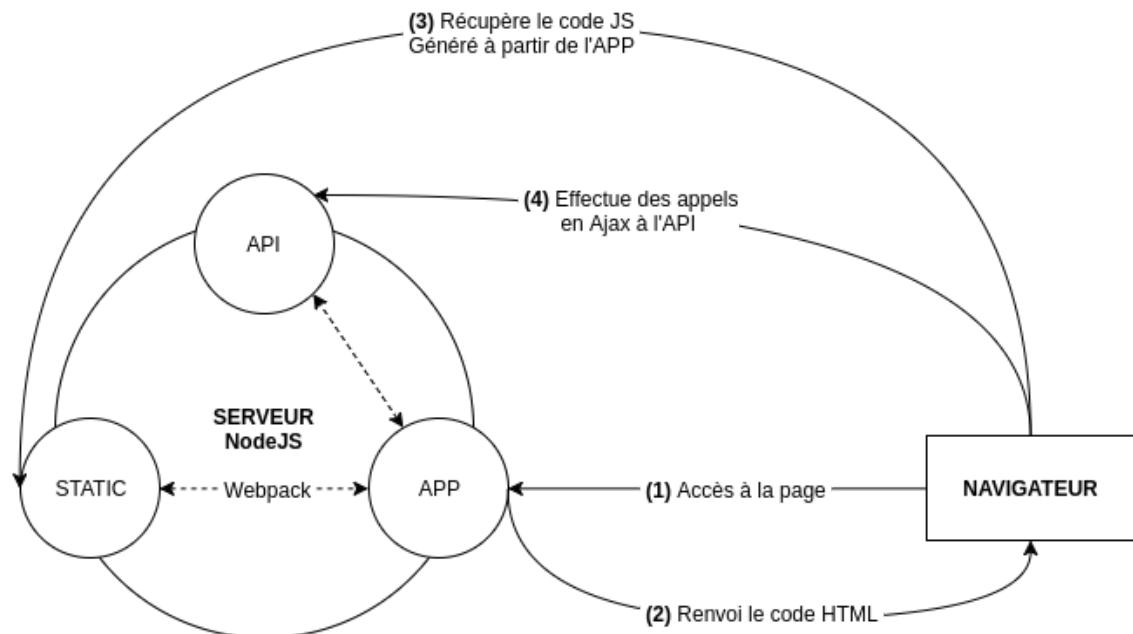
// Création de l'élément en "brut" (objet JS)
class Button extends React.Component {
    render() {
        return {
            type: 'button',
            props: {
                className: 'button',
                children: {
                    type: 'b',
                    props: {
         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

```

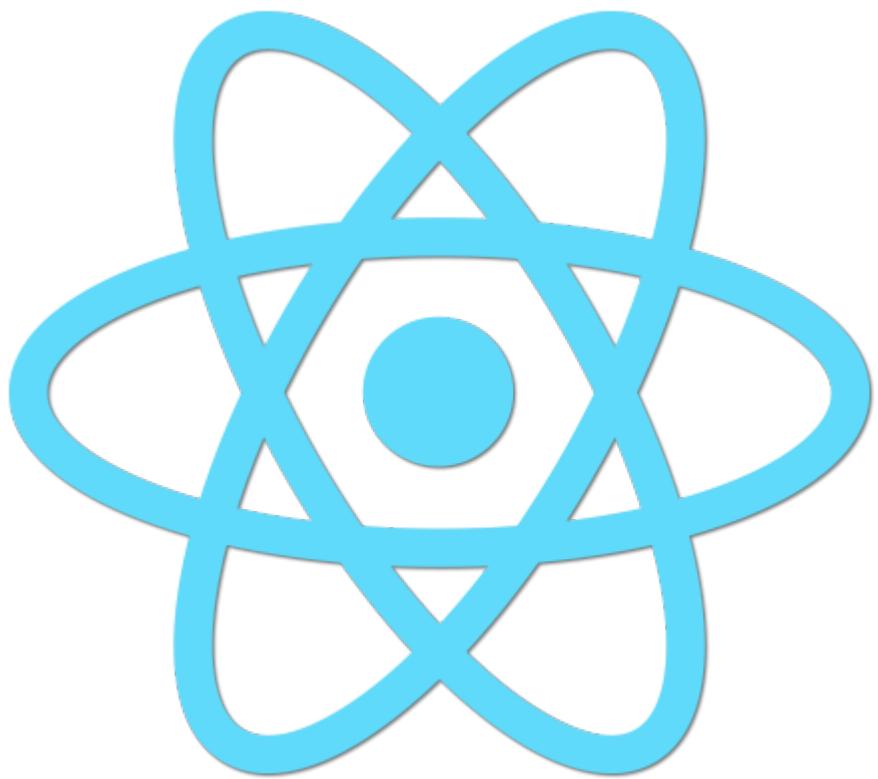
Ecosystème du JavaScript côté serveur.

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

Workflow d'application isomorphe.



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 soutien 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 personnalisés) : `<View>`, `<Text>`, `<Image>`, ...)

[Voir la liste complète dans la documentation](#)

Tout doit être packagé

Toutes les ressources doivent être appelées, 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 composants 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

