

Vue.js, maîtriser le framework JavaScript Open Source



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.**

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.

Exemple

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

(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 :

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.

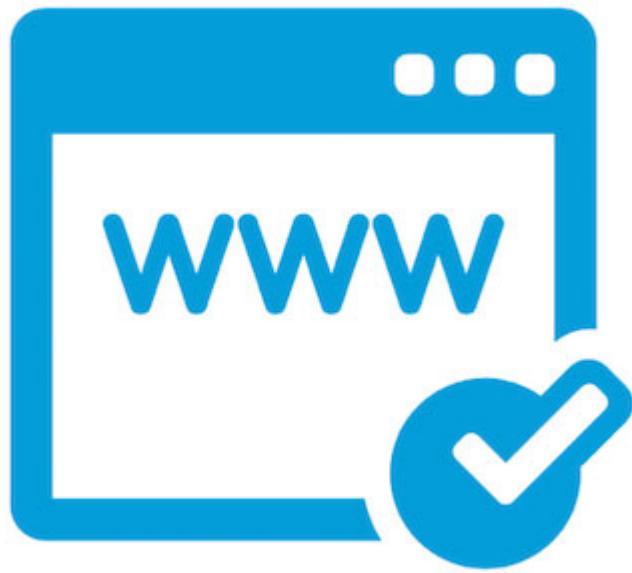
Durant ces deux premières phases il est recommandé de :

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***





Ecosystème moderne pour le développement
JavaScript.

Architecture des applications Web: composants, modèles et types

Internet évolue progressivement vers un engagement actif des utilisateurs et des fonctionnalités étendues offertes par les applications Web. De plus en plus de développeurs cherchent à développer des applications Web et à attirer davantage de visiteurs vers leurs ressources Web.

L'un des défis que tout développeur peut rencontrer avant de se lancer dans le développement d'applications Web consiste à choisir le type et le modèle de composant de l'architecture Web.



Types d'architecture d'application Web.

Les applications Web comprennent deux ensembles différents de programmes qui s'exécutent séparément mais simultanément, l'objectif commun étant de travailler en harmonie pour fournir des solutions.

En règle générale, les deux ensembles de programmes incluent le code dans le navigateur qui fonctionne selon les entrées de l'utilisateur et le code dans le serveur qui fonctionne selon les demandes de protocoles, HTTPS.

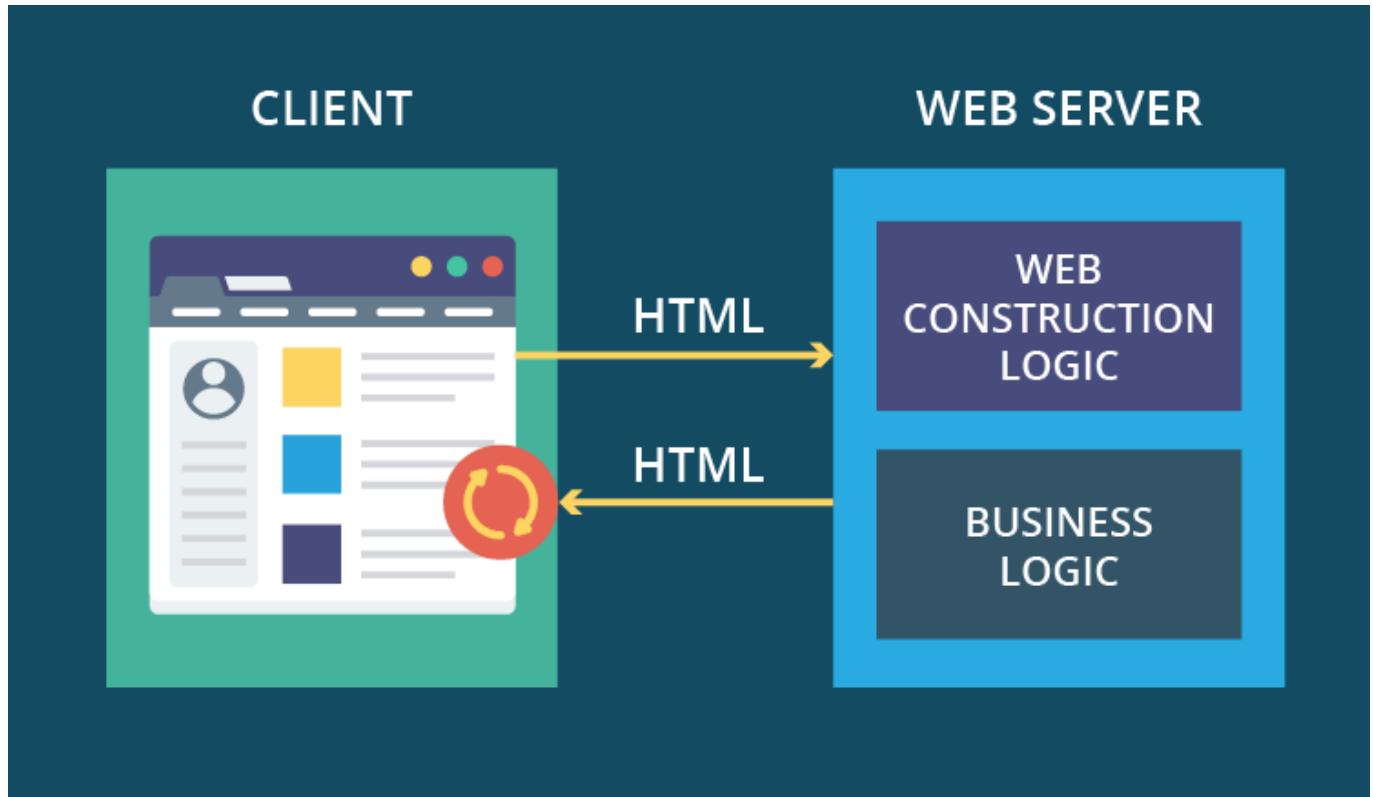
Quel que soit le modèle, tous les composants de l'application Web fonctionnent toujours simultanément et créent une application Web intégrale.

En fonction de la répartition de la logique de l'application entre le client et le serveur, il peut exister différents types d'architecture d'application Web.

Application Web HTML « Legacy »

Selon la toute première architecture de base d'applications Web, un serveur constitué d'une logique de construction de page Web et d'une logique métier interagit avec un client en envoyant une page HTML complète.

Pour voir une mise à jour, l'utilisateur doit entièrement recharger la page ou, en d'autres termes, demander au client d'envoyer une demande de page HTML au serveur et charger à nouveau son code complet.



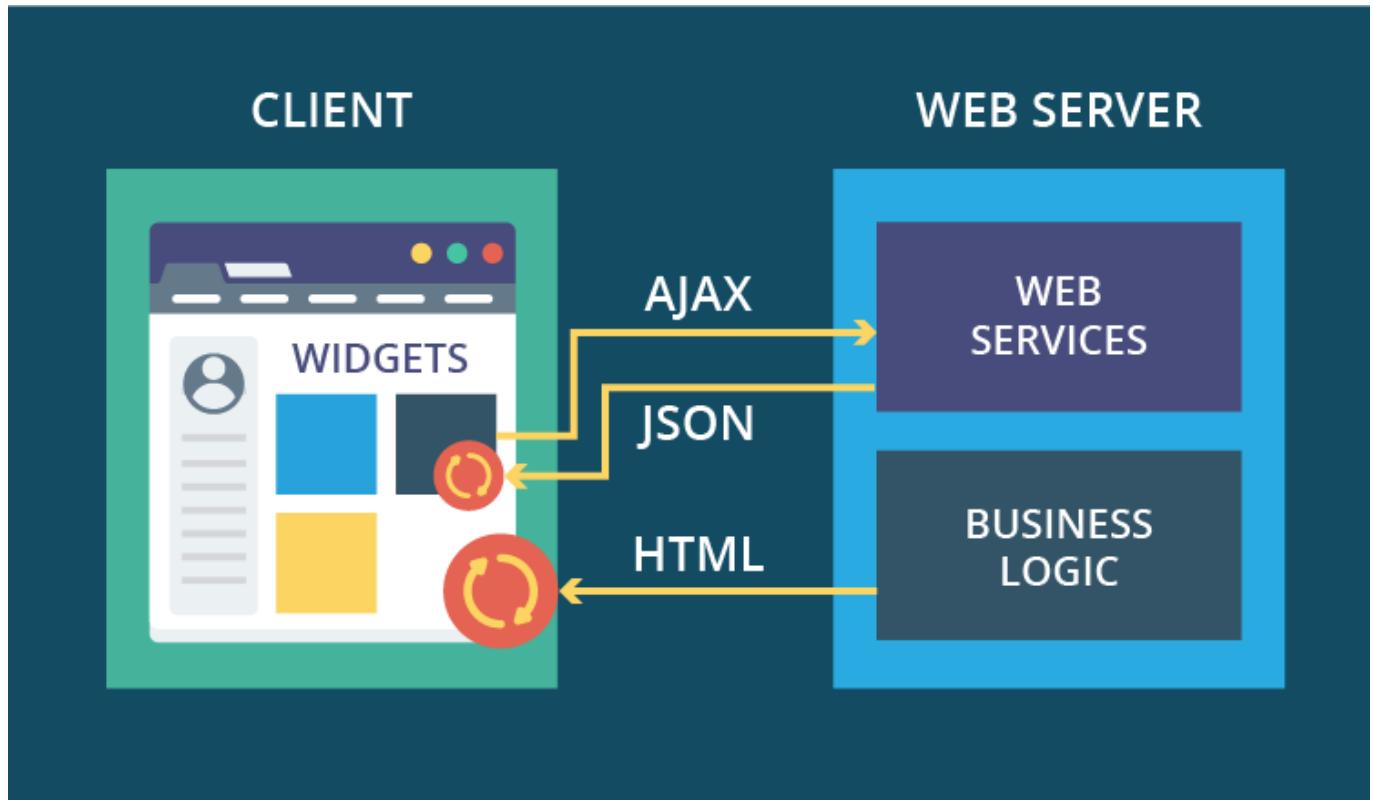
Étant donné que toutes les logiques et les données sont stockées sur le serveur et que l'utilisateur n'y a aucun accès, ce type d'architecture est hautement sécurisé.

Néanmoins, en raison du rechargement constant du contenu et de l'échange de données considérable, il est plus courant pour les sites Web statiques que pour les applications Web réelles.

Widget

Dans ce type, la logique de construction de page Web est remplacée par des services Web et chaque page du client possède des entités distinctes appelées widgets.

En envoyant des requêtes AJAX aux services Web, les widgets peuvent recevoir des fragments de données au format HTML ou JSON et les afficher sans recharger la page entière.



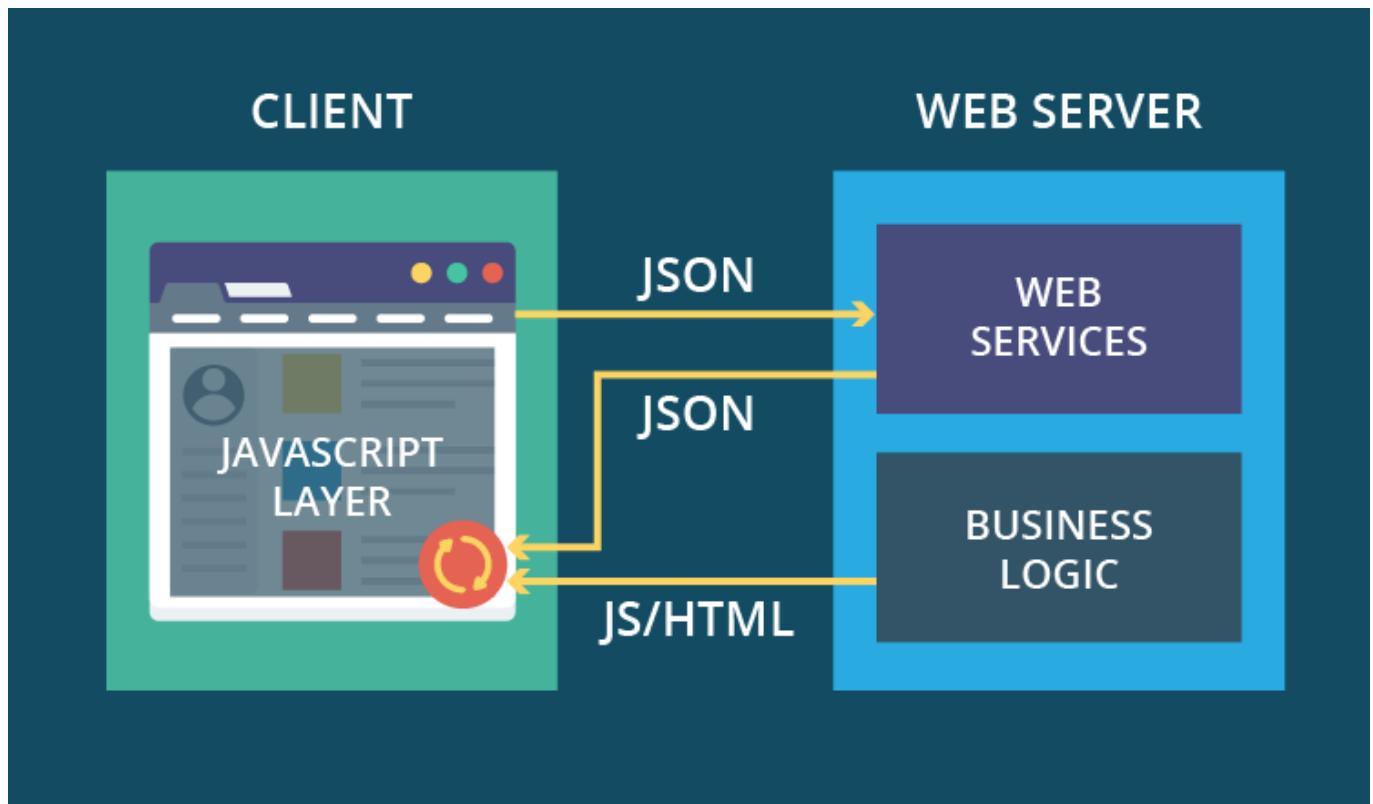
Avec les mises à jour des widgets en temps réel, ce type est plus dynamique, compatible avec les appareils mobiles et presque aussi populaire que le type suivant.

Cependant, cette architecture d'application Web nécessite un temps de développement plus long et est moins sécurisée en raison du transfert partiel de la logique de l'application vers le client exposé.

Single Page Application

Il s'agit de l'architecture d'application Web la plus moderne, l'utilisateur ne téléchargez qu'une seule page.

Côté client, cette page comporte une couche **JavaScript** qui peut communiquer librement avec des services Web sur le serveur et, à l'aide des données des services Web, se mettre à jour en temps réel.



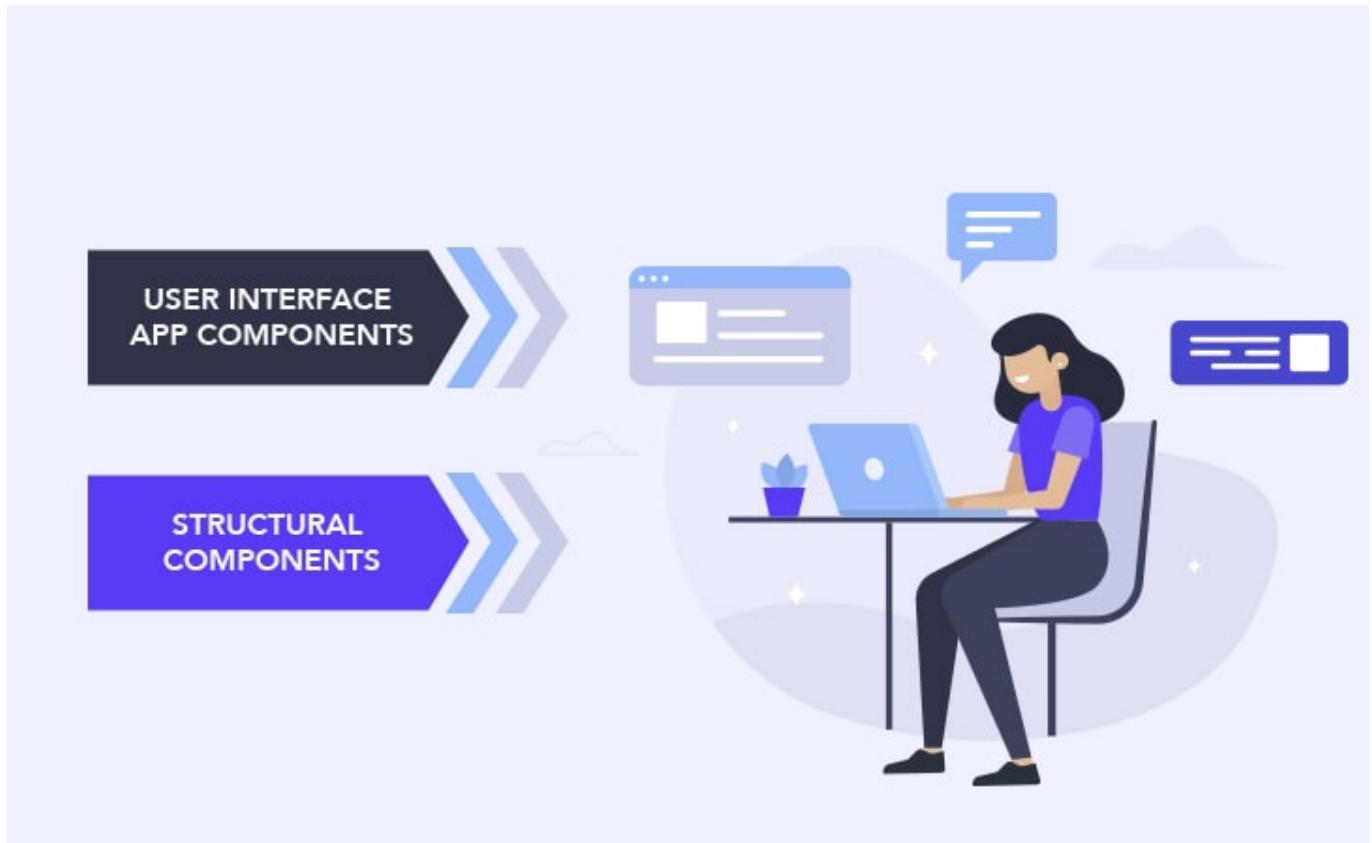
Les fragments de données transférés du serveur au client sont ici minimes, surtout par rapport au premier type.

C'est une application Web très agile, réactive et légère qui peut facilement être transformée en application mobile à l'aide d'emballeurs hybrides tels que **Cordova / PhoneGap**.

Composants

Les architectures d'applications Web comprennent divers composants qui sont séparés en deux catégories de composants:

- « **User Interface Components.** »
- « **Structural Components.** »

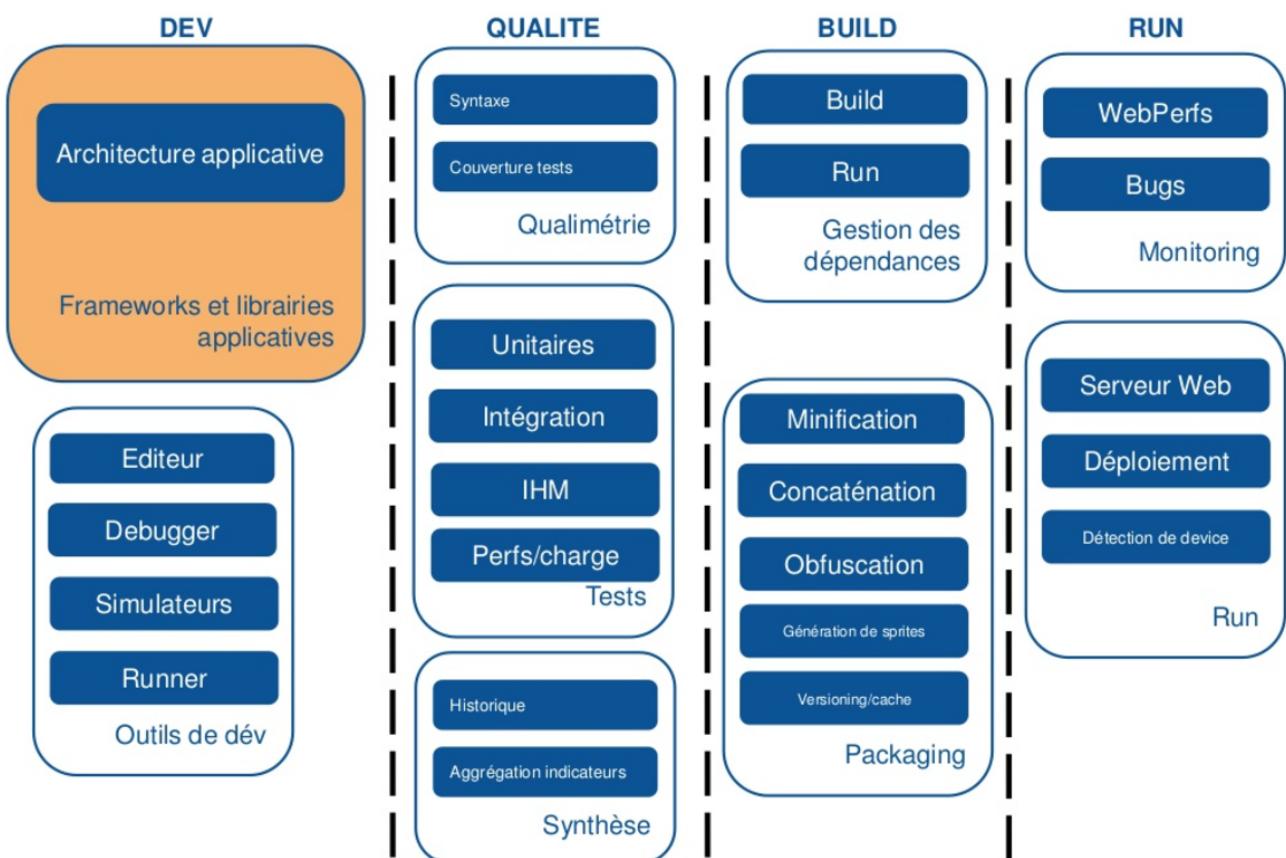


Composants d'interface utilisateur

Référence aux pages Web ayant pour rôle d'afficher des données, des paramètres et des configurations.

Liés à l'interface / expérience plutôt qu'au développement et, par conséquent, traite des tableaux de bord d'affichage, des paramètres de configuration, des notifications, des journaux, etc.

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



Outils indispensables pour le développeur Front End.

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

Il est possible de classer les outils selon trois catégories :

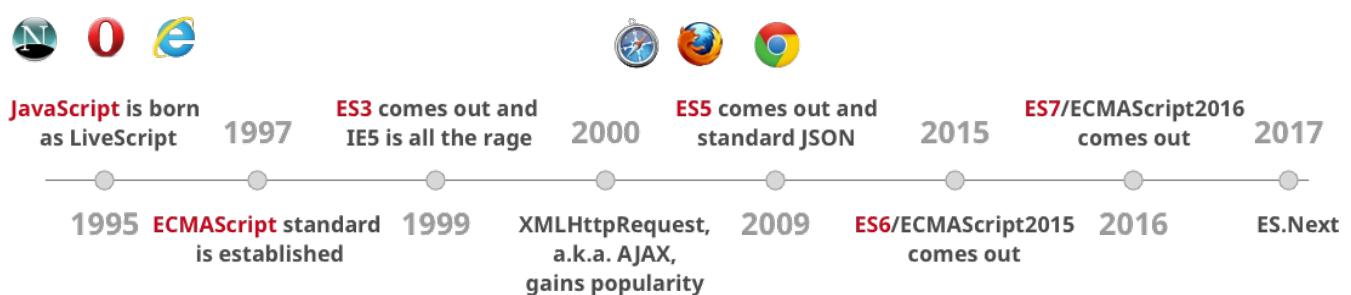
- **Logique** Dépendances tiers , tel que les différents frameworks.
- **Technique** ayant un impact sur la plate-forme de développement pour l'automatisation et le contrôle des tâches par exemple.
- **Productivité** toute autre solution non indispensable permettant d'accélérer le de développement, allant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



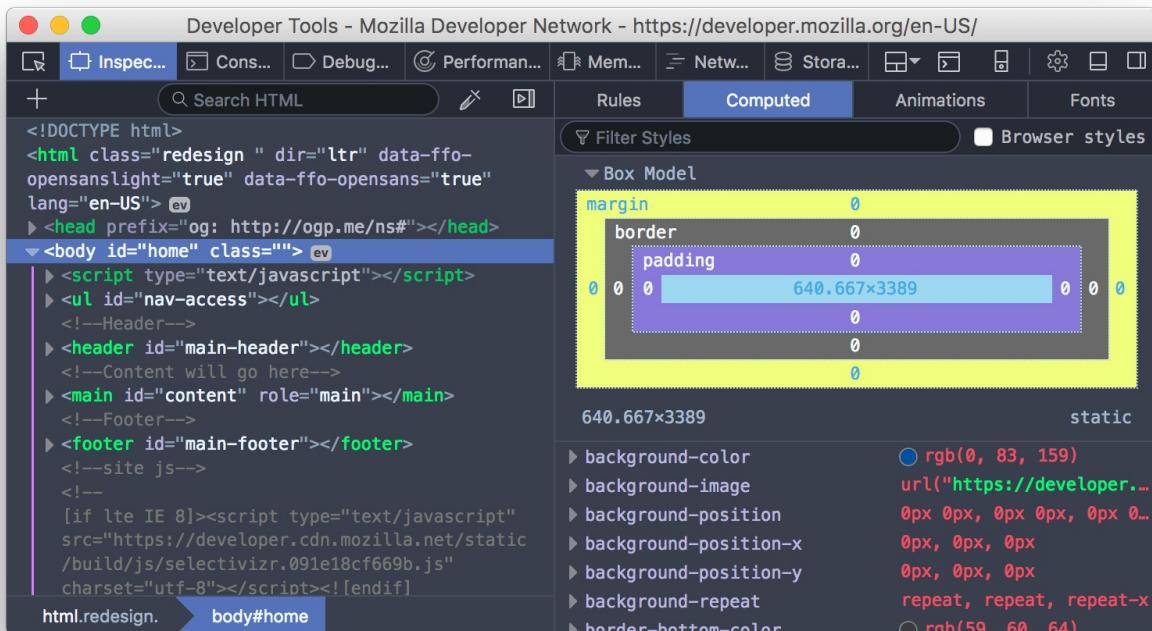
Configurer un environnement de développement moderne.

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

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 :

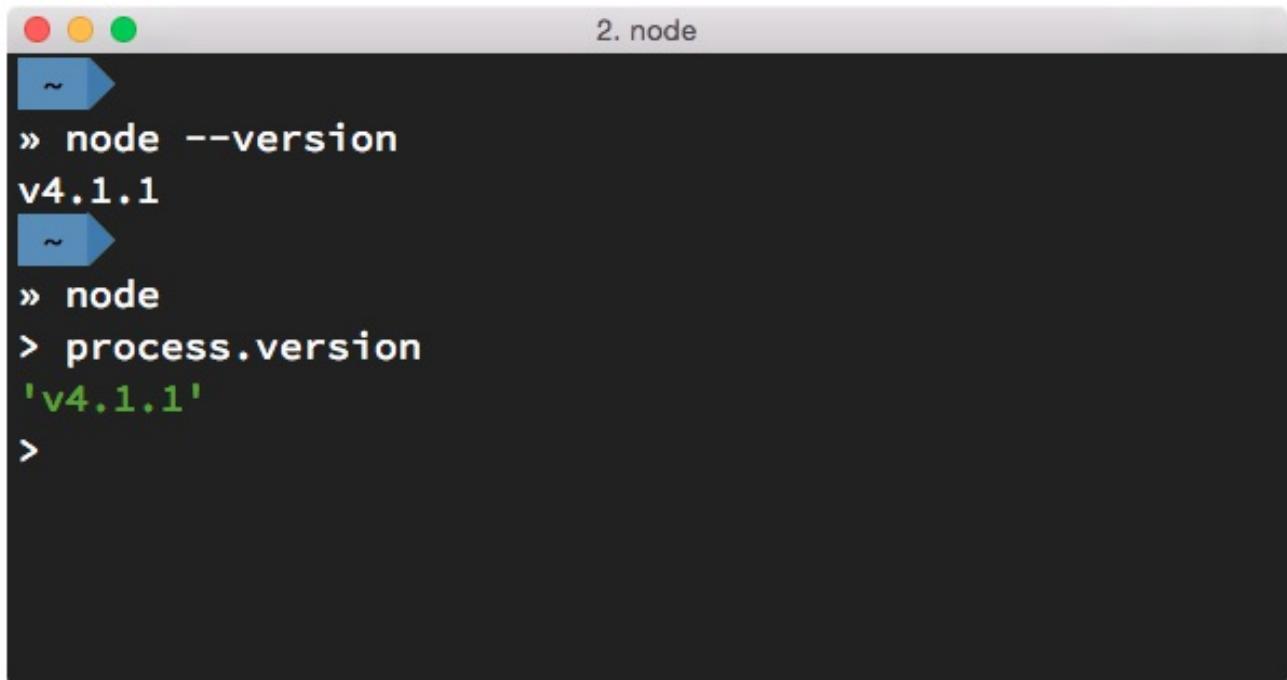
```
$> node --version  
x.x.x  
  
$> npm --version  
x.x.x  
  
$> git --version  
x.x.x
```

✓ Installation réussie !

Node.js utilitaire de développement.

Installation de Node.JS :

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



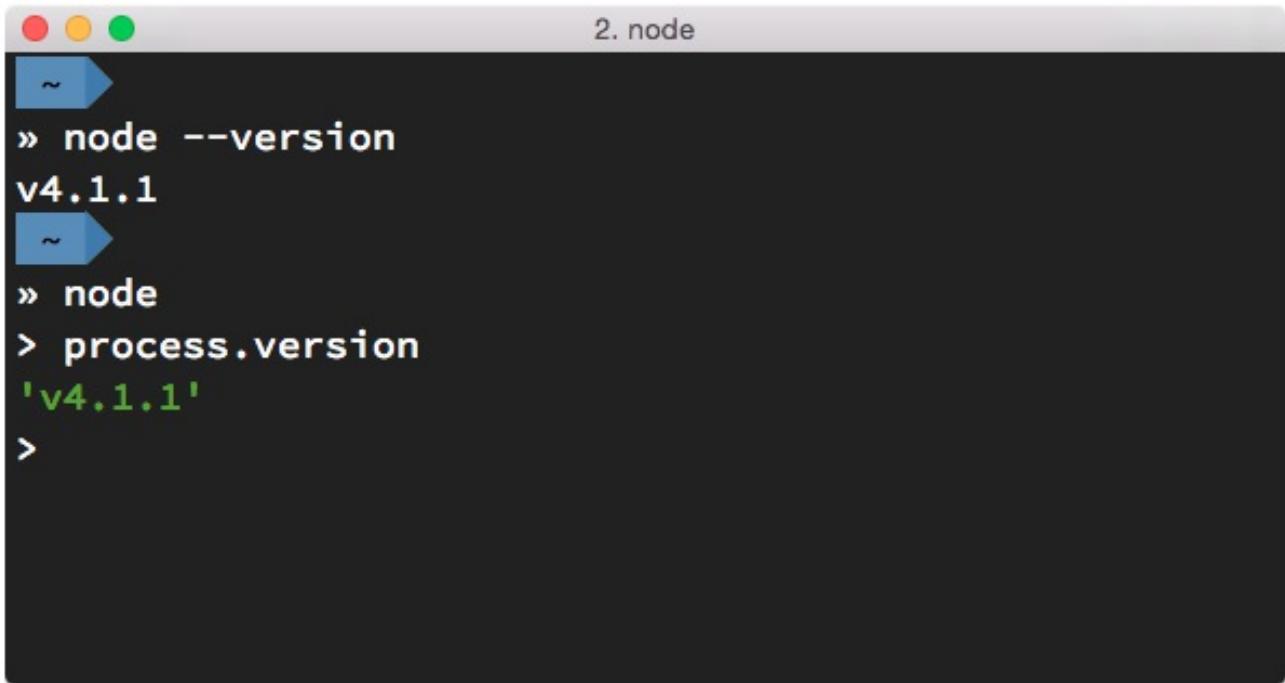
A screenshot of a macOS terminal window titled "2. node". The window has the standard OS X title bar with red, yellow, and green buttons. The terminal itself is black with white text. It shows the following command-line session:

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

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

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

Aisance avec l'invite de commande windows.



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

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

Manipulation 1

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

Manipulation 2

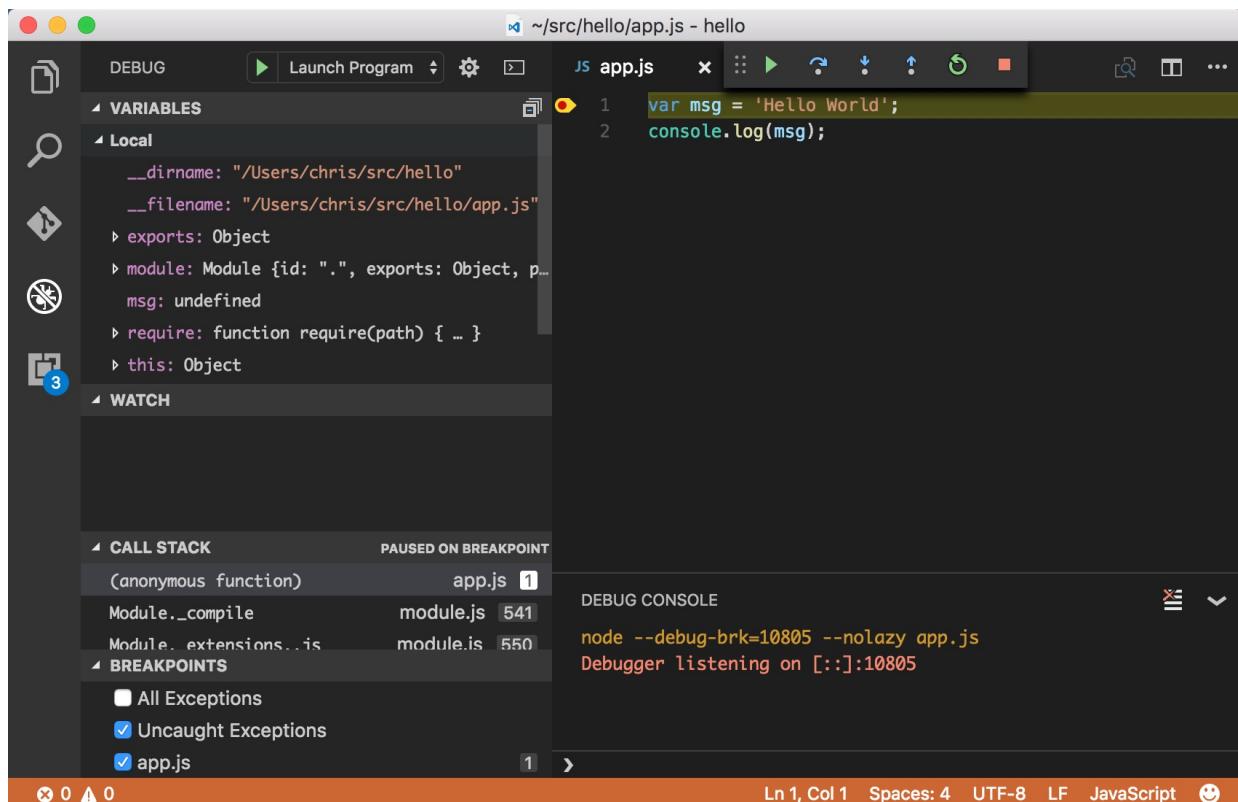
A l'aide de l'explorateur windows, se placer dans le dossier **workshops** Saisir **cmd + ENTER** dans la barre d'adresse

Manipulation 3

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

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

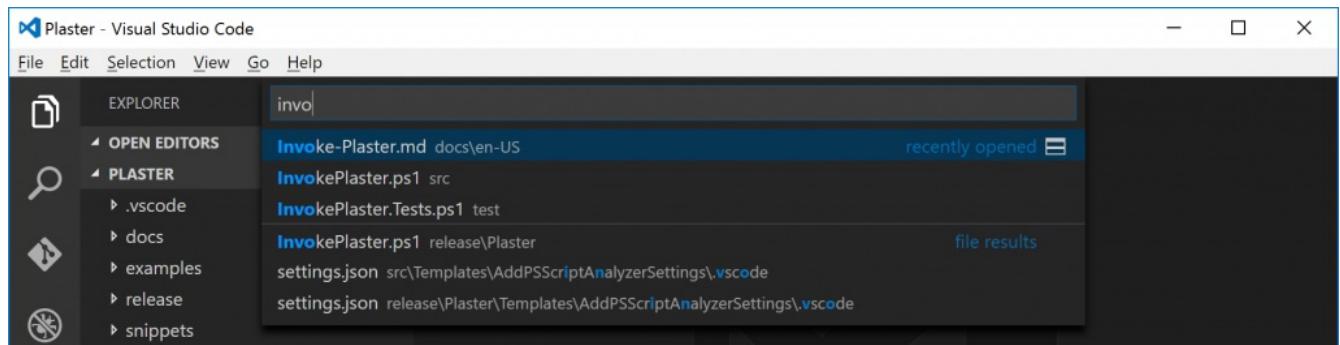
VS Code apporte toutes les fonctionnalités d'un éditeur moderne, sans nécessiter un investissement de formation.



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

Préparer son environnement.

Installer Node n'est pas très compliqué. Il existe cependant plusieurs mécanismes d'installation. Ces mécanismes vont du téléchargement d'un installateur à une compilation manuelle *via* un terminal.

Les installateurs, les binaires et les sources de Node sont disponibles sur le site officiel <https://nodejs.org/en/>

Téléchargez l'installateur adapté.

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

À noter il est recommandé une version de **Node 10+ et NPM 6+**

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

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

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
```

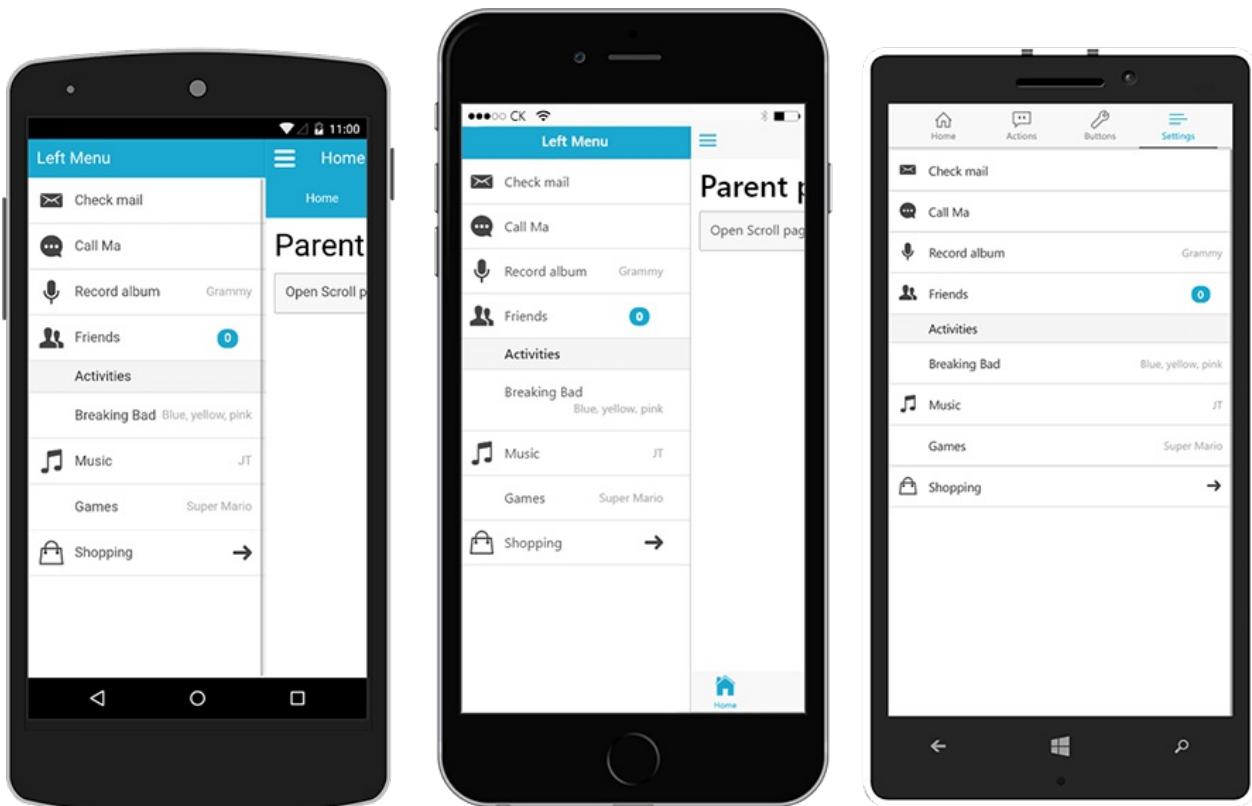
Installation des modules (CTRL+C ?)

Installer les modules suivants :

```
$> npm i -g lite-server
```

lite-server facilite le développement

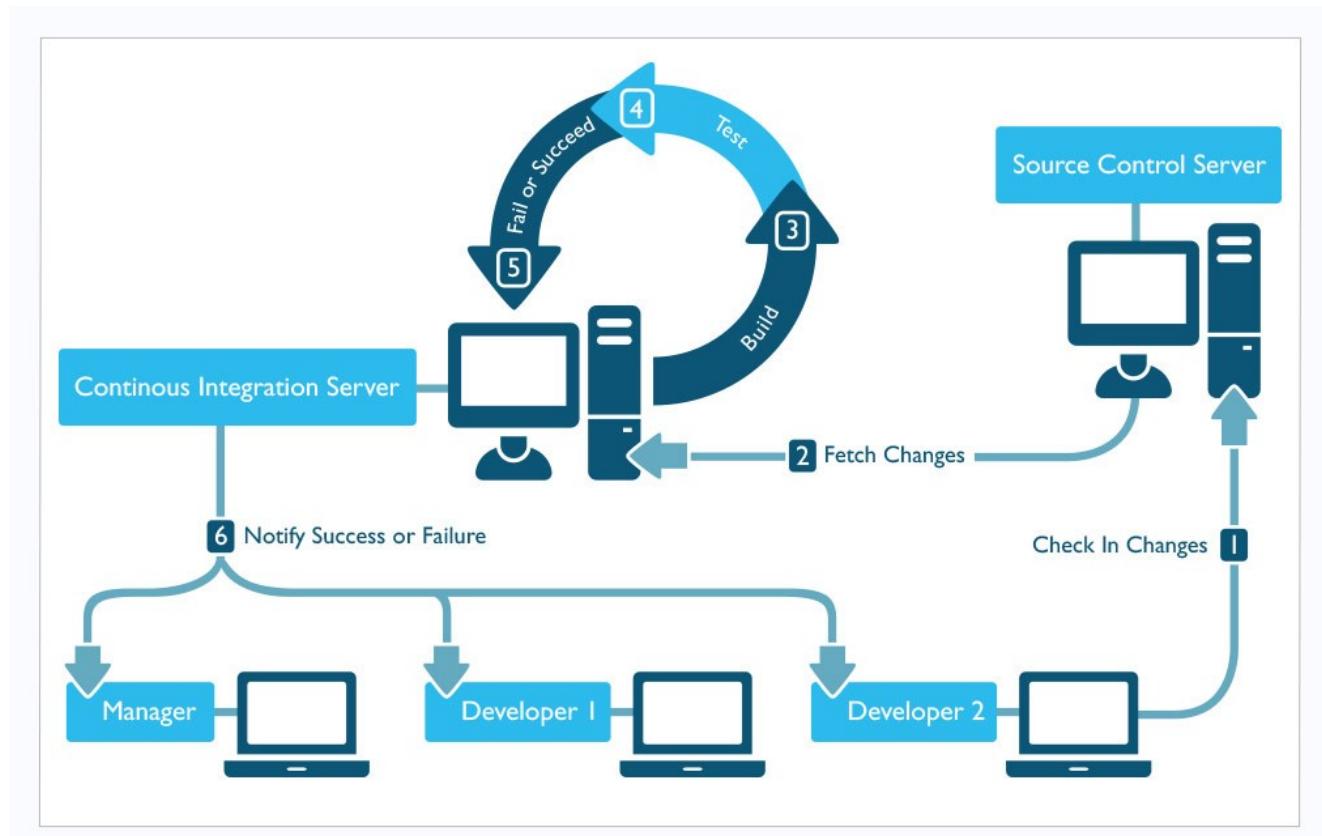
Synchronisation “multi-device”.



Il est utile de pouvoir afficher une application sur plusieurs terminaux.
lite-server fournit cette fonctionnalité par défaut

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.



Développement JavaScript : rappels.

Bonnes pratiques ECMAScript 5.

Il est important d'établir un socle de compétences solides sur JavaScript dans sa norme courante, communément appelée **ES5** avant d'aborder la 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 simple pour confirmer le socle de compétences.

- **Portée & Closure:** Saviez-vous que JavaScript utilise une portée lexicale basée 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` ?
- **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 “callback enfer”?

ES5 “use strict” et méthodes moins connues.

Tout vos scripts 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.

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 est risqué 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(){}}}`
- 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 `f(a, b, b){}`

Ces erreurs sont bienvenues car elles révèlent de mauvaises pratiques.

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

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.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

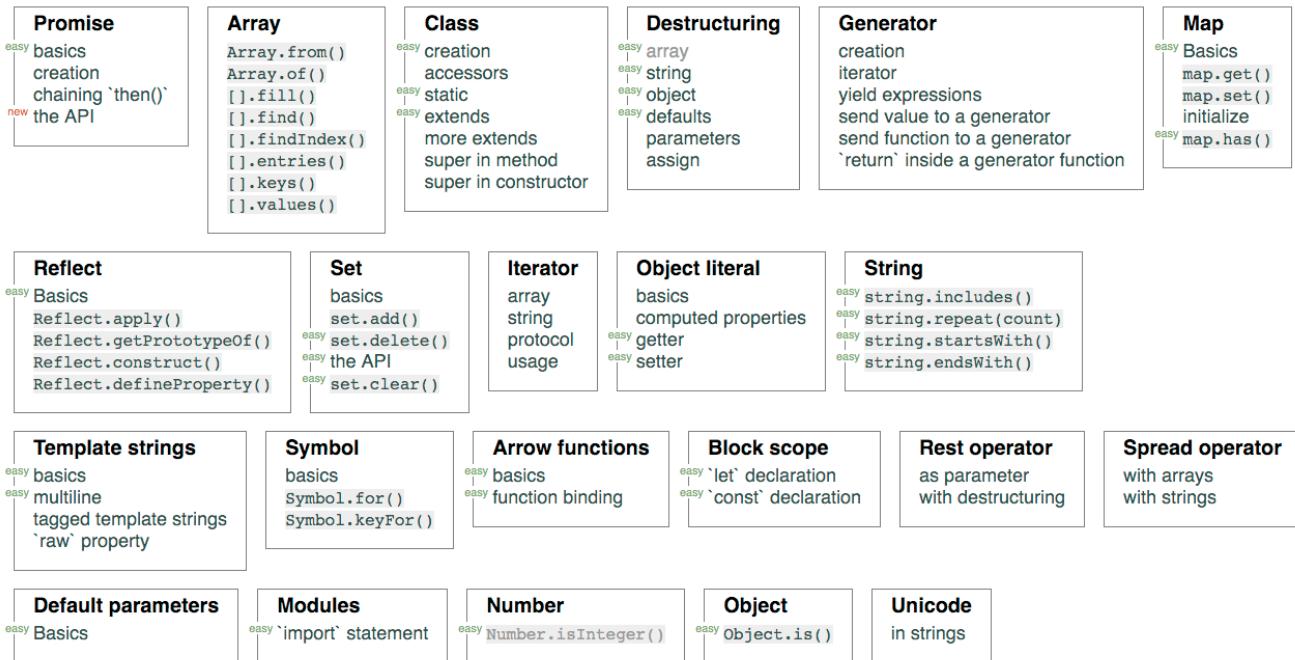
La méthode reduce() applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite) afin de la réduire à une seule valeur.

```
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)
    });
}
```

ES6/2015 : Evolutions syntaxiques fondamentales.

Aperçu général des apports.



Interpolation d'expression

- Le code desubstitution peut être toute expression JavaScript, appels de fonction, logique et arithmétique sont autorisés.
- Si l'une des valeurs n'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: `\``.
- 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.

“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 similaire à la fonction en C#, Java 8 et CoffeeScript.

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

```
var odds = evens.map(v => v + 1);
var nums = evens.map((v, i) => v + i);
var pairs = evens.map(v => ({even: v, odd: v + 1}));  
  
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});  
  
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
}
```

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

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

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

Constantes

La déclaration `const` permet de déclarer un identifiant constant.

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);
};
```

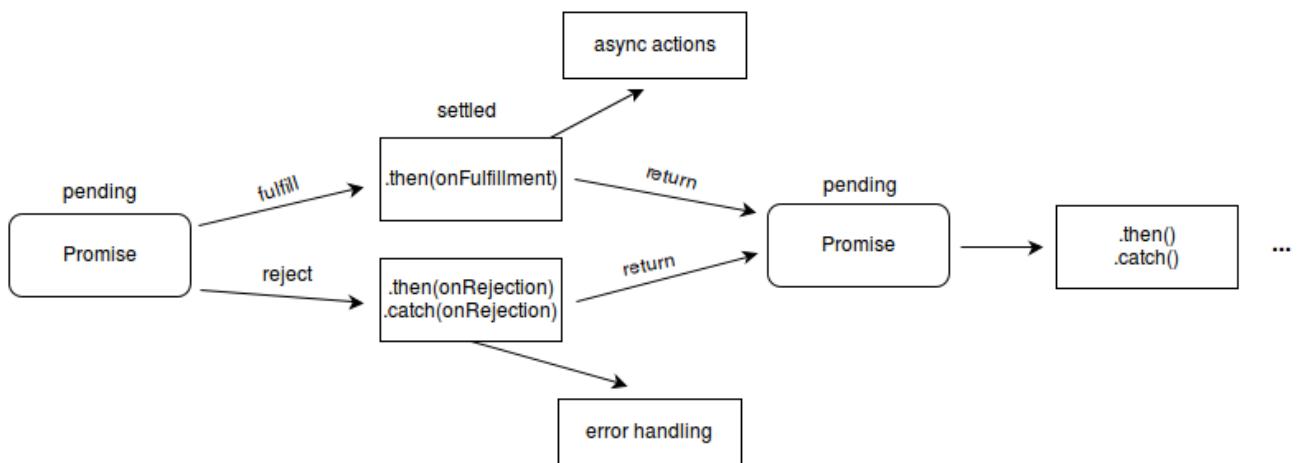
Nouvelles API JavaScript avec ES6.

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 consommations

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

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* de la même façon qu'il y a des expressions de fonctions et des déclarations de fonctions, on aura deux syntaxes :

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.

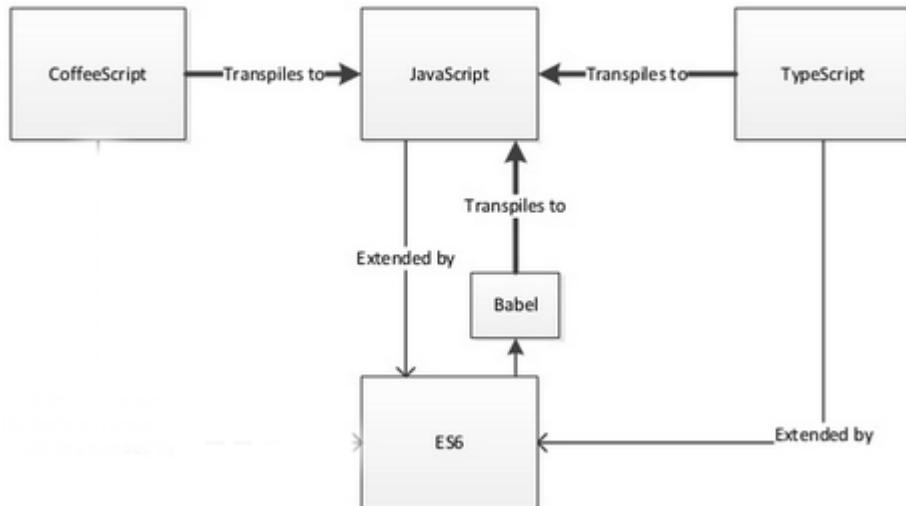
```
// anonyme
var Polygone = class {
    constructor(hauteur, largeur) {
        this.hauteur = hauteur;
        this.largeur = largeur;
    }
};

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

Outils indispensables. Babel, Traceur et Typescript.

À date le système natif de chargement n'est pas implémenté dans les navigateurs.

De plus les navigateurs ne supportent 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.

Choix du “transpiler” : présentation des solutions.

Il existe beaucoup de “transpiler” capables de convertir du code ES6 en ES5

Références

[Babel](#)

[Traceur compiler](#)

* [TypeScript](#)

À noter Ces trois références principales existent sous la forme de tâches pour **gulp** et **grunt**

Avec browserify

- [es6ify](#)
- [babelify](#)
- [es6-transpiler](#)

Modules

- Square’s [es6-module-transpiler](#)

Autres

Facebook’s [regenerator](#)

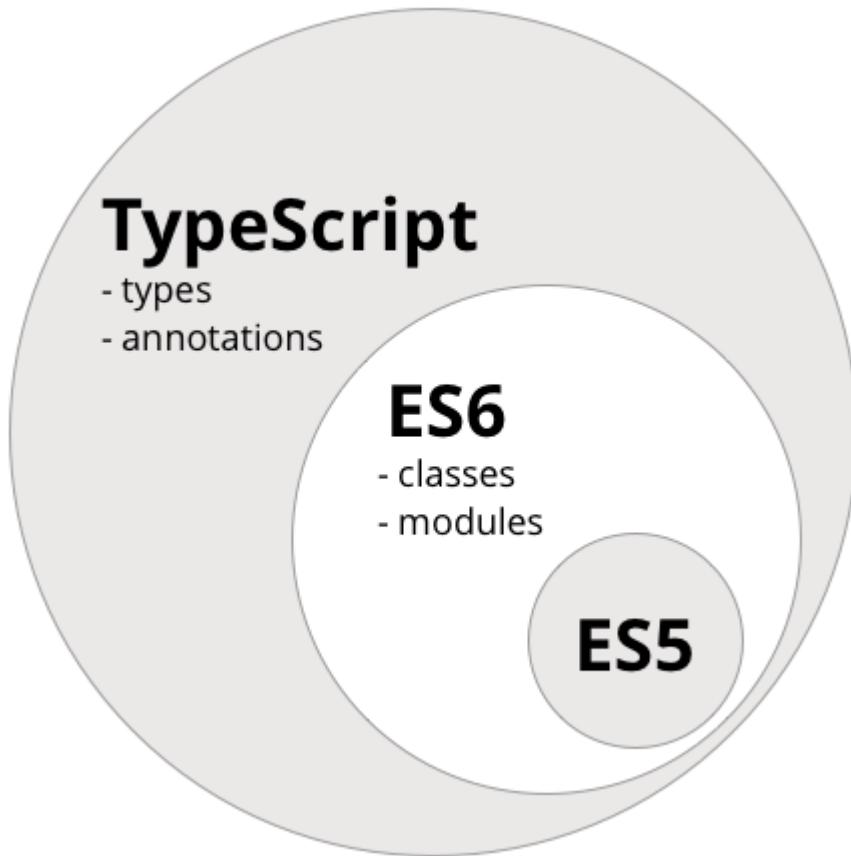
Facebook’s [jstransform](#)

[Et encore bien d’autres](#)

TypeScript en détail, configuration.

TypeScript est un langage facilitant le développement d'applications larges et "scalables", écrites en JavaScript.

TypeScript apporte un [support supérieur à 58%](#) des nouveautés syntaxique ES6/2015



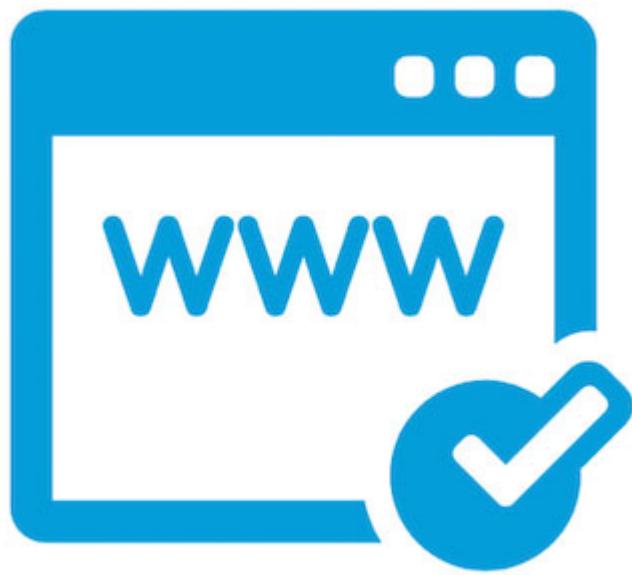
Le **transpiler** TypeScript permet l'ajout de concepts *propre* à la programmation orientée objet par **classe** tels que :

- Les interfaces
- Les génériques
- Le typage statique
- Décorateurs

C'est une surcouche de JavaScript : tout le code JavaScript est valide en TypeScript ce qui permet de l'ajouter de façon transparente à n'importe quel projet.

Parceque le code TypeScript est *transcomplié* en JavaScript en tant que language cible on le qualifie de [Superset JavaScript](#).

Le [Playground](#) permet de tester la syntaxe TypeScript en illustrant le mécanisme de transpilation.



Premiers pas avec Vue.js

Premiers pas avec Vue.js

Vue.js, qu'est-ce que c'est ?

Vue (prononcé /vju:/, comme le terme anglais **view**) est un **framework évolutif** pour construire des interfaces utilisateur.

Vue a été conçu et pensé pour pouvoir être adopté de manière incrémentale.

Le cœur de la bibliothèque est concentré uniquement sur la partie vue, et il est vraiment simple de l'intégrer avec d'autres bibliothèques ou projets existants.

D'un autre côté, Vue est tout à fait capable de faire tourner des applications web monopages quand il est couplé avec [des outils modernes](#) et [des bibliothèques complémentaires](#).

Pour commencer

Installation

La manière la plus simple d'essayer Vue.js est d'utiliser l'exemple Hello World sur [JSFiddle](#).

Vous pouvez aussi simplement [créer un fichier index.html](#) et ajouter Vue avec :

```
<!-- development version, includes helpful console warnings -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

ou :

```
<!-- production version, optimized for size and speed -->
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

La page d'[installation](#) vous offre d'autres manières d'installer Vue.

Notez que nous **ne recommandons pas** aux débutants de commencer avec **vue-cli**, surtout si vous n'êtes pas encore familier avec les outils de *build* basés sur Node.js.

Principes clés de Vue.js.

- VueJS.
- Composant.
 - Template.
 - Binding.
 - Rendu déclaratif.
- Directives.

Rendu déclaratif :

[Essayez cette partie sur Scrimba \(EN\)](#)

Au cœur de Vue.js, il y a un système qui va nous permettre de faire le rendu des données déclarativement dans le DOM en utilisant simplement cette syntaxe de template :

```
<div id="app" class="demo">
  {{ message }}
</div>
<script>
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue !'
  }
})
</script>
```

Les données et le DOM sont maintenant couplés, et tout est à présent **réactif**.

Comment s'en rendre compte ? Ouvrez la console JavaScript de votre navigateur et attribuez à **app.message** différentes valeurs.

En plus de l'interpolation de texte, nous pouvons également lier les attributs d'un élément comme ceci :

```
<div id="app-2" class="demo">
  <span v-bind:title="message">
    Passez votre souris sur moi pendant quelques secondes pour voir mon
    titre lié dynamiquement !
  </span>
</div>
<script>
var app2 = new Vue({
  el: '#app-2',
  data: {
    message: 'Vous avez affiché cette page le ' + new
Date().toLocaleString()
  }
})
</script>
```

L'attribut **v-bind** que vous voyez est appelé une **directive**.

Les directives sont préfixées par **v-** pour indiquer que ce sont des attributs spéciaux fournis par Vue, elles appliquent un comportement réactif spécifique au DOM après rendu.

Ici cela veut basiquement dire : « garde l'attribut **title** de cet élément à jour avec la propriété **message** de l'instance de Vue ».

Conditions et boucles

[Essayez cette partie sur Scrimba \(EN\)](#)

Il est assez simple de permuter la présence d'un élément :

```
<div id="app-3" class="demo">
  <span v-if="seen">Maintenant vous me voyez</span>
</div>
<script>
var app3 = new Vue({
  el: '#app-3',
  data: {
    seen: true
  }
})
</script>
```

Côté console entrez **app3.seen = false**. Vous devriez voir le message disparaître.

Nous pouvons lier des données non seulement aux textes et attributs, mais également à la **structure** du DOM.

De plus, Vue fournit un puissant système d'effets de transition qui peut automatiquement appliquer des [effets de transition](#) quand des éléments sont insérés/mis à jour/enlevés par Vue.

Il existe quelques autres directives, chacune avec leur propre fonction spécifique. Par exemple, la directive **v-for** peut être utilisée pour afficher une liste d'éléments en provenance d'un tableau de données.

```
<div id="app-4" class="demo">
  <ol>
    <li v-for="todo in todos">
      {{ todo.text }}
    </li>
  </ol>
</div>
<script>
var app4 = new Vue({
  el: '#app-4',
  data: {
    todos: [
      { text: 'Apprendre JavaScript' },
      { text: 'Apprendre Vue.js' },
      { text: 'Apprendre les composants' },
      { text: 'Apprendre les transitions' }
    ]
  }
})
```

```

        { text: 'Apprendre Vue' },
        { text: 'Créer quelque chose de génial' }
    ]
}
})
</script>

```

Dans la console, entrez `app4.todos.push({ text: 'Nouvel élément' })`.

Gestion des entrées utilisateur

[Essayez cette partie sur Scrimba \(EN\)](#)

Afin de permettre aux utilisateurs d'interagir avec votre application, nous pouvons utiliser la directive `v-on` pour attacher des écouteurs d'évènements qui invoquent des méthodes sur nos instances de Vue :

```

<div id="app-5" class="demo">
  <p>{{ message }}</p>
  <button v-on:click="reverseMessage">Message retourné</button>
</div>
<script>
var app5 = new Vue({
  el: '#app-5',
  data: {
    message: 'Hello Vue.js !'
  },
  methods: {
    reverseMessage: function () {
      this.message = this.message.split('').reverse().join('')
    }
  }
})
</script>

```

Toutes les manipulations de DOM sont prises en charge par Vue, ainsi le code que vous écrivez se concentre sur la logique sous-jacente.

Vue fournit aussi la directive `v-model` qui fait de la liaison de données bidirectionnelle entre les champs d'un formulaire et l'état de l'application.

```

<div id="app-6" class="demo">
  <p>{{ message }}</p>
  <input v-model="message">
</div>
<script>
var app6 = new Vue({
  el: '#app-6',
  data: {
    message: 'Hello Vue !'
  }
})
</script>

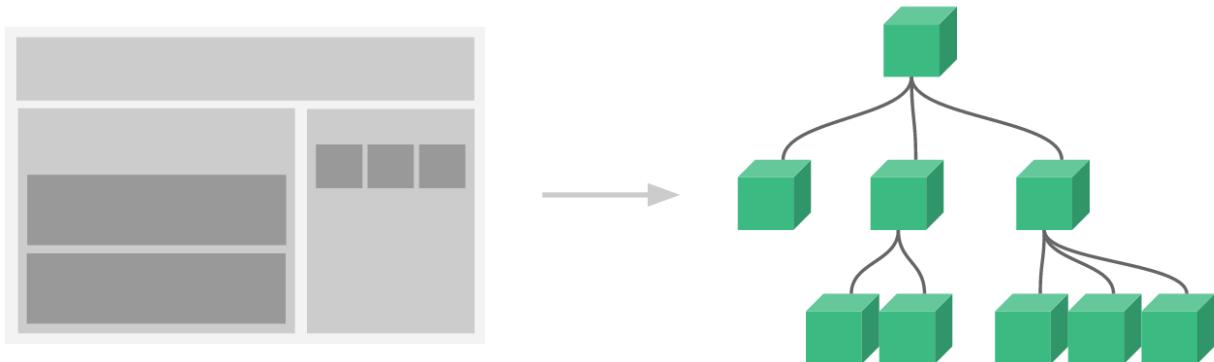
```

```
})  
</script>
```

Composer avec des composants

Essayez cette partie sur Scrimba (EN)

Le système de **composant est un concept important de Vue**, il permet de construire de plus grosses applications composées de plus petits composants réutilisables et autonomes.



Dans Vue, un composant est essentiellement une instance de Vue avec des options prédéfinies.

Déclarer un composant avec Vue est très simple :

```
// Définit un nouveau composant appelé todo-item
Vue.component('todo-item', {
  template: '<li>Ceci est une liste</li>'
})
```

Maintenant nous pouvons l'insérer dans le template d'un autre composant :

```
<ol>
  <!-- Crée une instance du composant todo-list -->
  <todo-item></todo-item>
</ol>
```

Pour être capables de passer les données de la portée parente dans le composant enfant,修改ons la définition du composant pour lui permettre d'accepter une *prop* :

```
Vue.component('todo-item', {
  // Le composant todo-item accepte maintenant une
  // « prop » qui est comme un attribut personnalisé.
  // Cette prop est appelée todo.
  props: ['todo'],
  template: '<li>{{ todo.text }}</li>'
})
```

Maintenant nous pouvons passer la liste dans chaque composant répété en utilisant `v-bind` :

```
<div id="app-7" class="demo">
  <ol>
    <todo-item v-for="item in groceryList" v-bind:todo="item"
:key="item.id"></todo-item>
  </ol>
</div>
<script>
Vue.component('todo-item', {
  props: ['todo'],
  template: '<li>{{ todo.text }}</li>'
})
var app7 = new Vue({
  el: '#app-7',
  data: {
    groceryList: [
      { id: 0, text: 'Légumes' },
      { id: 1, text: 'Fromage' },
      { id: 2, text: 'Tout ce que les humains sont supposés manger' }
    ]
  }
})
</script>
```

Nous avons réussi à séparer notre application en deux plus petites unités, et chaque enfant est raisonnablement bien découpé de son parent via l'utilisation des props.

Nous pouvons maintenant améliorer notre `<todo-item>` avec un template et une logique plus complexes sans affecter l'application parente.

Pour une grosse application, il est nécessaire de la diviser entièrement en composants afin de rendre le développement plus abordable.

```
<div id="app">
  <app-nav></app-nav>
  <app-view>
    <app-sidebar></app-sidebar>
    <app-content></app-content>
  </app-view>
</div>
```

Vue.js vs React vs Angular, et autres frameworks...

React

React et Vue ont beaucoup en commun. Tous les deux :

- utilisent un DOM virtuel,
- fournissent des composants de vue réactifs et composables,
- restent concentrés sur le cœur de la bibliothèque, en déléguant le routage et la gestion d'état à des bibliothèques connexes.

Plusieurs des sections plus bas peuvent être incorrectes du fait de la récente mise à jour de React 16+.

Performance à l'exécution

Vue comme React offrent des performances similairement rapide, tellement rapide que ce n'est pas un facteur de décision entre les deux.

Efforts d'optimisation

Avec React, quand l'état d'un composant change, cela enclenche de nouveau le rendu de tous ses sous-composants, en partant de ce composant comme racine.

Avec Vue, les dépendances d'un composant sont automatiquement tracées durant le rendu, ainsi le système sait précisément quels composants ont besoin d'être rafraîchis.

HTML & CSS

Avec React tout est JavaScript. Pas seulement la structure HTML exprimée via JSX, mais également le CSS qui tend également à être géré avec du JavaScript.

Vue adopte les technologies classiques du Web et construit par-dessus celles-ci. Pour vous montrer ce que cela signifie, nous allons nous plonger dans plusieurs exemples.

JSX vs Templates

Avec React, tous les composants expriment leur UI à travers des fonctions de rendu utilisant JSX, une syntaxe déclarative comme XML qui fonctionne au sein de JavaScript.

Vue, utilise également des [fonctions de rendu](#) et même [un support de JSX](#), car parfois, nous avons besoin de cette puissance. Cependant, pour une expérience par défaut nous offrons les templates comme une alternative simple.

- Les templates semblent tout simplement plus naturels à écrire.
- Les templates basés sur du HTML sont plus simples à migrer progressivement depuis une application existante qui souhaiterait tirer avantage des fonctionnalités de réactivités de Vue.
- Cela est également plus simple pour les designers et les développeurs moins expérimentés de comprendre et contribuer au code de base.

CSS à portée limitée au composant

À moins que vous ne répartissiez les composants dans plusieurs fichiers (par exemple avec les [Modules CSS](#)), limiter la portée du CSS dans React est souvent fait par des solutions « CSS-in-JS » (par ex : [styled-components](#), [glamorous](#) et [emotion](#)).

Beaucoup de bibliothèques « CSS-in-JS » supportent Vue (par ex. [styled-components-vue](#) et [vue-emotion](#)). La grande différence entre React et Vue ici est que la méthode de style par défaut de Vue est plus familière avec l'utilisation d'une balise [`<style>`](#) dans [un composant monofichier](#).

[Un composant monofichier](#) vous donne l'accès complet au CSS au sein d'un même fichier comme le reste du code du composant.

```
<style scoped>
  @media (min-width: 250px) {
    .list-container:hover {
      background: orange;
    }
  }
</style>
```

L'attribut optionnel [scoped](#) encapsule automatiquement ce CSS dans votre composant en ajoutant un unique attribut (par exemple [data-v-21e5b78](#)) à l'élément en compilant [.list-container:hover](#) en [.list-container \[data-v-21e5b78\]:hover](#).

Enfin, le style dans un composant monofichier Vue est vraiment flexible. Avec [vue-loader](#), vous pouvez utiliser n'importe quel préprocesseur, postprocesseur et même une intégration profonde avec les [Modules CSS](#) ; le tout dans un élément [`<style>`](#).

AngularJS (Angular 1)

Une partie de la syntaxe de Vue ressemblera très fortement à celle de AngularJS (ex : `v-if` vs `ng-if`).

Cela est dû au fait qu'il y a beaucoup de choses pour lesquelles AngularJS a vu juste et que cela a été une source d'inspiration pour Vue très tôt dans son développement.

Complexité

Vue est bien plus simple que AngularJS, autant en termes d'API que d'architecture.

Liaison de données

AngularJS utilise la liaison de données bidirectionnelle entre les **scopes**, tandis que Vue impose un flux de données unidirectionnel entre les composants.

Directives vs. Composants

Vue fait une distinction plus claire entre directives et composants.

Les directives sont conçues pour encapsuler uniquement des manipulations du DOM, tandis que les composants sont des unités indépendantes ayant leur propre vue et logique de données.

Dans AngularJS, les directives peuvent faire un peu tout alors que les composants sont un type plus spécifique de directive.

Runtime Performance

Vue a de meilleures performances et il est bien plus optimisé car il n'utilise pas de *dirty checking*. AngularJS devient lent quand il y a un grand nombre d'observateurs, car chaque fois que quelque chose change dans le **scope**, tous les observateurs ont besoin d'être réévalués. De plus, le *digest cycle* peut avoir besoin de s'exécuter plusieurs fois pour se « stabiliser » si un observateur déclenche une autre mutation.

Angular (Plus connu sous le nom de Angular 2)

Nous avons une section dédiée pour le nouvel Angular car c'est vraiment un framework complètement différent de AngularJS. Par exemple, il comporte un système de composants de première classe, beaucoup de détails d'implémentation ont été complètement réécrits, et l'API a également changé assez drastiquement.

TypeScript

Angular demande essentiellement l'utilisation de TypeScript, basant toute sa documentation et son apprentissage sur des ressources en TypeScript.

TypeScript a ses propres bénéfices ; la vérification de type peut-être très utile dans de grosses applications, et peut-être un grand gain de productivité pour les développeurs Java ou C#.

Pour finir, sans être profondément intégré à TypeScript comme peut l'être Angular, Vue offre également un [typage officiel](#) et des [décorateurs officiels](#) à ceux qui souhaiteraient utiliser TypeScript avec Vue.

Performance à l'exécution

Les deux frameworks sont exceptionnellement rapides avec des métriques similaires au benchmark.

Taille

Un projet complet Vue 2, avec Vuex et Vue Router inclus (~30ko gzippé), est toujours significativement plus léger qu'une application avec compilation anticipée et générée par [angular-cli](#) (~65ko gzippée).

Courbe d'apprentissage

Pour commencer avec Vue, vous avez seulement besoin de connaissances en HTML et JavaScript ES5 (c.-à-d. JavaScript de base). Avec ces compétences de base, vous pouvez commencer à construire des applications complexes sans perdre des jours à lire [la documentation](#).

La surface API du framework est simplement plus grosse et les utilisateurs ont besoin de se familiariser eux-mêmes avec beaucoup de concepts avant d'être productifs.

Manifestement, la complexité d'Angular est largement due au fait que son design est conçu uniquement pour répondre à de grandes et complexes applications.

Ember

Ember est un framework plein de fonctionnalités qui a été conçu pour prendre beaucoup de décisions à la place du développeur.

Il fournit beaucoup de conventions et une fois que vous êtes assez familiers avec celles-ci, il peut vous rendre réellement productif. Cependant, cela signifie que la courbe d'apprentissage est élevée et la flexibilité en pâtit. C'est un compromis lorsque vous essayez de choisir entre un framework avec des opinions tranchées et une bibliothèque avec un ensemble d'outils à couplage faible qui travaillent ensemble. Ces derniers vous offrent la liberté mais également vous laissent prendre plus de décisions d'architecture.

Cela dit, il serait probablement plus judicieux de faire une comparaison entre le cœur de Vue et le système de template d'Ember et les couches de [modèles d'objet](#) :

- Vue fournit une réactivité discrète sur de purs objets JavaScript et des propriétés calculées automatiquement. Avec Ember, vous devez encapsuler le tout dans des objets Ember et manuellement déclarer toutes les dépendances des propriétés calculées.
- La syntaxe des templates de Vue exploite toute la puissance des expressions JavaScript alors que les expressions Handlebars et les aides à la syntaxe sont intentionnellement limitées en comparaison.
- Côté performance, Vue surpassé Ember [avec une bonne avance](#), même après la dernière mise à jour du moteur Glimmer dans Ember 3.x. Vue regroupe automatiquement les rafraîchissements des vues par lot, alors que dans Ember, vous devez gérer manuellement les boucles d'exécution dans les situations où la performance est critique.

Knockout

Knockout fut un pionnier dans le domaine du MVVM et du suivi de dépendances, et son système de réactivité est vraiment très similaire à Vue.

C'est son [support des navigateurs](#) qui est vraiment impressionnant considérant tout ce qu'il permet de faire avec un support jusqu'à IE6 ! Vue d'un autre côté ne supporte que IE9+.

Avec le temps cependant, le développement de Knockout a ralenti et il commence à se montrer un peu âgé.

Par exemple, son système de composant manque d'un ensemble complet de hooks au cycle de vie et même si c'est un cas d'utilisation commun, l'interface pour passer des composants enfants à un composant est quelque peu laborieuse en comparaison de Vue.

Polymer

Polymer est encore un autre projet sponsorisé par Google qui a également été une source d'inspiration pour Vue.

Les composants de Vue peuvent être grosso modo comparés à ceux des éléments personnalisés de Polymer et les deux fournissent un style de développement vraiment similaire.

La plus grosse différence est que Polymer est construit sur la base des dernières fonctionnalités de Web Components et requiert donc des polyfills complexes pour fonctionner (avec des performances dégradées) dans les navigateurs qui ne supportent pas ces fonctionnalités nativement.

En revanche, Vue fonctionne sans aucune dépendance dans tous les navigateurs après IE9.

Roadmap et prochaines versions.

Compatibilité

Vue **ne** supporte **pas** IE8 et les versions antérieures, car il utilise des fonctionnalités ECMAScript 5 qui ne peuvent pas être émulées sur IE8. Cela dit, Vue supporte tous les [navigateurs compatibles ECMAScript 5](#).

Notes de version

Les notes de version détaillées pour chaque version sont disponibles sur [GitHub](#).

Installation

Inclusion directe <script>

Il suffit de télécharger et de l'inclure avec une balise script. Vue sera déclaré comme une variable globale.

N'utilisez pas la version minifiée pendant le développement.

Vous ne bénéficieriez alors pas des avertissements pour les erreurs courantes !

CDN

Pour du prototypage ou de l'apprentissage, vous pouvez utiliser la dernière version avec :

```
<script src="https://cdn.jsdelivr.net/npm/vue.js"></script>
```

Pour la production, nous vous recommandons de vous figer à une version et un build défini pour éviter les changements non compatibles des nouvelles versions :

```
<script src="https://cdn.jsdelivr.net/npm/vue@2.6.0/dist/vue.js"></script>
```

Si vous utilisez des Modules ES natif, il y a également un build compatible avec les Modules ES :

```
<script type="module">
  import Vue from
  'https://cdn.jsdelivr.net/npm/vue@2.6.0/dist/vue.esm.browser.js'
</script>
```

Vous pouvez parcourir la source du package npm à l'adresse : cdn.jsdelivr.net/npm/vue.

Vue est également disponible sur [unpkg](#) et [cdnjs](#) (cdnjs met du temps à se synchroniser ce qui signifie que la dernière version peut ne pas être encore disponible).

Assurez vous de lire la partie dédiée [aux différents builds de Vue](#) et d'utiliser la **version de production** dans vos sites publiés, en remplaçant `vue.js` par `vue.min.js`.

C'est un build plus léger optimisé pour la rapidité plutôt que l'expérience de développement.

npm

npm est la méthode d'installation recommandée lors du développement de grosses applications avec Vue.

Il s'associe bien avec des empaqueteurs de modules comme [webpack](#) ou [Browserify](#).

Vue fournit également des outils d'accompagnement pour la rédaction de [Composants Mono-fichier](#).

```
# dernière version stable  
$ npm install vue
```

Créer et gérer un projet avec vue-cli.

Vue offre un [outil de génération de projet en ligne de commande](#) qui **facilite** le démarrage d'un nouveau projet en utilisant le système de système de build de votre choix.

Vous pouvez même l'utiliser pour [prototyper instantanément](#) un composant.

- Il permet la configuration durant la génération du projet, qui peut être étendu avec des plugins [plugins](#).
- Il offre une large variété de templates pour différents objectifs et outils de build.
- Il permet pas la génération de projets depuis des [templates](#) faits par les utilisateurs, ce qui peut être particulièrement utile en entreprise avec des conventions préétablies.

Commandes utiles :

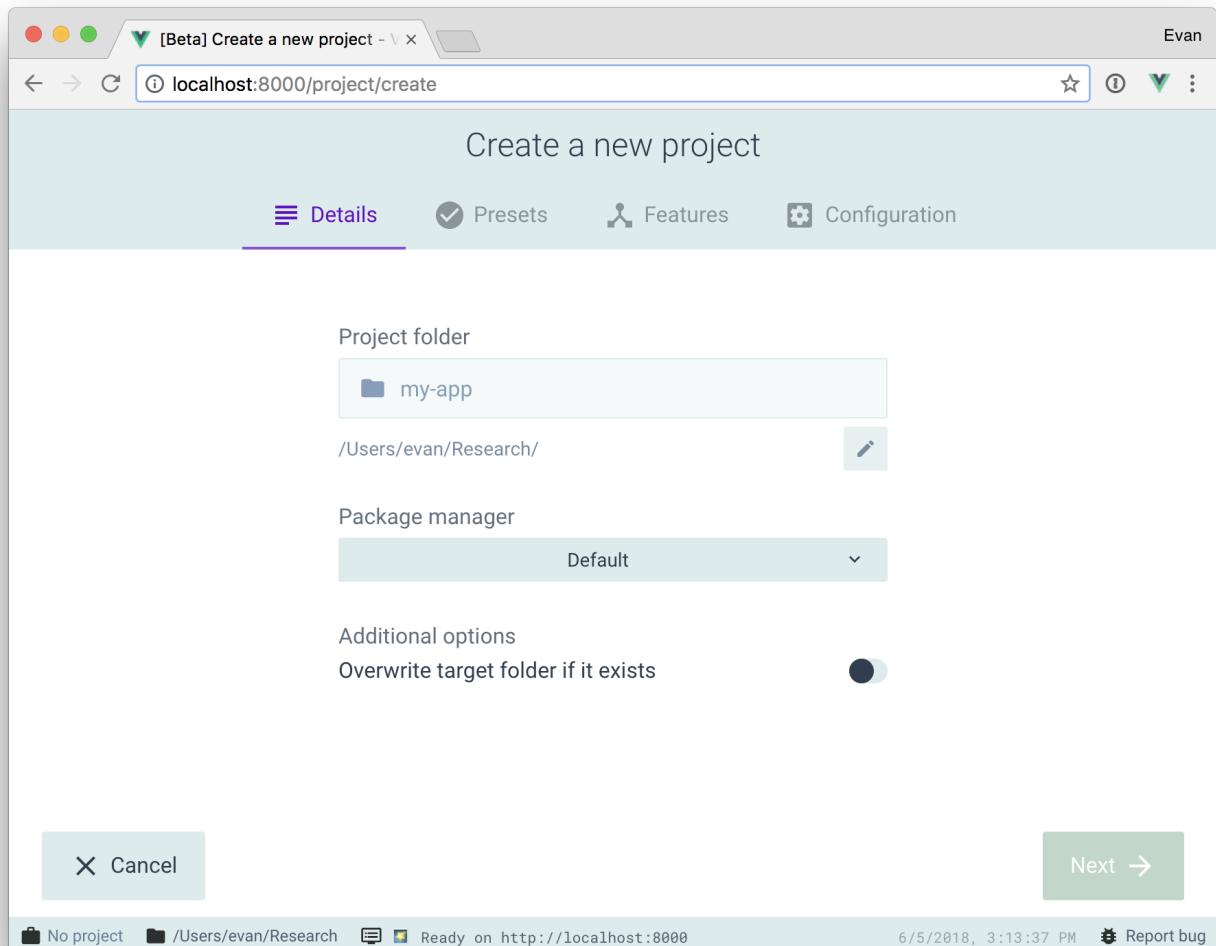
A saisir dans une invite de commande.

Installation de Vue CLI

```
npm install -g @vue/cli  
#OU  
yarn global add @vue/cli
```

Créer et gérer un projet avec vue-cli.

```
vue create my-project  
#0U  
vue ui
```



La base d'une application : l'instance Vue.

Chaque application Vue est initialisée en créant une nouvelle **instance de Vue** avec la fonction **Vue** :

```
var vm = new Vue({  
  // options  
})
```

Bien que n'étant pas strictement associée au patron d'architecture [MVVM pattern](#), la conception de Vue s'en est en partie inspirée.

Par convention, nous utilisons souvent la variable **vm** (abréviation pour **ViewModel**) pour faire référence à nos instances de Vue.

Quand vous créez une instance de Vue, vous devez passer un **objet d'options** pouvant être utilisées pour créer les comportements que vous souhaitez.

Vous pouvez parcourir la liste complète de référence [dans la documentation de l'API](#).

Une application Vue consiste en une **instance racine de Vue** créée avec `new Vue` et optionnellement organisée en un arbre de composants imbriqués et réutilisables.

```
Instance racine  
└ TodoList  
  └ TodoItem  
    └ DeleteTodoButton  
    └ EditTodoButton  
  └ TodoListFooter  
    └ ClearTodosButton  
    └ TodoListStatistics
```

Tous les composants de Vue sont également des instances de Vue et qu'ils acceptent donc le même objet d'option (à l'exception de quelques options spécifiques à l'instance racine).

Données et méthodes

Quand une instance de Vue est créée, cela ajoute toutes les propriétés trouvées dans son objet **data** au **système réactif** de Vue.

Quand une valeur de ces propriétés change, la vue va « réagir », se mettant à jour pour concorder avec les nouvelles valeurs.

```
// Notre objet de données
var data = { a: 1 }

// L'objet est ajouté à une instance de Vue
var vm = new Vue({
  data: data
})

// Récupérer la propriété depuis l'instance
// retourne celle des données originales
vm.a == data.a // => true

// assigner la propriété à une instance
// affecte également la donnée originale
vm.a = 2
data.a // => 2

// ... et vice-versa
data.a = 3
vm.a // => 3
```

Quand ces données changent, le rendu de la vue est refait.

Il est à noter que les propriétés dans **data** sont **réactives** si elles existaient quand l'instance a été créée.

Cela signifie que si vous ajoutez une nouvelle propriété ainsi :

```
vm.b = 'salut'
```

Les changements de **b** ne déclencheront aucune mise à jour.

Si vous savez que vous allez avoir besoin d'une propriété plus tard qui n'a pas de valeur dès le début, vous avez juste besoin de la créer avec n'importe quelle valeur initiale. Par exemple :

```
data: {
  newTodoText: '',
  visitCount: 0,
  hideCompletedTodos: false,
  todos: [],
  error: null
}
```


La seule exception à cela est l'utilisation de `Object.freeze()`, qui empêche les propriétés existantes d'être changées, ce qui implique que le système de réactivité ne peut pas traquer les changements.

```
var obj = {
  foo: 'bar'
}

Object.freeze(obj)

new Vue({
  el: '#app',
  data: obj
})
```

```
<div id="app">
  <p>{{ foo }}</p>
  <!-- cela ne va plus mettre à jour `foo` ! -->
  <button v-on:click="foo = 'baz'">Change le</button>
</div>
```

En plus des propriétés de données, les instances de Vue exposent de nombreuses méthodes et propriétés utiles.

Ces propriétés et méthodes sont préfixées par `$` pour les différencier des propriétés proxifiées de data.

Par exemple :

```
var data = { a: 1 }
var vm = new Vue({
  el: '#example',
  data: data
})

vm.$data === data // => true
vm.$el === document.getElementById('example') // => true

// $watch est une méthode de l'instance
vm.$watch('a', function (newVal, oldVal) {
  // cette fonction de rappel sera appelée quand `vm.a` changera
})
```

Consultez l'API (en Annexe) pour une liste complète des propriétés et méthodes d'une instance.

Créer son premier composant Vue.js.

Voici un exemple de composant Vue :

```
// Définition d'un nouveau composant appelé `button-counter`  
Vue.component('button-counter', {  
  data: function () {  
    return {  
      count: 0  
    }  
  },  
  template: '<button v-on:click="count++">Vous \'avez cliqué {{ count }}  
fois.</button>'  
})
```

Les composants sont des instances de Vue réutilisables avec un **nom** : dans notre cas `<button-counter>`.

Nous pouvons utiliser ce composant en tant qu'élément personnalisé à l'intérieur d'une instance de Vue racine créée avec `new Vue` :

```
<div id="components-demo" class="demo">  
  <button-counter></button-counter>  
</div>  
<script>  
Vue.component('button-counter', {  
  data: function () {  
    return {  
      count: 0  
    }  
  },  
  template: '<button v-on:click="count += 1">Vous \'avez cliqué {{ count }}  
fois.</button>'  
})  
new Vue({ el: '#components-demo' })  
</script>
```

Puisque les composants sont des instances de Vue réutilisables, ils acceptent les mêmes options que `new Vue` comme `data`, `computed`, `watch`, `methods`, et les hooks du cycle de vie. Les seules exceptions sont quelques options spécifiques à la racine comme `el`.

Réutilisation de composants

Les composants peuvent être réutilisés autant de fois que souhaité :

```
<div id="components-demo">
  <button-counter></button-counter>
  <button-counter></button-counter>
  <button-counter></button-counter>
</div>
```

```
<div id="components-demo2" class="demo">
  <button-counter></button-counter>
  <button-counter></button-counter>
  <button-counter></button-counter>
</div>
<script>
new Vue({ el: '#components-demo2' })
</script>
```

Notez que lors du clic sur les boutons, chacun d'entre eux maintient son propre compteur séparé des autres.

Chaque fois que vous utilisez un composant, une nouvelle **instance** est créée.

data doit être une fonction

Quand vous définissez le composant `<button-counter>`, vous devez faire attention que **data** ne soit pas directement fourni en tant qu'objet, comme ceci :

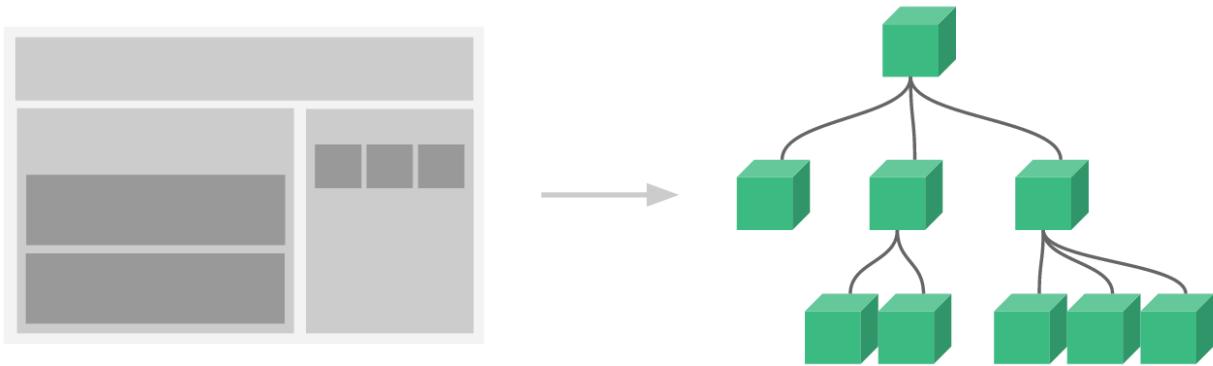
```
data: {
  count: 0
}
```

À la place, **la propriété du composant data doit être une fonction**, afin que chaque instance puisse conserver une copie indépendante de l'objet retourné :

```
data: function () {
  return {
    count: 0
  }
}
```

Organisation des composants

Il est commun pour une application d'être organisée en un arbre de composants imbriqués :



Par exemple, vous pouvez avoir des composants pour l'entête, la barre latérale, la zone de contenu ; chacun contenant lui aussi d'autres composants pour la navigation, les liens, les billets de blog, etc.

Pour utiliser ces composants dans des templates, ils doivent être enregistrés pour que Vue les connaisse.

Il y a deux types d'enregistrement de composant : **global** et **local**. Jusqu'ici, nous avons uniquement enregistré des composants globalement en utilisant `Vue.component` :

```
Vue.component('my-component-name', {  
  // ... options ...  
})
```

Les composants enregistrés globalement peuvent être utilisés dans le template de n'importe quelle instance racine de Vue (`new Vue`) créée après coup, ainsi que dans les sous-composants de l'arbre des composants de cette instance de Vue.

Passer des données aux composants enfants avec les props

Les **props** sont des attributs personnalisables que vous pouvez enregistrer dans un composant.

Quand une valeur est passée à un attribut prop, elle devient une propriété de l'instance du composant.

```
Vue.component('blog-post', {
  props: ['title'],
  template: '<h3>{{ title }}</h3>'
})
```

Un composant peut avoir autant de props que vous le souhaitez et par défaut, n'importe quelle valeur peut être passée à une prop.

Dans le template ci-dessus, vous devriez voir cette valeur dans l'instance du composant, comme pour **data**.

Une fois une prop enregistrée, vous pouvez lui passer des données en tant qu'attribut personnalisé comme ceci :

```
<div id="blog-post-demo" class="demo">
  <blog-post1 title="Mon initiation avec Vue"></blog-post1>
  <blog-post1 title="Blogger avec Vue"></blog-post1>
  <blog-post1 title="Pourquoi Vue est tellement cool"></blog-post1>
</div>
<script>
Vue.component('blog-post1', {
  props: ['title'],
  template: '<h3>{{ title }}</h3>'
})
new Vue({ el: '#blog-post-demo' })
</script>
```

Dans une application typique, cependant, vous préféreriez avoir un tableau de billets dans **data** :

```
new Vue({
  el: '#blog-post-demo',
  data: {
    posts: [
      { id: 1, title: 'Mon initiation avec Vue' },
      { id: 2, title: 'Blogger avec Vue' },
      { id: 3, title: 'Pourquoi Vue est tellement cool' }
    ]
  }
})
```

Maintenant, faisons le rendu d'un composant pour chacun :

```
<blog-post
  v-for="post in posts"
  v-bind:key="post.id"
  v-bind:title="post.title"
></blog-post>
```

Vous voyez au-dessus que nous pouvons utiliser **v-bind** pour dynamiquement passer des props. Cela est particulièrement utile quand vous ne connaissez pas exactement le contenu dont vous êtes en train de faire le rendu à l'avance.

Élément racine unique

Quand nous réalisons un composant `<blog-post>`, votre template va éventuellement contenir plus que juste le titre :

```
<h3>{{ title }}</h3>
```

Vous allez au moins vouloir inclure le contenu du billet :

```
<h3>{{ title }}</h3>
<div v-html="content"></div>
```

Si vous essayez cela dans votre template cependant, Vue va afficher une erreur, expliquant que **tout composant doit avoir un unique élément racine**. Vous pouvez corriger cette erreur en imbriquant le template dans un élément parent comme :

```
<div class="blog-post">
  <h3>{{ title }}</h3>
  <div v-html="content"></div>
</div>
```

À mesure que nos composants grandissent, il ne sera plus question uniquement d'un titre et d'un contenu pour le billet, mais également de la date de publication, des commentaires et bien plus. Définir une prop indépendamment pour chaque information pourrait devenir gênant :

```
<blog-post
  v-for="post in posts"
  v-bind:key="post.id"
  v-bind:title="post.title"
  v-bind:content="post.content"
  v-bind:publishedAt="post.publishedAt"
  v-bind:comments="post.comments"
></blog-post>
```

Le temps sera alors venu de refactoriser le composant `<blog-post>` pour accepter une propriété `post` unique à la place :

```
<blog-post
  v-for="post in posts"
  v-bind:key="post.id"
  v-bind:post="post"
></blog-post>
```

```

Vue.component('blog-post', {
  props: ['post'],
  template: `
    <div class="blog-post">
      <h3>{{ post.title }}</h3>
      <div v-html="post.content"></div>
    </div>
  `
})

```

Maintenant, chaque fois qu'une nouvelle propriété sera ajoutée à l'objet `post`, elle sera automatiquement disponible dans `<blog-post>`.

Envoyer des messages aux parents avec les évènements

Maintenant, ajoutons un bouton pour élargir le texte :

```

Vue.component('blog-post', {
  props: ['post'],
  template: `
    <div class="blog-post">
      <h3>{{ post.title }}</h3>
      <button>
        Enlarge text
      </button>
      <div v-html="post.content"></div>
    </div>
  `
})

```

Le parent peut choisir d'écouter n'importe quel évènement de l'instance du composant enfant avec `v-on`, juste comme il le ferait avec un évènement natif du DOM :

```

<blog-post
  ...
  v-on:enlarge-text="postFontSize += 0.1"
></blog-post>

```

Puis le composant enfant peut émettre un évènement lui-même en appelant la méthode préconçue `$emit`, en lui passant le nom de l'évènement :

```

<button v-on:click="$emit('enlarge-text')">
  Élargir le texte
</button>

```

Grace à l'écouteur `v-on:enlarge-text="postFontSize += 0.1"`, le parent va recevoir l'évènement et mettre à jour la valeur de `postFontSize`.

```
<div id="blog-posts-events-demo" class="demo">
  <div :style="{ fontSize: postFontSize + 'em' }">
    <blog-post
      v-for="post in posts"
      v-bind:key="post.id"
      v-bind:post="post"
      v-on:enlarge-text="postFontSize += 0.1"
    ></blog-post>
  </div>
</div>
<script>
Vue.component('blog-post', {
  props: ['post'],
  template: `\
    <div class="blog-post">\
      <h3>{{ post.title }}</h3>\
      <button v-on:click="$emit('enlarge-text')">\
        Élargir le texte\
      </button>\
      <div v-html="post.content"></div>\
    </div>`\
  })
new Vue({
  el: '#blog-posts-events-demo',
  data: {
    posts: [
      { id: 1, title: 'Mon initiation avec Vue', content: '...content...' },
      { id: 2, title: 'Blogger avec Vue', content: '...content...' },
      { id: 3, title: 'Pourquoi Vue est tellement cool', content: '...content...' }
    ],
    postFontSize: 1
  }
})
</script>
```

Émettre une valeur avec un évènement

Il est parfois utile d'émettre une valeur spécifique avec un évènement. Dans ce cas, nous pouvons utiliser `$emit` en second paramètre pour fournir cette valeur :

```
<button v-on:click="$emit('enlarge-text', 0.1)">
    Élargir le texte
</button>
```

Puis quand nous écoutons l'évènement dans le parent, nous pouvons accéder à la valeur de l'évènement émis avec `$event` :

```
<blog-post
  ...
  v-on:enlarge-text="postFontSize += $event"
></blog-post>
```

Ou, si le gestionnaire d'évènement est une méthode :

```
<blog-post
  ...
  v-on:enlarge-text="onEnlargeText"
></blog-post>
```

Puis la valeur sera fournie en tant que premier argument de cette méthode :

```
methods: {
  onEnlargeText: function (enlargeAmount) {
    this.postFontSize += enlargeAmount
  }
}
```

Utiliser `v-model` sur les composants

Les évènements personnalisés peuvent aussi être utilisés pour créer des champs qui fonctionnent avec `v-model`. Rappelez-vous cela :

```
<input v-model="searchText">
```

réalise la même chose que :

```
<input  
  v-bind:value="searchText"  
  v-on:input="searchText = $event.target.value"  
>
```

Quand il est utilisé sur un composant, `v-model` fait plutôt cela :

```
<custom-input  
  v-bind:value="searchText"  
  v-on:input="searchText = $event"  
></custom-input>
```

Pour que cela puisse fonctionner, la balise `<input>` à l'intérieur du composant doit :

- Lier l'attribut `value` à la prop `value`
- Et sur l'`input`, émettre son propre évènement personnalisé `input` avec la nouvelle valeur

Voici un exemple en action :

```
Vue.component('custom-input', {  
  props: ['value'],  
  template: `<input  
    v-bind:value="value"  
    v-on:input="$emit('input', $event.target.value)"  
  >`  
})
```

Maintenant `v-model` fonctionnera parfaitement avec le composant :

```
<custom-input v-model="searchText"></custom-input>
```

C'est tout ce que vous avez besoin de savoir à propos des évènements pour le moment, mais une fois que vous aurez fini de lire cette page et que vous vous sentirez à l'aise avec son contenu, nous vous recommandons de revenir pour lire le guide complet à propos des [événements personnalisés](#).

Distribution de contenu avec les slots

Exactement comme les éléments HTML, il est souvent utile de passer du contenu à un composant comme ceci :

```
<alert-box>
  Quelque chose s'est mal passé.
</alert-box>
```

Qui pourrait faire le rendu de quelque chose comme :

```
<div id="slots-demo" class="demo">
  <alert-box>
    Quelque chose s'est mal passé.
  </alert-box>
</div>
<script>
Vue.component('alert-box', {
  template: `\
    <div class="demo-alert-box"> \
      <strong>Erreur !</strong> \
      <slot></slot> \
    </div> \
  `
})
new Vue({ el: '#slots-demo' })
</script>
<style>
.demo-alert-box {
  padding: 10px 20px;
  background: #f3beb8;
  border: 1px solid #f09898;
}
</style>
```

Heureusement, cette tâche est vraiment simple avec l'élément personnalisé `<slot>` de Vue :

```
Vue.component('alert-box', {
  template: `
    <div class="demo-alert-box">
      <strong>Erreur !</strong>
      <slot></slot>
    </div>
  `
})
```

Comme vous pouvez le constater plus haut, nous avons seulement ajouté un slot là où nous souhaitions faire atterrir le contenu - et c'est tout. C'est fait !

Composants dynamiques

Parfois, il est utile de dynamiquement interchanger des composants, comme dans une interface à onglet :

```
<div id="dynamic-component-demo" class="demo">
  <button
    v-for="tab in tabs"
    v-bind:key="tab"
    class="dynamic-component-demo-tab-button"
    v-bind:class="{ 'dynamic-component-demo-tab-button-active': tab ===
currentTab }"
    v-on:click="currentTab = tab"
  >
    {{ tab }}
  </button>
  <component
    v-bind:is="currentTabComponent"
    class="dynamic-component-demo-tab"
  ></component>
</div>
<script>
Vue.component('tab-home', { template: '<div>Home component</div>' })
Vue.component('tab-posts', { template: '<div>Posts component</div>' })
Vue.component('tab-archive', { template: '<div>Archive component</div>' })
new Vue({
  el: '#dynamic-component-demo',
  data: {
    currentTab: 'Home',
    tabs: ['Home', 'Posts', 'Archive']
  },
  computed: {
    currentTabComponent: function () {
      return 'tab-' + this.currentTab.toLowerCase()
    }
  }
})
</script>
```

```

<style>
.dynamic-component-demo-tab-button {
  padding: 6px 10px;
  border-top-left-radius: 3px;
  border-top-right-radius: 3px;
  border: 1px solid #ccc;
  cursor: pointer;
  background: #f0f0f0;
  margin-bottom: -1px;
  margin-right: -1px;
}
.dynamic-component-demo-tab-button:hover {
  background: #e0e0e0;
}
.dynamic-component-demo-tab-button-active {
  background: #e0e0e0;
}
.dynamic-component-demo-tab {
  border: 1px solid #ccc;
  padding: 10px;
}
</style>

```

Ce qu'il y a ci-dessus est rendu possible grâce à l'élément `<component>` de Vue avec l'attribut spécial `is` :

```

<!-- Component changes when currentTabComponent changes -->
<component v-bind:is="currentTabComponent"></component>

```

Dans l'exemple ci-dessus, `currentTabComponent` peut contenir soit :

- le nom du composant enregistré, ou
- un objet d'option de composant

Regardez [ce fiddle](#) pour expérimenter cela avec un code complet, ou [cette version](#) pour un exemple lié à un objet d'option de composant plutôt qu'à un nom enregistré.

Gérer les événements du DOM.

Écouter des évènements

Nous pouvons utiliser l'instruction `v-on` pour écouter les évènements du DOM afin d'exécuter du JavaScript lorsque ces évènements surviennent.

```
<div id="example-1" class="demo">
  <button v-on:click="counter += 1">Add 1</button>
  <p>Le bouton ci-dessus a été cliqué {{ counter }} fois.</p>
</div>
<script>
var example1 = new Vue({
  el: '#example-1',
  data: {
    counter: 0
  }
})
</script>
```

Méthodes des gestionnaires d'évènements

La logique avec beaucoup de gestionnaires d'évènements sera très certainement plus complexe, `v-on` peut aussi accepter le nom d'une méthode que vous voudriez appeler.

```
<div id="example-2" class="demo">
  <button v-on:click="greet">Dire bonjour</button>
</div>
<script>
var example2 = new Vue({
  el: '#example-2',
  data: {
    name: 'Vue.js'
  },
  methods: {
    greet: function (event) {
      alert('Bonjour ' + this.name + ' !')
      if (event) {
        alert(event.target.tagName)
      }
    }
  }
})
</script>
```

Méthodes appelées dans les valeurs d'attributs

Au lieu de lier directement la méthode par son nom dans l'attribut, nous pouvons également utiliser des méthodes dans une instruction JavaScript :

```

<div id="example-3" class="demo">
  <button v-on:click="say('salut')">Dire salut</button>
  <button v-on:click="say('quoi')">Dire quoi</button>
</div>
<script>
new Vue({
  el: '#example-3',
  methods: {
    say: function (message) {
      alert(message)
    }
  }
})
</script>

```

Pour accéder à l'évènement original du DOM depuis l'instruction dans l'attribut.

Vous pouvez le passer à une méthode en utilisant la variable spéciale `$event` :

```

<button v-on:click="warn('Le formulaire ne peut être soumis pour le
moment.', $event)">
  Soumettre
</button>

```

```

// ...
methods: {
  warn: function (message, event) {
    // maintenant nous avons accès à l'évènement natif
    if (event) event.preventDefault()
    alert(message)
  }
}

```

Modificateurs d'évènements

C'est un besoin courant que de faire appel à `event.preventDefault()` ou `event.stopPropagation()` à l'intérieur d'une déclaration de gestionnaire d'évènements.

Bien que nous puissions réaliser ceci aisément à l'intérieur des méthodes, il serait préférable que les méthodes restent purement dédiées à la logique des données au lieu d'avoir à gérer les détails des évènements du DOM.

Pour résoudre ce problème, Vue propose des modificateurs d'évènements pour `v-on`.

- `.stop`
- `.prevent`
- `.capture`
- `.self`
- `.once`
- `.passive`

```
<!-- la propagation de l'évènement `click` sera stoppée -->
<a v-on:click.stop="doThis"></a>

<!-- l'évènement `submit` ne rechargera plus la page -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- les modificateurs peuvent être chainés -->
<a v-on:click.stop.prevent="doThat"></a>

<!-- seulement le modificateur -->
<form v-on:submit.prevent></form>

<!-- utilise le mode « capture » lorsque l'écouteur d'évènements est
ajouté -->
<!-- c.-à-d. qu'un évènement destiné à un élément interne est géré ici
avant d'être géré par ses éléments parents -->
<div v-on:click.capture="doThis">...</div>

<!-- seulement déclenché si l'instruction `event.target` est l'élément
lui-même -->
<!-- c.-à-d. que cela ne s'applique pas aux éléments enfants -->
<div v-on:click.self="doThat">...</div>

<!-- l'évènement « click » sera déclenché au plus une fois -->
<a v-on:click.once="doThis"></a>
```

Contrairement aux autres modificateurs, qui sont exclusifs aux évènements natifs du DOM, le modificateur `.once` peut également être utilisé pour les [événements des composants](#).

Nouveau en 2.3.0+

Vue offre également un modificateur `.passive` correspondant à l'[option passive de addEventListener](#).

```
<!-- le comportement par défaut de l'évènement de défilement se produit immédiatement, -->
<!-- au lieu d'attendre que `onScroll` soit terminé -->
<!-- au cas où il contienne `event.preventDefault()` -->
<div v-on:scroll.passive="onScroll">...</div>
```

Le modificateur `.passive` est particulièrement pratique pour améliorer les performances sur mobile.

N'utilisez pas `.passive` et `.prevent` ensemble. `.passive` sera ignoré et votre navigateur va probablement vous montrer un message.

Modificateurs de code des touches

Vue permet également d'ajouter un modificateur de touches pour `v-on`:

```
<!-- faire appel à `vm.submit()` uniquement quand le code de la touche est  
`Enter` -->  
<input v-on:keyup.enter="submit">
```

Vous pouvez également utiliser n'importe quel nom de touche clavier valide fourni par `KeyboardEvent.key` en tant que modificateur en les écrivant au format kebab-case :

```
<input @keyup.page-down="onPageDown">
```

Dans l'exemple ci-dessus, le gestionnaire va uniquement être appelé si `$event.key` est égale à 'PageDown'.

Code des touches

Utilisez des attributs `keyCode` est également possible :

```
<input v-on:keyup.13="submit">
```

Vue fournit des alias pour la plupart des clés communes utilisés quand nécessaire pour le support des anciens navigateurs :

- `.enter`
- `.tab`
- `.delete` (fonctionne pour les touches « Suppression » et « Retour arrière »)
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`

Vous pouvez également définir des raccourcis personnalisés pour vos modificateurs grâce à l'objet global `config.keyCode` :

```
// active `v-on:keyup.f1`  
Vue.config.keyCode.f1 = 112
```

Modificateurs de touches système

Nouveau dans la 2.1.0+

Vous pouvez utiliser les modificateurs suivants pour déclencher un évènement du clavier ou de la souris seulement lorsque la touche du modificateur correspondante est pressée :

- .ctrl
- .alt
- .shift
- .meta

Note: Sur les claviers Macintosh, meta est la touche commande (⌘). Sur Windows, meta est la touche windows (⊞). Sur les claviers Sun Microsystems, meta est symbolisée par un diamant plein (◆). Sur certains claviers, spécifiquement sur les claviers des machines MIT et Lisp et leurs successeurs, comme le clavier « Knight » et « space-cadet », meta est écrit « META ». Sur les claviers Symboliques, meta est étiqueté « META » ou « Meta ».

Par exemple :

```
<!-- Alt + C -->
<input @keyup.alt.67="clear">

<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

Modificateur .exact

Nouveau dans la 2.5.0+

Le modificateur .exact permet le contrôle de la combinaison de touches système exacte requise pour déclencher le gestionnaire d'évènements.

```
<!-- ceci va aussi émettre un évènement si les touches Alt et Shift sont
pressées -->
<button @click.ctrl="onClick">A</button>

<!-- ceci va émettre un évènement seulement si la touche Ctrl est pressée
sans aucune autre touche -->
<button @click.ctrl.exact="onCtrlClick">A</button>

<!-- ceci va émettre un évènement si aucune touche n'est pressée -->
<button @click.exact="onClick">A</button>
```

Modificateurs de boutons de la souris

Nouveau dans la 2.2.0+

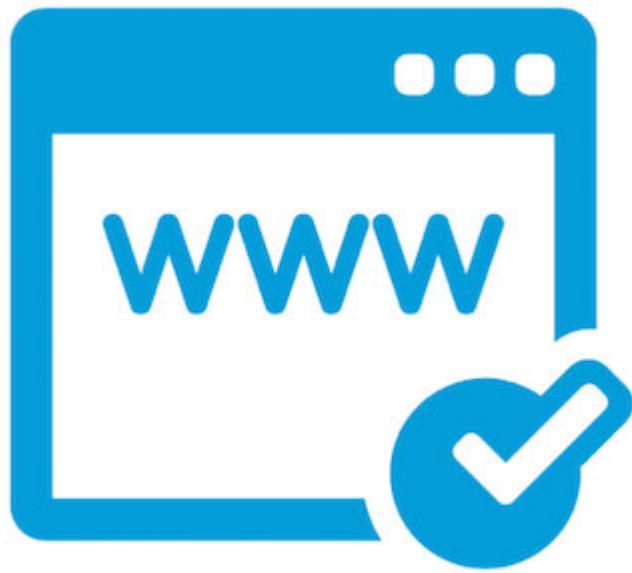
- .left

- `.right`
- `.middle`

Ces modificateurs n'autorisent la gestion de l'évènement que s'il a été déclenché par un bouton spécifique de la souris.

Débugger avec vue-devtools.

Quand vous utilisez Vue, nous vous recommandons également d'installer [Vue Devtools](#) dans votre navigateur. Cela vous permettra d'inspecter et de déboguer vos applications Vue dans une interface dédiée et intuitive.



Organiser le code en composants.

Organiser le code en composants.

Noms de composant

Lors de la création de composants, il faudra toujours spécifier un nom. Par exemple, la déclaration se fera comme suit :

```
Vue.component('my-component-name', { /* ... */ })
```

Le nom du composant est le premier argument de `Vue.component`.

Le nom que vous donnez à un composant peut dépendre de l'endroit où vous avez l'intention de l'utiliser.

Vous pouvez voir les recommandations pour les noms de composants dans le guide des [Conventions](#).

Casse des noms

Vous avez deux options pour définir vos noms de composant :

En kebab-case

```
Vue.component('my-component-name', { /* ... */ })
```

Lors de la définition d'un composant en kebab-case, vous devez également utiliser kebab-case lors du référencement de l'élément, comme ceci `<my-component-name>`.

En PascalCase

```
Vue.component('MyComponentName', { /* ... */ })
```

Lors de la définition d'un composant en PascalCase, vous pouvez utiliser l'un ou l'autre cas lors du référencement de l'élément. Cela signifie que `<my-component-name>` et `<MyComponentName>` sont acceptables. À noter, cependant, que seuls les noms en kebab-case sont directement valides dans le DOM (c.-à-d. la forme non-chaine).

Création globale

Les composants créés avec `Vue.component` :

```
Vue.component('my-component-name', {  
  // ... options ...  
})
```

Sont des composants **enregistrés globalement**. Cela signifie qu'ils peuvent être utilisés dans le template de n'importe quelle instance Vue (`new Vue`) créée ultérieurement.

Par exemple :

```
Vue.component('component-a', { /* ... */ })  
Vue.component('component-b', { /* ... */ })  
Vue.component('component-c', { /* ... */ })  
  
new Vue({ el: '#app' })
```

```
<div id="app">  
  <component-a></component-a>  
  <component-b></component-b>  
  <component-c></component-c>  
</div>
```

Cela s'applique même à tous les sous-composants, ce qui signifie que ces trois composants seront également disponibles *les uns des autres*.

Création locale

Vous pouvez définir vos composants en tant qu'objets JavaScript simples :

```
var ComponentA = { /* ... */ }
var ComponentB = { /* ... */ }
var ComponentC = { /* ... */ }
```

Puis utiliser les composants que vous souhaitez utiliser dans l'option `components` :

```
new Vue({
  el: '#app',
  components: {
    'component-a': ComponentA,
    'component-b': ComponentB
  }
})
```

Pour chaque propriété de l'objet `components`, la clé sera le nom de l'élément personnalisé, tandis que la valeur contiendra l'objet d'options du composant.

Notez que **les composants créés localement ne sont pas disponibles dans les sous-composants**. Par exemple, si vous voulez que `ComponentA` soit disponible dans `ComponentB`, vous devez utiliser :

```
var ComponentA = { /* ... */ }

var ComponentB = {
  components: {
    'component-a': ComponentA
  },
  // ...
}
```

Ou si vous utilisez des modules ES2015, tels que Babel et webpack, cela pourrait plus ressembler à :

```
import ComponentA from './ComponentA.vue'

export default {
  components: {
    ComponentA
  },
  // ...
}
```

Notez que dans ES2015+, placer un nom de variable comme **ComponentA** dans un objet est un raccourci à **ComponentA: ComponentA**, signifiant que le nom de la variable est à la fois :

- le nom de l'élément personnalisé à utiliser dans le template et
- le nom de la variable contenant les options du composant

Systèmes de module

Si vous n'utilisez pas un système de module avec `import/require`, vous pouvez probablement ignorer cette section pour l'instant. Si oui, nous avons quelques instructions spéciales et conseils pour vous.

Création locale dans un système de module

Si vous êtes toujours là, il est probable que vous utilisez un système de modules, comme avec Babel et webpack. Dans ces cas, nous recommandons de créer un répertoire `components`, avec chaque composant dans son propre fichier.

Ensuite, vous devrez importer chaque composant que vous souhaitez utiliser avant de l'enregistrer localement. Par exemple, dans un fichier hypothétique `ComponentB.js` ou `ComponentB.vue` :

```
import ComponentA from './ComponentA'  
import ComponentC from './ComponentC'  
  
export default {  
  components: {  
    ComponentA,  
    ComponentC  
  },  
  // ...  
}
```

Maintenant, et `ComponentA` et `ComponentC` peuvent être utilisés dans le template du composant `ComponentB`.

Enregistrement global automatique des composants de base

La plupart de vos composants seront relativement génériques, ce qui ne fera qu'englober un élément comme un champ ou un bouton et ils ont tendance à être utilisés très fréquemment à travers vos composants.

Le résultat est que de nombreux composants peuvent inclure de longues listes de composants de base :

```
import BaseButton from './BaseButton.vue'  
import BaseIcon from './BaseIcon.vue'  
import BaseInput from './BaseInput.vue'  
  
export default {  
  components: {  
    BaseButton,  
    BaseIcon,  
    BaseInput  
  }  
}
```

Juste pour supporter relativement peu de balise dans un template :

```

<BaseInput
  v-model="searchText"
  @keydown.enter="search"
/>
<BaseButton @click="search">
  <BaseIcon name="search"/>
</BaseButton>

```

Heureusement, si vous utilisez webpack (ou [Vue CLI 3+](#), qui utilise webpack en interne), vous pouvez utiliser `require.context` pour enregistrer globalement précisément ces composants de base très courants. Voici un exemple de code que vous pouvez utiliser pour importer globalement des composants de base dans le fichier d'entrée de votre application. (ex. `src/main.js`) :

```

import Vue from 'vue'
import upperFirst from 'lodash/upperFirst'
import camelCase from 'lodash/camelCase'

const requireComponent = require.context(
  // Le chemin relatif du dossier composants
  './components',
  // Suivre ou non les sous-dossiers
  false,
  // L'expression régulière utilisée pour faire concorder les noms de
  // fichiers de composant de base
  /Base[A-Z]\w+\.(vue|js)$/
)

requireComponent.keys().forEach(fileName => {
  // Récupérer la configuration du composant
  const componentConfig = requireComponent(fileName)

  // Récupérer le nom du composant en PascalCase
  const componentName = upperFirst(
    camelCase(
      // Retrouver le nom du fichier indépendamment de la profondeur de
      // dossier
      fileName
        .split('/')
        .pop()
        .replace(/\.\w+$/, '')
    )
  )

  // Créer un composant global
  Vue.component(
    componentName,
    // Chercher les options du composant dans `default`, qui
    // existera si le composant a été exporté avec `export default`,
    // sinon revenez à la racine du module.
    componentConfig.default || componentConfig
)

```

```
)  
})
```

N'oubliez pas que **la création globale doit avoir lieu avant la création de l'instance racine Vue (avec new Vue)**. Voici un exemple de ce modèle dans un contexte de projet réel.

Arborescence de composants et props.

Casse des props (camelCase vs. kebab-case)

Les noms d'attributs HTML sont insensibles à la casse, aussi les navigateurs interpréteront de la même manière les majuscules et les minuscules. Cela signifie que pour l'utilisation des templates directement dans le DOM, les props en camelCase doivent utiliser leur équivalent kebab-case (délimités par des tirets) :

```
Vue.component('blog-post', {  
  // camelCase en JavaScript  
  props: ['postTitle'],  
  template: '<h3>{{ postTitle }}</h3>'  
})
```

```
<!-- kebab-case en HTML -->  
<blog-post post-title="Hello !"></blog-post>
```

Cependant, si vous utilisez les templates de chaîne de caractères directement dans le JavaScript, il n'y a pas cette limitation.

Types des props

Pour l'instant, nous n'avons vu que les props listées dans des tableaux de chaînes de caractères :

```
props: ['title', 'likes', 'isPublished', 'commentIds', 'author']
```

Plus fréquemment, vous souhaiterez que chaque prop possède un type de valeur spécifique. Dans ce cas, vous pouvez lister vos propriétés comme un objet, où les noms et les valeurs de propriétés sont respectivement les noms et types de props :

```
props: {
  title: String,
  likes: Number,
  isPublished: Boolean,
  commentIds: Array,
  author: Object,
  callback: Function,
  contactsPromise: Promise // ou n'importe quel autre constructeur
}
```

Cela ne documente pas seulement votre composant mais va également permettre à l'utilisateur d'être averti par la console JavaScript du navigateur s'il passe le mauvais type.

Passage de props statiques ou dynamiques

Plus tôt, vous avez vu comment passer des props de manière statique comme ceci :

```
<blog-post title="Mon initiation avec Vue"></blog-post>
```

Vous avez également vu qu'il était possible d'affecter des props dynamiquement avec **v-bind** comme ceci :

```
<!-- Affecter dynamiquement la valeur d'une variable -->
<blog-post v-bind:title="post.title"></blog-post>

<!-- Affecter dynamiquement la valeur d'une expression complexe -->
<blog-post
  v-bind:title="post.title + ' par ' + post.author.name"
></blog-post>
```

Dans les deux exemples précédents, nous passons en fait une valeur sous forme de chaîne de caractère mais **tous** les types de valeur peuvent en fait être passées à la prop.

Passer un Number

```
<!-- Même si `42` est statique, nous avons besoin de `v-bind` pour dire à Vue -->
<!-- que c'est une expression JavaScript et non pas une chaîne de caractères. -->
<blog-post v-bind:likes="42"></blog-post>

<!-- Affecter dynamiquement la valeur d'une variable. -->
<blog-post v-bind:likes="post.likes"></blog-post>
```

Passer un Boolean

```
<!-- Inclure une prop sans valeur signifie passer `true`. -->
<blog-post is-published></blog-post>

<!-- Même si `false` est statique, nous avons besoin de `v-bind` pour dire à Vue -->
<!-- que c'est une expression JavaScript et non pas une chaîne de caractères. -->
<blog-post v-bind:is-published="false"></blog-post>

<!-- Affecter dynamiquement la valeur d'une variable. -->
<blog-post v-bind:is-published="post.isPublished"></blog-post>
```

Passer un Array

```
<!-- Même si le tableau est statique, nous avons besoin de `v-bind` pour dire à Vue -->
<!-- que c'est une expression JavaScript et non pas une chaîne de caractères. -->
<blog-post v-bind:comment-ids="[234, 266, 273]"></blog-post>

<!-- Affecter dynamiquement la valeur d'une variable. -->
<blog-post v-bind:comment-ids="post.commentIds"></blog-post>
```

Passer un Object

```
<!-- Même si un objet est statique, nous avons besoin de `v-bind` pour dire à Vue -->
<!-- que c'est une expression JavaScript et non pas une chaîne de caractères. -->
<blog-post
  v-bind:author={
    name: 'Veronica',
    company: 'Veridian Dynamics'
}></blog-post>
```

```
<!-- Affecter dynamiquement la valeur d'une variable. -->
<blog-post v-bind:author="post.author"></blog-post>
```

Passage des propriétés d'un objet

Si vous souhaitez passer toutes les propriétés d'un objet en tant que props, vous devez utiliser **v-bind** sans l'argument (**v-bind** au lieu de **v-bind:prop-name**). Par exemple, avec un objet **post** :

```
post: {
  id: 1,
  title: 'Mon initiation avec Vue'
}
```

le template suivant :

```
<blog-post v-bind="post"></blog-post>
```

sera équivalent à :

```
<blog-post
  v-bind:id="post.id"
  v-bind:title="post.title"
></blog-post>
```

Flux de données unidirectionnel

Toutes les données forment **un flux de donnée descendant unidirectionnel** (« **one-way-down binding** ») entre la propriété enfant et la propriété parente : quand la propriété du parent est mise à jour, cela va mettre à jour celle de l'enfant mais pas l'inverse.

De plus, chaque fois que le composant parent est mis à jour, toutes les props du composant enfant vont être mises à jour avec les dernières valeurs.

Cela signifie que vous ne devriez **pas** essayer de muter une prop depuis l'intérieur d'un composant. Si vous le faites, Vue lancera un avertissement dans la console.

Il y a couramment deux cas où vous seriez tenté de muter une prop :

1. **La prop est seulement utilisée pour passer une valeur initiale ; le composant enfant doit seulement l'utiliser comme donnée de propriété local.** Dans ce cas, le mieux est de définir une propriété locale qui utilise la prop comme valeur initiale :

```
props: ['initialCounter'],
data: function () {
  return {
    counter: this.initialCounter
  }
}
```

2. **La prop est passée dans un format qui demande d'être transformé.** Dans ce cas, le mieux est de définir une propriété calculée utilisant la valeur de la prop :

```
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

Notez que les objets et les tableaux en JavaScript sont passés par référence, ce qui signifie que si la prop est un objet ou un tableau, muter l'objet ou le tableau lui-même dans l'enfant **va** affecter l'état du parent.

Attributs non props

Un attribut non prop est un attribut passé à un composant mais qui n'a pas de prop correspondante de définie.

Bien que définir explicitement les props soit préféré pour passer des informations à un composant enfant, les auteurs de bibliothèques ne peuvent pas deviner dans quel contexte leur composant va être utilisé. C'est pourquoi il est possible qu'un composant puisse accepter des attributs arbitraires qui seront ajoutés sur l'élément racine du composant.

Par exemple, imaginez que vous utilisez un composant tiers `bootstrap-date-input` avec un plugin Bootstrap qui nécessite un `data-date-picker` attribut sur l'`input`. Nous pouvons ajouter cet attribut à notre instance de composant :

```
<bootstrap-date-input data-date-picker="activated"></bootstrap-date-input>
```

et l'attribut `data-date-picker="activated"` sera automatiquement ajouté sur l'élément racine de `bootstrap-date-input`.

Remplacement ou fusion avec des attributs existants

Imaginez que ceci est le template pour `bootstrap-date-input` :

```
<input type="date" class="form-control">
```

Pour spécifier un thème pour notre plugin date picker, nous allons avoir besoin d'ajouter une classe comme ceci :

```
<bootstrap-date-input  
  data-date-picker="activated"  
  class="date-picker-theme-dark"  
></bootstrap-date-input>
```

Dans ce cas, deux valeurs différentes de `class` sont définies :

- `form-control`, qui va être appliquée dans le template du composant,
- `date-picker-theme-dark`, qui va être passé au composant depuis son parent.

Pour la plupart des attributs, la valeur fournie par le parent va remplacer la valeur définie par le composant. Aussi par exemple, passer `type="text"` va remplacer `type="date"` et probablement tout casser !

Les attributs `class` et `style` sont fusionnables, fournissant la valeur finale `form-control date-picker-theme-dark`.

Désactiver l'héritage d'attribut

Si vous **ne voulez pas** que l'élément racine d'un composant hérite de tels attributs, vous pouvez mettre `inheritAttrs: false` dans les options de votre composant. Par exemple :

```
Vue.component('my-component', {
  inheritAttrs: false,
  // ...
})
```

Cela est particulièrement utile avec l'utilisation combinée de la propriété d'instance `$attrs` qui contient les noms et valeurs passés à un composant comme ceci :

```
{
  required: true,
  placeholder: 'Entrez votre nom d\'utilisateur'
}
```

Avec `inheritAttrs: false` et `$attrs`, vous pouvez manuellement décider sur quel élément vous souhaitez déposer les attributs passés:

```
Vue.component('base-input', {
  inheritAttrs: false,
  props: ['label', 'value'],
  template: `
    <label>
      {{ label }}
      <input
        v-bind="$attrs"
        v-bind:value="value"
        v-on:input="$emit('input', $event.target.value)"
      >
    </label>
  `
})
```

Notez que l'option `inheritAttrs: false` n'affecte **pas** les liaisons `style` et `class`.

Ce modèle vous permet d'utiliser des composants de base comme des éléments HTML standard sans avoir à vous soucier de quel élément est actuellement à sa racine :

```
<base-input
  v-model="username"
  required
  placeholder="Entrez votre nom d\'utilisateur"
></base-input>
```

Transmission du contenu: les slots/children.

Les contenus de slot

Vue implémente une API de distribution de contenu utilisant l'élément `<slot>` comme zone de distribution de contenu.

Cela vous permet de composer vos composants ainsi :

```
<navigation-link url="/profile">
  Mon profil
</navigation-link>
```

Dans le template `<navigation-link>`, nous aurons :

```
<a
  v-bind:href="url"
  class="nav-link"
>
  <slot></slot>
</a>
```

Lors du cycle de rendu du composant, l'élément `<slot></slot>` est remplacé par « Mon profil ». Les éléments `<slot>` peuvent contenir n'importe quel code de template, incluant le HTML :

```
<navigation-link url="/profile">
  <!-- Mon icône -->
  <span class="fa fa-user"></span>
  Mon profil
</navigation-link>
```

Ou encore faire appel à d'autres composants :

```
<navigation-link url="/profile">
  <!-- Utilisation d'un composant dédié à l'ajout d'une icône -->
  <font-awesome-icon name="user"></font-awesome-icon>
  Mon profil
</navigation-link>
```

Si `<navigation-link>` ne contient **pas** d'élément `<slot>`, n'importe quel contenu fourni entre la balise d'ouverture et de fermeture sera ignoré.

Portée de compilation

Quand vous voulez utiliser des données à l'intérieur d'un slot, comme ici :

```
<navigation-link url="/profile">  
  Se connecter en tant que {{ user.name }}  
</navigation-link>
```

Le slot a accès à la même propriété d'instance (c.-à-d. la même « portée ») que le reste du template. Le slot n'a **pas** accès à la portée de `<navigation-link>`. Par exemple, essayer d'accéder à `url` ne fonctionnera pas :

```
<navigation-link url="/profile">  
  Cliquer ici vous amènera à : {{ url }}  
  <!--  
  Ici `url` sera `undefined` car le contenu est passé  
  _à l'intérieur de_ <navigation-link>, au lieu d'être défini _entre_ le  
  composant <navigation-link>.  
  -->  
</navigation-link>
```

Souvenez-vous de cette règle :

Tout ce qui est dans le template parent est compilé dans la portée parente, tout ce qui est dans le template enfant est compilé dans la portée enfant.

Contenu par défaut

Il y a des cas où il est utile de spécifier un contenu par défaut pour un slot qui sera rendu uniquement si aucun contenu n'est fourni. Par exemple, dans le composant `<submit-button>` :

```
<button type="submit">
  <slot></slot>
</button>
```

Nous pourrions vouloir que le texte « Envoyer » soit rendu à l'intérieur de `<button>` la plupart du temps.

Pour faire de « Envoyer » le contenu par défaut, nous pouvons le placer à l'intérieur des balises `<slot>` :

```
<button type="submit">
  <slot>Envoyer</slot>
</button>
```

Les slots nommés

Dans certains cas, il peut être intéressant d'avoir plusieurs éléments `<slot>`. Dans un exemple hypothétique, voici le template d'un composant `<base-layout>` :

```
<div class="container">
  <header>
    <!-- Ici le contenu de l'entête -->
  </header>
  <main>
    <!-- Ici le contenu courant -->
  </main>
  <footer>
    <!-- Ici le pied de page -->
  </footer>
</div>
```

Dans le cas suivant, l'élément `<slot>` à un attribut spécial `name`, qui peut être utilisé pour désigner des slots additionnels :

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
```

```
</footer>  
</div>
```

Un `<slot>` sans `name` obtient implicitement le nom "default".

Pour fournir du contenu à des slots nommés, nous pouvons utiliser la directive `v-slot` sur un `<template>`, fournissant le nom du slot en tant qu'argument de `v-slot` :

```
<base-layout>  
  <template v-slot:header>  
    <h1>Le titre de ma page</h1>  
  </template>  
  
  <p>Un paragraphe pour le slot par défaut.</p>  
  <p>Un autre paragraphe.</p>  
  
  <template v-slot:footer>  
    <p>Ici les infos de contact</p>  
  </template>  
</base-layout>
```

Maintenant, tout à l'intérieur des éléments `<template>` sera passé aux slots correspondants. Tout contenu non inclus dans un `<template>` utilisant `v-slot` est considéré comme étant destiné au slot par défaut `default`.

Cependant, vous pouvez toujours entourer le contenu du slot par défaut dans un `<template>` si vous désirez que cela soit plus explicite :

```
<base-layout>  
  <template v-slot:header>  
    <h1>Le titre de ma page</h1>  
  </template>  
  
  <template v-slot:default>  
    <p>Un paragraphe pour le slot par défaut.</p>  
    <p>Un autre paragraphe.</p>  
  </template>  
  
  <template v-slot:footer>  
    <p>Ici les infos de contact</p>  
  </template>  
</base-layout>
```

Dans tous les cas, le rendu HTML sera :

```
<div class="container">  
  <header>  
    <h1>Le titre de ma page</h1>  
  </header>
```

```
<main>
  <p>Un paragraphe pour le slot par défaut.</p>
  <p>Un autre paragraphe.</p>
</main>
<footer>
  <p>Ici les infos de contact</p>
</footer>
</div>
```

Notez que **v-slot** ne peut seulement être ajouté à un **<template>** (avec une exception) contrairement aux attributs de **slot** dépréciés.

Slots avec portée

Parfois, il est utile pour les contenus de slot d'avoir accès aux données uniquement disponibles dans le composant enfant. Par exemple, imaginez un composant <current-user> avec le template suivant :

```
<span>
  <slot>{{ user.lastName }}</slot>
</span>
```

Nous souhaiterions remplacer le contenu par défaut pour afficher le nom de famille de l'utilisateur à la place de son prénom comme ceci :

```
<current-user>
  {{ user.firstName }}
</current-user>
```

Ce qui ne fonctionnera pas puisque le composant <current-user> n'a pas accès à **user** ni au contenu que nous avons fourni lors du rendu du parent.

Pour rendre **user** disponible dans le contenu du slot dans le parent, nous pouvons lier **user** comme un attribut de l'élément <slot> :

```
<span>
  <slot v-bind:user="user">
    {{ user.lastName }}
  </slot>
</span>
```

Les attributs liés à l'élément <slot> sont appelés des **props de slot**. Maintenant, dans la portée parente, nous pouvons utiliser **v-slot** avec une valeur pour définir un nom pour les props de slot qui nous ont été fournies :

```
<current-user>
  <template v-slot:default="slotProps">
    {{ slotProps.user.firstName }}
  </template>
</current-user>
```

Dans cet exemple, nous avons choisi de nommer l'objet contenant tous nos props de slot **slotProps** mais vous pouvez choisir n'importe quel nom.

Syntaxe abrégée pour les slots par défaut uniques

Dans les cas comme au-dessus, quand *uniquement* le slot par défaut a un contenu fourni, la balise du composant peut utiliser le template de slot. Cela nous permet d'utiliser `v-slot` directement sur le composant :

```
<current-user v-slot:default="slotProps">
  {{ slotProps.user.firstName }}
</current-user>
```

Cela peut même être raccourci encore plus. Comme un contenu non spécifié est considéré comme étant destiné au slot par défaut, `v-slot` sans un argument est considéré comme faisant référence au slot par défaut :

```
<current-user v-slot="slotProps">
  {{ slotProps.user.firstName }}
</current-user>
```

Notez que la syntaxe abrégée pour le slot par défaut **ne** peut **pas** être mélangée avec les slots nommées, ce qui mènerait à une ambiguïté de portée :

```
<!-- INVALIDE, résultera en un avertissement -->
<current-user v-slot="slotProps">
  {{ slotProps.user.firstName }}
  <template v-slot:other="otherSlotProps">
    `slotProps` n'est PAS disponible ici
  </template>
</current-user>
```

À chaque fois qu'il y a de multiples slots, utilisez la syntaxe complète pour le `<template>` pour **tous** les slots :

```
<current-user>
  <template v-slot:default="slotProps">
    {{ slotProps.user.firstName }}
  </template>

  <template v-slot:other="otherSlotProps">
    ...
  </template>
</current-user>
```

Événements personnalisés.

Noms d'événements

Contrairement aux composants et aux props, les noms d'événements ne fournissent pas de conversion kebab-case/camelCase.

Le nom de l'événement émis doit correspondre exactement au nom utilisé pour écouter cet événement.

```
this.$emit('my-event')
```

```
<!-- Ne fonctionne pas -->
<my-component v-on:myEvent="doSomething"></my-component>
<!-- Fonctionne -->
<my-component v-on:my-event="doSomething"></my-component>
```

Pour ces raisons, nous recommandons de **toujours utiliser la kebab-case pour les noms d'événements.**

Personnaliser le `v-model` du composant

Nouveauté de la 2.2.0+

Par défaut, `v-model` sur un composant utilise `value` comme prop et `input` comme événement, mais certains types de champs input tels que les cases à cocher et les boutons radio peuvent vouloir utiliser l'attribut `value` à d'autres fins.

Utiliser l'option `model` permet d'éviter un conflit dans ce genre de cas :

```
Vue.component('base-checkbox', {
  model: {
    prop: 'checked',
    event: 'change'
  },
  props: {
    checked: Boolean
  },
  template: `
    <input
      type="checkbox"
      v-bind:checked="checked"
      v-on:change="$emit('change', $event.target.checked)"
    >
  `
})
```

À présent, quand vous utilisez `v-model` sur ce composant :

```
<base-checkbox v-model="lovingVue"></base-checkbox>
```

la valeur de `lovingVue` sera passée à la prop `checked`. La propriété `lovingVue` sera ensuite mise à jour quand `<base-checkbox>` émettra un événement `change` avec une nouvelle valeur.

Notez que vous avez toujours besoin de déclarer la prop `checked` dans l'option `props` du composant.

Relier des événements natifs aux composants

Il peut y avoir des fois où vous voudrez écouter directement un événement natif sur l'élément racine d'un composant. Dans ces cas-là, vous pouvez utiliser le modificateur `.native` pour `v-on` :

```
<base-input v-on:focus.native="onFocus"></base-input>
```

Cela peut être utile parfois, mais ce n'est pas une bonne idée quand vous essayez d'écouter l'événement sur un élément bien spécifique tel qu'un `<input>`. Par exemple, le composant `<base-input>` pourrait être revu de façon à ce que l'élément racine soit en fait un élément `<label>` :

```

<label>
  {{ label }}
  <input
    v-bind="$attrs"
    v-bind:value="value"
    v-on:input="$emit('input', $event.target.value)"
  >
</label>

```

Dans ce cas, l'écouteur `.native` dans le parent ne fonctionnera plus et vous ne serez pas prévenus. Il n'y aura pas d'erreurs, mais le gestionnaire d'événement `onFocus` ne sera pas appelé quand vous vous y attendrez.

Pour résoudre ce problème, Vue fournit une propriété `$listeners` contenant un objet avec les écouteurs utilisés sur votre composant. Par exemple :

```

{
  focus: function (event) { /* ... */ }
  input: function (value) { /* ... */ },
}

```

En utilisant la propriété `$listeners`, vous pouvez renvoyer tous les écouteurs d'événements sur le composant vers un élément enfant spécifique avec `v-on="$listeners"`. Pour les éléments tels que `<input>`, pour lesquels vous voulez aussi que `v-model` fonctionne, il est souvent utile de créer une nouvelle propriété calculée pour les écouteurs, telle que la propriété `inputListeners` ci-dessous:

```

Vue.component('base-input', {
  inheritAttrs: false,
  props: ['label', 'value'],
  computed: {
    inputListeners: function () {
      var vm = this
      // `Object.assign` fusionne les objets ensemble pour en former un
      nouveau
      return Object.assign({},
        // Nous ajoutons tous les écouteurs du parent
        this.$listeners,
        // Puis nous pouvons ajouter des écouteurs personnalisés
        // ou surcharger le comportement de certains écouteurs.
        {
          // Cela garantit que le composant fonctionne avec v-model
          input: function (event) {
            vm.$emit('input', event.target.value)
          }
        }
      )
    },
    template: `
      <label>

```

```
  {{ label }}  
  <input  
    v-bind="$attrs"  
    v-bind:value="value"  
    v-on="inputListeners"  
  >  
  </label>  
)
```

À présent, le composant `<base-input>` est une **enveloppe complètement transparente**, c'est-à-dire qu'il peut être utilisé comme un élément `<input>` normal : tous les mêmes attributs et écouteurs d'événements fonctionneront, sans le modificateur `.native`.

Modificateur `.sync`

Nouveauté de la 2.3.0+

Dans certains cas, nous pouvons avoir besoin d'une « liaison à double sens » (*two-way binding*) pour une prop. Malheureusement, une vraie liaison à double sens peut créer des problèmes de maintenance, car les composants enfant peuvent faire muter le parent sans que la source de cette mutation soit explicite que ce soit dans le parent ou l'enfant.

C'est pourquoi à la place, nous recommandons d'émettre des événements en suivant le modèle `update:myPropName`. Par exemple, dans un composant hypothétique avec une prop `title`, nous pourrions communiquer l'intention d'assigner une nouvelle valeur avec :

```
this.$emit('update:title', nouveauTitre)
```

Ensuite le parent peut écouter cet événement et mettre à jour une propriété de donnée locale s'il le désire. Par exemple :

```
<text-document  
  v-bind:title="doc.title"  
  v-on:update:title="doc.title = $event"  
></text-document>
```

Pour plus de commodité, nous proposons un raccourci pour cette technique avec le modificateur `.sync` :

```
<text-document v-bind:title.sync="doc.title"></text-document>
```

Notez que `v-bind` avec le modificateur `.sync` ne fonctionne **pas** avec des expressions (par ex. `v-bind:title.sync="doc.title + '!''` est invalide). À la place, vous devez seulement fournir le nom de la propriété que vous voulez lier, similaire à `v-model`.

Le modificateur `.sync` peut également être utilisé avec `v-bind` quand on utilise un objet pour assigner plusieurs props à la fois :

```
<text-document v-bind.sync="doc"></text-document>
```

Cela passe chaque propriété dans l'objet `doc` (p. ex. `title`) en tant que prop individuelle, puis ajoute des écouteurs `v-on` de mise à jour pour chacune.

Utiliser `v-bind.sync` avec un objet littéral, par exemple `v-bind.sync="{ title: doc.title }"`, ne fonctionnera pas. En effet, il y a trop de cas particuliers à considérer pour pouvoir analyser une telle expression.

Hooks de cycle de vie d'une instance

Chaque instance de vue traverse une série d'étapes d'initialisation au moment de sa création - par exemple, elle doit mettre en place l'observation des données, compiler le template, monter l'instance sur le DOM et mettre à jour le DOM quand les données changent.

En cours de route, elle va aussi invoquer des **hooks de cycle de vie**, qui nous donnent l'opportunité d'exécuter une logique personnalisée à chaque niveau.

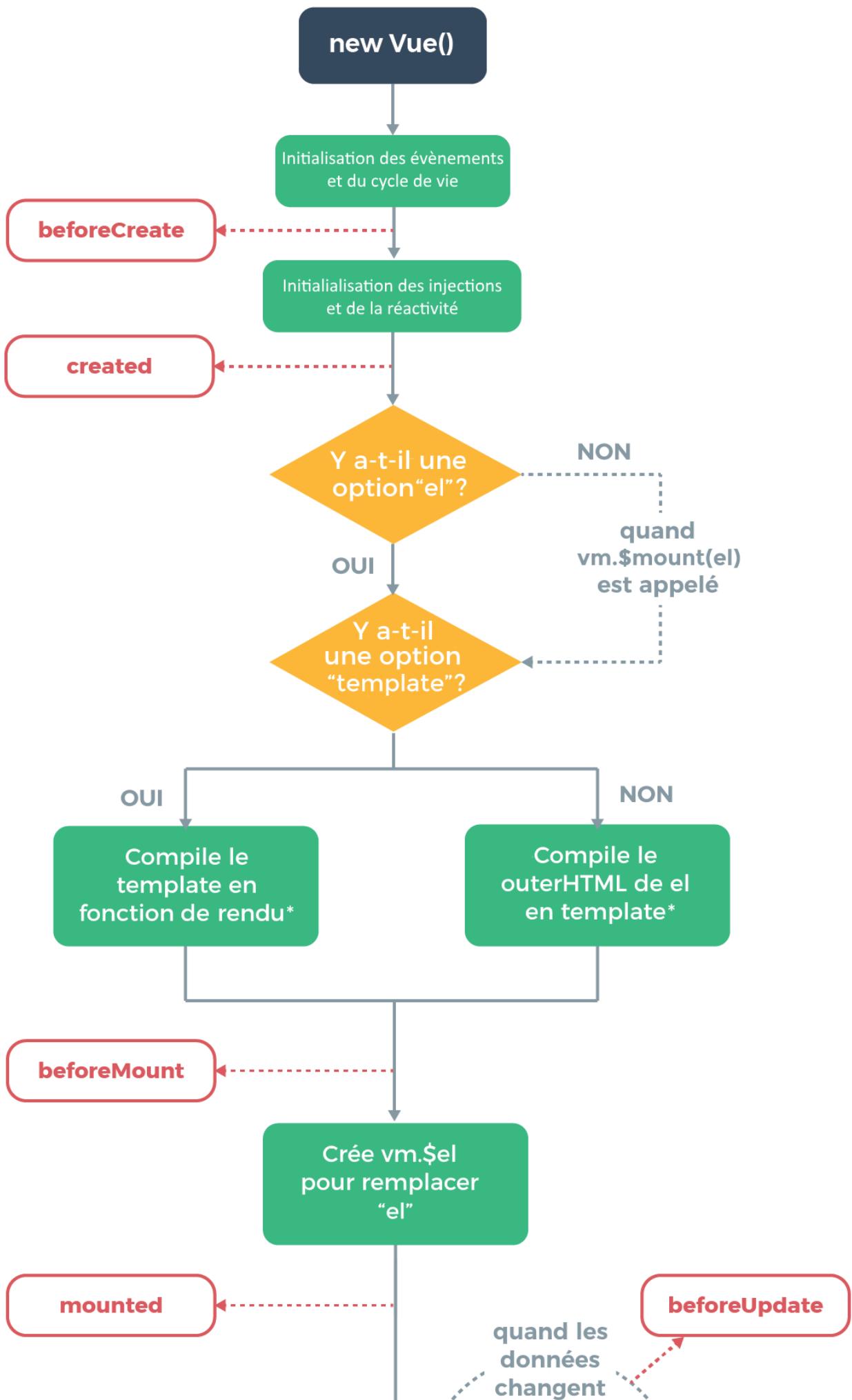
Par exemple, le hook `created` est appelé une fois l'instance créée :

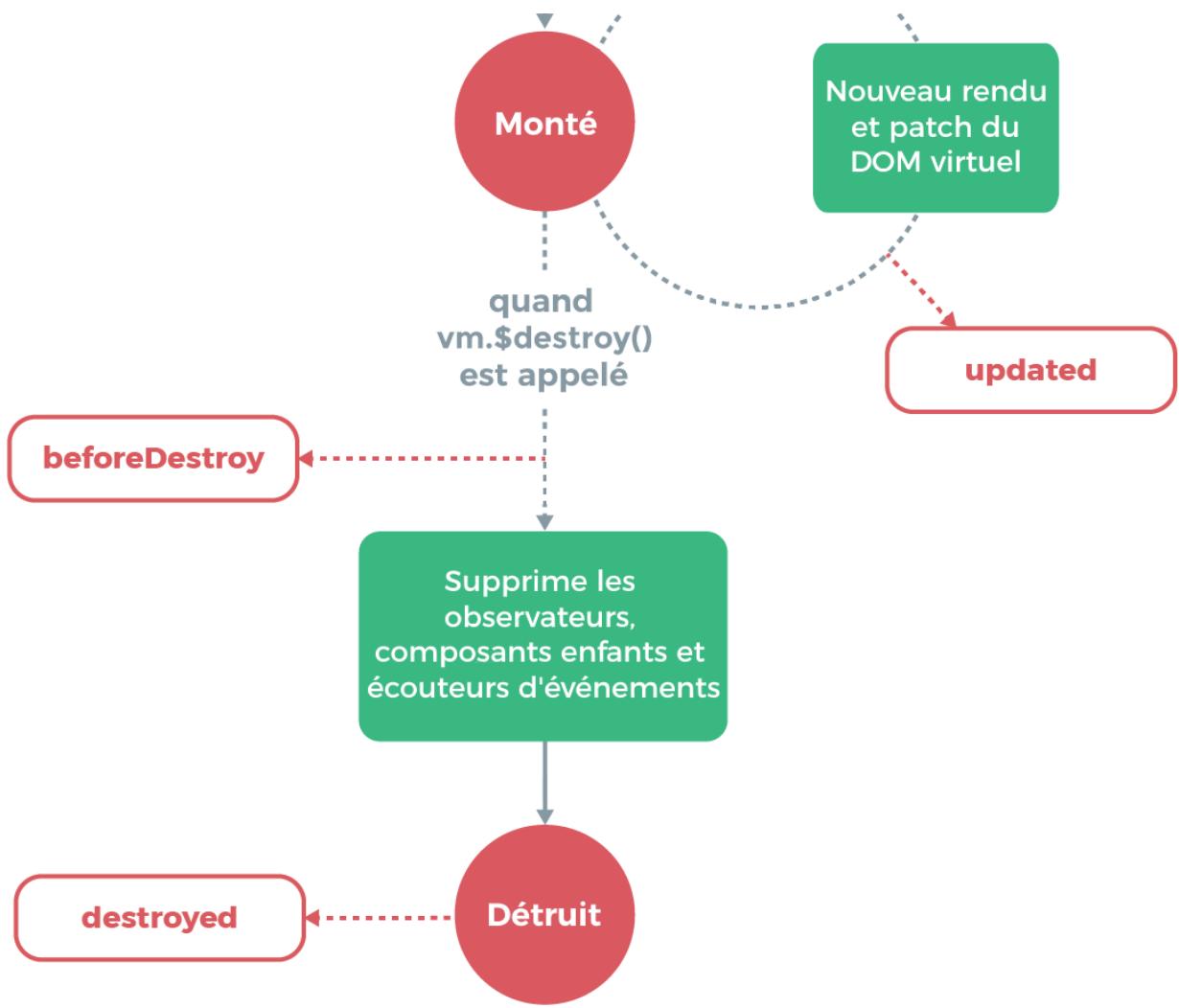
```
new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` est une référence à l'instance de vm
    console.log('a is: ' + this.a)
  }
})
// => "a is: 1"
```

Il y a aussi d'autres hooks qui seront appelés à différentes étapes du cycle de vie d'une instance, par exemple `mounted`, `updated` et `destroyed`.

Tous ces hooks de cycle de vie sont appelés avec leur `this` pointant sur l'instance de la vue qui les invoque.

Diagramme du cycle de vie





* la compilation est faite à l'avance si vous utilisez une étape de build, par ex. des composants monofichiers

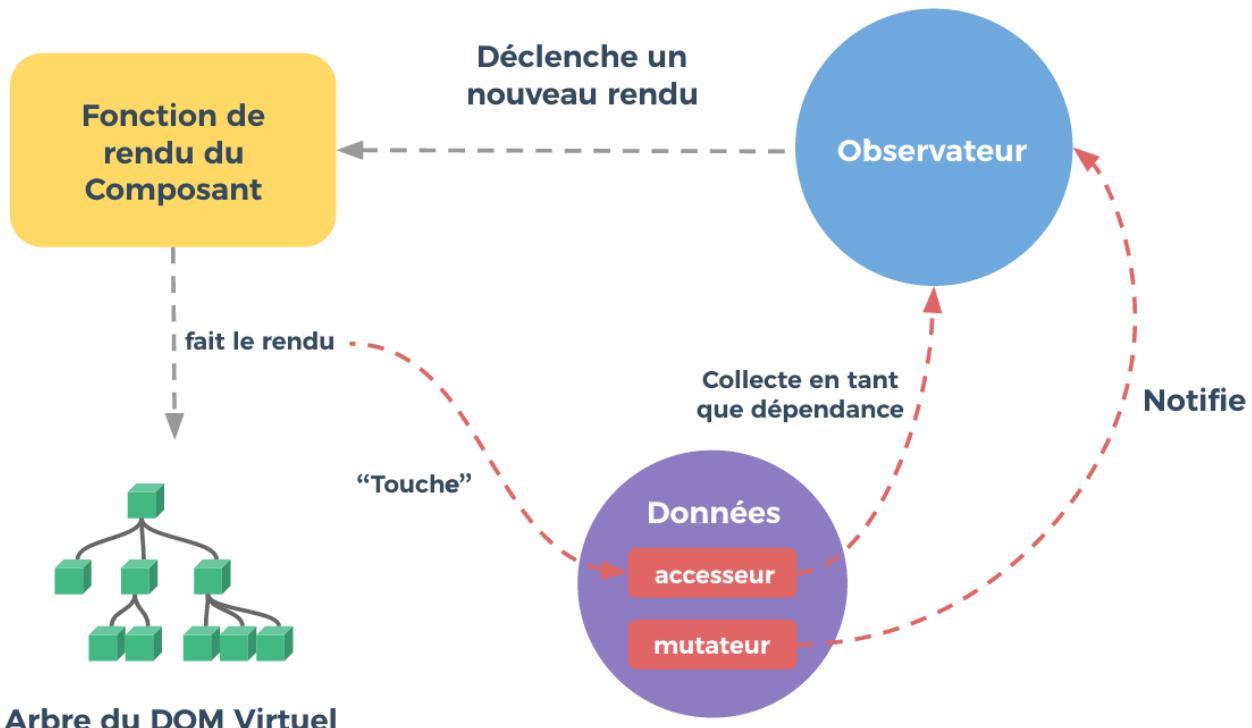


Syntaxe des templates

Syntaxe des templates

Vue.js utilise une syntaxe de template basée sur le HTML qui vous permet de lier déclarativement le DOM rendu aux données de l'instance sous-jacente de Vue.

Tous les templates de Vue.js sont du HTML valide qui peut être interprété par les navigateurs et les interpréteurs HTML conformes aux spécifications.



Si vous êtes familiers avec les concepts de DOM virtuel et que vous préférez la puissance du JavaScript pur, vous pouvez aussi écrire directement des fonctions de rendu à la place des templates, avec un support facultatif de JSX.

Interpolations

Texte

La forme la plus élémentaire de la liaison de données est l'interpolation de texte en utilisant la syntaxe "Mustache" (les doubles accolades)

```
<span>Message: {{ msg }}</span>
```

La balise moustache sera remplacée par la valeur de la propriété `msg` de l'objet data correspondant. Elle sera également mise à jour à chaque fois que la propriété `msg` de l'objet data changera.

Vous pouvez également réaliser des interpolations à usage unique qui ne se mettront pas à jour lors de la modification des données en utilisant la directive `v-once`, mais gardez à l'esprit que cela affectera toutes les liaisons de données présentes sur le même nœud :

```
<span v-once>Ceci ne changera jamais : {{ msg }}</span>
```

Interprétation du HTML

Les doubles moustaches interprètent la donnée en tant que texte brut, pas en tant que HTML. Pour afficher réellement du HTML, vous aurez besoin d'utiliser la directive `v-html`

```
<p>Using mustaches: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

```
<div id="example1" class="demo">
  <p>En utilisant les moustaches : {{ rawHtml }}</p>
  <p>En utilisant la directive v-html : <span v-html="rawHtml"></span></p>
</div>
<script>
new Vue({
  el: '#example1',
  data: function () {
    return {
      rawHtml: '<span style="color: red">Ceci devrait être rouge.</span>'
    }
  }
)
</script>
```

Attributs

Les moustaches ne peuvent pas être utilisées à l'intérieur des attributs HTML, à la place utilisez une directive `v-bind` :

```
<div v-bind:id="dynamicId"></div>
```

Dans le cas des attributs booléens qui impliquent la présence d'une valeur évaluée à `true`, `v-bind` fonctionne un peu différemment. Dans cet exemple :

```
<button v-bind:disabled="isButtonDisabled">Button</button>
```

Si `isButtonDisabled` a la valeur `null`, `undefined`, ou `false`, l'attribut `disabled` ne sera pas inclus dans l'élément `<button>` généré.

Utilisation des expressions JavaScript

Jusqu'ici, nous avons seulement lié de simples clés de propriétés dans nos templates. Mais Vue.js supporte en réalité toute la puissance des expressions JavaScript à l'intérieur de toutes les liaisons de données :

```
{{ number + 1 }}  
{{ ok ? 'OUI' : 'NON' }}  
{{ message.split('').reverse().join('') }}  
<div v-bind:id="'list-' + id"></div>
```

Ces expressions seront évaluées en tant que JavaScript au sein de la portée des données de l'instance de Vue propriétaire. Il y a une restriction : chacune de ces liaisons ne peut contenir **qu'une seule expression**, donc ce qui suit **NE** fonctionnera **PAS**

```
<!-- ceci est une déclaration, pas une expression: -->  
{ { var a = 1 } }  
  
<!-- le contrôle de flux ne marchera pas non plus, utilisez des  
expressions ternaires -->  
{ { if (ok) { return message } } }
```

Les expressions de template sont isolées et ont seulement accès à une liste blanche de globales telles que `Math` et `Date`. Vous ne devriez pas tenter d'accéder à des variables globales définies par l'utilisateur dans les expressions de template.

Directives

Les directives sont des attributs spéciaux avec le préfixe `v-`. Les valeurs attendues pour les attributs de directives sont **une unique expression JavaScript** (à l'exception de `v-for`, qui sera expliquée plus loin). Le travail d'une directive est d'appliquer réactivement des effets secondaires au DOM quand la valeur de son expression change. Revenons à l'exemple vu dans l'introduction :

```
<p v-if="seen">Maintenant vous me voyez</p>
```

Ici, la directive `v-if` retirerait / insèrerait l'élément `<p>` selon que l'expression `seen` soit considérée comme fausse / vraie.

Arguments

Certaines directives peuvent prendre un "argument", indiqué par un deux-points après le nom de la directive. Par exemple, la directive `v-bind` est utilisée pour mettre à jour réactivement un attribut HTML :

```
<a v-bind:href="url"> ... </a>
```

Ici `href` est un argument, qui dit à la directive `v-bind` de lier l'attribut `href` de l'élément à la valeur de l'expression `url`.

Un autre exemple est la directive `v-on`, qui écoute les événements du DOM :

```
<a v-on:click="doSomething"> ... </a>
```

Ici l'argument est le nom de l'évènement à écouter. Nous parlerons aussi plus en détail de la gestion des évènements.

Arguments dynamiques

Nouveau dans la 2.6.0+

Introduit dans la version 2.6.0, il est aussi possible d'utiliser des expressions JavaScript dans un argument de directive inclus entre crochets :

```
<!--  
Notez qu'il y a diverses contraintes aux expressions d'argument, comme  
expliqué  
dans la section « Contraintes des expressions d'argument dynamique » ci-  
après.  
-->  
<a v-bind:[attributeName]="url"> ... </a>
```

Ici `attributeName` va dynamiquement être évalué comme une expression JavaScript et cette valeur d'évaluation va être utilisée comme valeur finale pour l'argument. Par exemple, si votre instance de Vue a une

propriété de donnée, `attributeName`, et que cette valeur est "`href`", alors la liaison sera équivalente à `v-bind:href`.

De manière similaire, vous pouvez utiliser les arguments dynamiques pour lier un gestionnaire d'évènement à un nom d'évènement dynamique :

```
<a v-on:[eventName] = "doSomething"> ... </a>
```

De manière similaire, donc, quand la valeur `eventName` est "`focus`", par exemple, `v-on:[eventName]` sera équivalent à `v-on:focus`.

Contrainte des valeurs d'argument dynamique

Les arguments dynamiques sont destinés à être évalués comme des chaînes de caractères à l'exception de `null`. La valeur spéciale `null` peut être utilisée pour explicitement retirer la liaison. N'importe quelle autre valeur n'étant pas une chaîne de caractères lèvera un avertissement.

Contraintes des expressions d'argument dynamique

Les expressions d'argument dynamique ont quelques contraintes de syntaxe car certains caractères sont invalides à l'intérieur d'un nom d'attribut HTML comme les espaces et les guillemets. Par exemple, ce qui suit est invalide :

```
<!-- Ceci va lever un avertissement de compilation. -->
<a v-bind:['foo' + bar] = "value"> ... </a>
```

La solution consiste à utiliser des expressions sans espaces ni guillemets, ou à remplacer l'expression complexe par une propriété calculée.

De plus, si vous utilisez des templates dans le DOM (templates directement écrits dans un fichier HTML), vous devez éviter les majuscules dans vos clés car les navigateurs convertissent les noms d'attribut en lettre minuscule :

```
<!--
Ceci va être converti en v-bind:[someattr] dans un template dans le DOM. --
-->
À moins que vous ne fassiez référence à la propriété `someattr` dans
votre instance, votre code ne fonctionnera pas.
-->
<a v-bind:[someAttr] = "value"> ... </a>
```

Modificateurs

Les modificateurs sont des suffixes spéciaux indiqués par un point, qui indique qu'une directive devrait être liée d'une manière spécifique. Par exemple, le modificateur `.prevent` dit à la directive `v-on` d'appeler `event.preventDefault()` lorsque l'évènement survient.

```
<form v-on:submit.prevent="onSubmit"> ... </form>
```

Nous verrons plus de cas d'utilisation des modificateurs plus loin quand nous porterons un regard plus attentif sur **v-on** et **v-model**.

Abréviations

Le préfixe **v-** sert d'indicateur visuel pour identifier les attributs spécifiques à Vue dans vos templates. C'est pratique lorsque vous utilisez Vue.js pour appliquer des comportements dynamiques sur un balisage existant, mais peut sembler verbeux pour des directives utilisées fréquemment. Par ailleurs, le besoin d'un préfixe **v-** devient moins important quand vous développez une [application monopage](#) où Vue.js gère tous les templates. C'est pourquoi Vue.js fournit des abréviations pour deux des directives les plus utilisées, **v-bind** et **v-on** :

Abréviation pour **v-bind**

```
<!-- syntaxe complète -->
<a v-bind:href="url"> ... </a>

<!-- abréviation -->
<a :href="url"> ... </a>

<!-- abréviation avec argument dynamique (2.6.0+) -->
<a :[key]="url"> ... </a>
```

Abréviation pour **v-on**

```
<!-- syntaxe complète -->
<a v-on:click="doSomething"> ... </a>

<!-- abréviation -->
<a @click="doSomething"> ... </a>

<!-- abréviation avec argument dynamique (2.6.0+) -->
<a @[event]="doSomething"> ... </a>
```

Cela peut paraître un peu différent du HTML classique, mais **:** et **@** sont des caractères valides pour des noms d'attributs et tous les navigateurs supportés par Vue.js peuvent l'interpréter correctement. De plus, ils n'apparaissent pas dans le balisage final. La syntaxe abrégée est totalement optionnelle, mais vous allez probablement l'apprécier quand vous en apprendrez plus sur son utilisation plus loin.

v-if

La directive **v-if** est utilisée pour conditionnellement faire le rendu d'un bloc. Le rendu du bloc sera effectué uniquement si l'expression de la directive retourne une valeur évaluée à vrai.

```
<h1 v-if="awesome">Vue est extraordinaire !</h1>
```

Il est également possible d'ajouter une structure « sinon » avec **v-else** :

```
<h1 v-if="awesome">Vue est extraordinaire !</h1>
<h1 v-else>Oh non 😢 </h1>
```

Groupes conditionnels avec **v-if** dans un **<template>**

Comme **v-if** est une directive, elle doit être attachée à un seul élément. Mais comment faire si nous voulons permuter plusieurs éléments ? Dans ce cas, nous pouvons utiliser **v-if** sur un élément **<template>**, qui sert d'enveloppe invisible. Le résultat final rendu n'inclura pas l'élément **<template>**.

```
<template v-if="ok">
  <h1>Titre</h1>
  <p>Paragraphe 1</p>
  <p>Paragraphe 2</p>
</template>
```

v-else

Vous pouvez utiliser la directive **v-else** pour indiquer une « structure sinon » pour **v-if** :

```
<div v-if="Math.random() > 0.5">
  Maintenant vous me voyez
</div>
<div v-else>
  Maintenant vous ne me voyez pas
</div>
```

Un élément **v-else** doit immédiatement suivre un élément **v-if** ou un élément **v-else-if** (sinon il ne sera pas reconnu).

v-else-if

Nouveau dans la 2.1.0+

Le **v-else-if**, comme le nom le suggère, sert comme une « structure sinon si » pour **v-if**. Il peut également être enchainé plusieurs fois :

```

<div v-if="type === 'A'>
  A
</div>
<div v-else-if="type === 'B'>
  B
</div>
<div v-else-if="type === 'C'>
  C
</div>
<div v-else>
  Ni A, ni B et ni C
</div>

```

Semblable à `v-else`, un élément `v-else-if` doit immédiatement suivre un élément `v-if` ou un élément `v-else-if`.

Contrôle des éléments réutilisables avec `key`

Vue tente de restituer les éléments aussi efficacement que possible, en les réutilisant souvent au lieu de faire de la restitution à partir de zéro. En plus de permettre à Vue d'être très rapide, cela peut avoir quelques avantages utiles. Par exemple, si vous autorisez les utilisateurs à choisir entre plusieurs types de connexion :

```

<template v-if="loginType === 'username'>
  <label>Nom d'utilisateur</label>
  <input placeholder="Entrez votre nom d'utilisateur">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Entrez votre adresse email">
</template>

```

Le fait de changer de `loginType` dans le code ci-dessus n'effacera pas ce que l'utilisateur a déjà saisi. Puisque les deux templates utilisent les mêmes éléments, le `<input>` n'est pas remplacé (juste son `placeholder`).

Vérifiez-le par vous-même en entrant un texte dans la zone de saisie, puis en appuyant sur le bouton de permutation :

```

<div id="no-key-example" class="demo">
  <div>
    <template v-if="loginType === 'username'>
      <label>Nom d'utilisateur</label>
      <input placeholder="Entrez votre nom d'utilisateur">
    </template>
    <template v-else>
      <label>Email</label>
      <input placeholder="Entrez votre adresse email">
    </template>
  </div>
  <button @click="toggleLoginType">Changer de type de connexion</button>

```

```

</div>
<script>
new Vue({
  el: '#no-key-example',
  data: {
    loginType: 'username'
  },
  methods: {
    toggleLoginType: function () {
      return this.loginType = this.loginType === 'username' ? 'email' :
    'username'
    }
  }
})
</script>

```

Ce n'est pas toujours souhaitable cependant, c'est pourquoi Vue vous offre un moyen de dire, « Ces deux éléments sont complètement distincts, ne les réutilisez pas ». Ajoutez juste un attribut `key` avec des valeurs uniques :

```

<template v-if="loginType === 'username'">
  <label>Nom d'utilisateur</label>
  <input placeholder="Entrez votre nom d'utilisateur" key="username-
input">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Entrez votre adresse email" key="email-input">
</template>

```

Maintenant ces zones de saisie seront restituées à partir de zéro à chaque fois que vous permuterez. Voyez par vous-même :

```

<div id="key-example" class="demo">
  <div>
    <template v-if="loginType === 'username'">
      <label>Nom d'utilisateur</label>
      <input placeholder="Entrez votre nom d'utilisateur" key="username-
input">
    </template>
    <template v-else>
      <label>Email</label>
      <input placeholder="Entrez votre adresse email" key="email-input">
    </template>
  </div>
  <button @click="toggleLoginType">Changer de type de connexion</button>
</div>
<script>
new Vue({
  el: '#key-example',

```

```

data: {
  loginType: 'username'
},
methods: {
  toggleLoginType: function () {
    return this.loginType = this.loginType === 'username' ? 'email' :
  'username'
  }
}
</script>

```

Remarquez que les éléments <label> sont réutilisés efficacement, car ils n'ont pas d'attributs key.

v-show

Une autre option pour afficher conditionnellement un élément est la directive v-show. L'utilisation est en grande partie la même :

```
<h1 v-show="ok">Bonjour !</h1>
```

La différence est qu'un élément avec v-show sera toujours restitué et restera dans le DOM ; v-show permute simplement la propriété CSS display de l'élément.

v-if vs v-show

v-if est un « vrai » rendu conditionnel car il garantit que les écouteurs d'évènements et les composants enfants à l'intérieur de la structure conditionnelle sont correctement détruits et recréés lors des permutations.

v-if est également paresseux : si la condition est fausse sur le rendu initial, il ne fera rien (la structure conditionnelle sera rendue quand la condition sera vraie pour la première fois).

En comparaison, v-show est beaucoup plus simple. L'élément est toujours rendu indépendamment de la condition initiale, avec juste une simple permutation basée sur du CSS.

D'une manière générale, v-if a des couts à la permutation plus élevés alors que v-show a des couts au rendu initial plus élevés. Donc préférez v-show si vous avez besoin de permuter quelque chose très souvent et préférez v-if si la condition ne change probablement pas à l'exécution.

v-if avec v-for

Utiliser v-if et v-for ensemble n'est pas recommandé.

Associer un tableau à des éléments avec `v-for`

Nous pouvons utiliser la directive `v-for` pour faire le rendu d'une liste d'éléments en nous basant sur un tableau. La directive `v-for` utilise une syntaxe spécifique de la forme `item in items`, où `items` représente le tableau source des données et où `item` est un **alias** représentant l'élément du tableau en cours d'itération :

```
<ul id="example-1" class="demo">
  <li v-for="item in items">
    {{item.message}}
  </li>
</ul>
<script>
var example1 = new Vue({
  el: '#example-1',
  data: {
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
</script>
```

À l'intérieur des structures `v-for`, nous avons un accès complet aux propriétés de la portée parente. `v-for` supporte également un second argument optionnel représentant l'index de l'élément courant.

```
<ul id="example-2" class="demo">
  <li v-for="(item, index) in items">
    {{ parentMessage }} - {{ index }} - {{ item.message }}
  </li>
</ul>
<script>
var example2 = new Vue({
  el: '#example-2',
  data: {
    parentMessage: 'Parent',
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
</script>
```

Vous pouvez également utiliser `of` en tant que mot-clé à la place de `in` pour être plus proche de la syntaxe JavaScript concernant l'utilisation des itérateurs :

```
<div v-for="item of items"></div>
```

v-for avec l'objet

Vous pouvez aussi utiliser **v-for** pour itérer sur les propriétés d'un objet.

```
<ul id="v-for-object" class="demo">
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>
<script>
new Vue({
  el: '#v-for-object',
  data: {
    object: {
      title: 'How to do lists in Vue',
      author: 'Jane Doe',
      publishedAt: '2016-04-10'
    }
  }
})
</script>
```

Vous pouvez également fournir un deuxième argument représentant le nom de la propriété courante (c.-à-d. la clé) :

```
<div id="v-for-object-value-name" class="demo">
  <div v-for="(value, name) in object">
    {{ name }}: {{ value }}
  </div>
</div>
<script>
new Vue({
  el: '#v-for-object-value-name',
  data: {
    object: {
      title: 'How to do lists in Vue',
      author: 'Jane Doe',
      publishedAt: '2016-04-10'
    }
  }
})
</script>
```

Et un autre pour l'index :

```

<div id="v-for-object-value-name-index" class="demo">
  <div v-for="(value, name, index) in object">
    {{ index }}. {{ name }}: {{ value }}
  </div>
</div>
<script>
new Vue({
  el: '#v-for-object-value-name-index',
  data: {
    object: {
      title: 'How to do lists in Vue',
      author: 'Jane Doe',
      publishedAt: '2016-04-10'
    }
  }
)
</script>

```

Maintaining State

Quand Vue met à jour une liste d'éléments rendus avec `v-for`, il utilise par défaut une stratégie de « modification localisée » (*in-place patch*).

Pour expliquer à Vue comment suivre l'identité de chaque nœud, afin que les éléments existants puissent être réutilisés et réordonnés, vous devez fournir un attribut unique `key` pour chaque élément :

```

<div v-for="item in items" :key="item.id">
  <!-- contenu -->
</div>

```

Il est recommandé de fournir une `key` avec `v-for` chaque fois que possible, à moins que le contenu itéré du DOM soit simple ou que vous utilisiez intentionnellement le comportement de base pour un gain de performance.

Détection de changement dans un tableau

Méthodes de mutation

Vue surcharge les méthodes de mutation d'un tableau observé afin qu'elles déclenchent également des mises à jour de la vue. Les méthodes encapsulées sont :

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

Vous pouvez ouvrir la console et jouer avec la liste des éléments `items` des exemples précédents en appelant leurs méthodes de mutation. Par exemple : `example1.items.push({ message: 'Baz' })`.

Remplacer un tableau

Les méthodes de mutation, comme leur nom le suggère, modifient le tableau d'origine sur lequel elles sont appelées. En comparaison, il y a aussi des méthodes non-mutatives comme par ex. `filter()`, `concat()` et `slice()`, qui ne changent pas le tableau original mais **retourne toujours un nouveau tableau**. Quand vous travaillez avec des méthodes non-mutatives, vous pouvez juste remplacer l'ancien tableau par le nouveau :

```
example1.items = example1.items.filter(function (item) {  
  return item.message.match(/Foo/  
})
```

Limitations

À cause des limitations en JavaScript, Vue **ne peut pas** détecter les changements suivants dans un tableau :

1. Quand vous affectez directement un élément à un index. Par ex. : `vm.items[indexOfItem] = newValue`
2. Quand vous modifiez la longueur du tableau. Par ex. : `vm.items.length = newLength`

Par exemple :

```
var vm = new Vue({  
  data: {  
    items: ['a', 'b', 'c']  
  }  
})  
vm.items[1] = 'x' // N'est PAS réactive  
vm.items.length = 2 // N'est PAS réactive
```

Pour contourner la première limitation, les deux exemples suivants accomplissent la même chose que `vm.items[indexOfItem] = newValue`, mais vont également déclencher des mises à jour de l'état dans le système de réactivité :

```
// Vue.set  
Vue.set(vm.items, indexOfItem, newValue)
```

```
// Array.prototype.splice  
vm.items.splice(indexOfItem, 1, newValue)
```

Vous pouvez également utiliser la méthode d'instance `vm.$set`, qui est un alias pour la méthode globale `Vue.set` :

```
vm.$set(vm.items, indexOfItem, newValue)
```

Pour gérer la seconde limitation, vous pouvez également utiliser `splice` :

```
vm.items.splice(newLength)
```

Limitation de détection de changement dans un objet

De nouveau, à cause de limitation du JavaScript, **Vue ne peut détecter l'ajout ou la suppression d'une propriété**. Par exemple :

```

var vm = new Vue({
  data: {
    a: 1
  }
})
// `vm.a` est maintenant réactive

vm.b = 2
// `vm.b` N'est PAS réactive

```

Vue ne permet pas d'ajouter dynamiquement de nouvelles propriétés réactives au niveau racine sur des instances déjà créées. Cependant, il est possible d'ajouter des propriétés réactives aux objets imbriqués en utilisant la méthode `Vue.set(object, propertyName, value)`. Par exemple avec :

```

var vm = new Vue({
  data: {
    userProfile: {
      name: 'Anika'
    }
  }
})

```

Vous pourriez ajouter une nouvelle propriété `age` à l'objet imbriqué `userProfile` avec :

```
Vue.set(vm.userProfile, 'age', 27)
```

Vous pouvez également utiliser la méthode d'instance `vm.$set`, qui est juste un alias de la méthode globale `Vue.set` :

```
vm.$set(vm.userProfile, 'age', 27)
```

Parfois vous voudrez affecter plusieurs nouvelles propriétés à un objet existant, par exemple en utilisant `Object.assign()` ou `_.extend()`. Dans ce cas, vous devrez créer un nouvel objet avec les propriétés des deux objets. Donc au lieu de :

```

Object.assign(vm.userProfile, {
  age: 27,
  favoriteColor: 'Vert Vue'
})

```

Vous ajouterez une nouvelle propriété réactive avec :

```
vm.userProfile = Object.assign({}, vm.userProfile, {  
  age: 27,  
  favoriteColor: 'Vert Vue'  
})
```

Affichage de résultats filtrés/triés

Parfois nous voulons afficher une version filtrée ou triée d'un tableau sans pour autant modifier ou réassigner les données d'origine. Dans ce cas, vous pouvez créer une propriété calculée qui retourne le tableau filtré ou trié.

Par exemple :

```
<li v-for="n in evenNumbers">{{ n }}</li>
```

```
data: {  
  numbers: [ 1, 2, 3, 4, 5 ]  
,  
computed: {  
  evenNumbers: function () {  
    return this.numbers.filter(function (number) {  
      return number % 2 === 0  
    })  
  }  
}
```

Dans les situations où les propriétés calculées ne sont pas utilisables (par ex. à l'intérieur d'une boucle `v-for` imbriquée), vous pouvez juste utiliser une méthode :

```
<li v-for="n in even(numbers)">{{ n }}</li>
```

```
data: {  
  numbers: [ 1, 2, 3, 4, 5 ]  
,  
methods: {  
  even: function (numbers) {  
    return numbers.filter(function (number) {  
      return number % 2 === 0  
    })  
  }  
}
```

`v-for` et plage de valeurs

`v-for` peut également prendre un nombre entier. Dans ce cas, il répètera le template autant de fois qu'indiqué.

```
<div id="range" class="demo">
  <span v-for="n in 10">{{ n }} </span>
</div>
<script>
  new Vue({ el: '#range' })
</script>
```

Template `v-for`

De la même manière qu'avec `v-if`, vous pouvez également utiliser la balise `<template>` avec `v-for` pour faire le rendu d'une structure contenant de multiples éléments. Par exemple :

```
<ul>
  <template v-for="item in items">
    <li>{{ item.msg }}</li>
    <li class="divider" role="presentation"></li>
  </template>
</ul>
```

Composants et `v-for`

Vous pouvez directement utiliser `v-for` sur un composant personnalisé, comme sur n'importe quel autre élément standard :

```
<my-component v-for="item in items" :key="item.id"></my-component>
```

En 2.2.0+, lors de l'utilisation de `v-for` avec un composant, une `key` (clé) est maintenant requise.

Cependant, cela ne passera pas automatiquement les données au composant parce que les composants ont leurs propres portées isolées. Pour passer les données itérées au composant, nous devons utiliser les props en plus :

```
<my-component
  v-for="(item, index) in items"
  v-bind:item="item"
  v-bind:index="index"
  v-bind:key="item.id"
></my-component>
```

La raison pour ne pas injecter automatiquement `item` dans le composant est que cela le rendrait fortement couplé au fonctionnement de `v-for`. Être explicite sur l'endroit d'où proviennent les données rend le composant réutilisable dans d'autres situations.

Voici un exemple complet d'une simple liste de tâches :

```
<div id="todo-list-example">
  <form v-on:submit.prevent="addNewTodo">
    <label for="new-todo">Ajouter une tâche</label>
    <input
      v-model="newTodoText"
      id="new-todo"
      placeholder="Ex. nourrir le chat"
    >
    <button>Add</button>
  </form>
  <ul>
    <li
      is="todo-item"
      v-for="(todo, index) in todos"
      v-bind:key="todo.id"
      v-bind:title="todo.title"
      v-on:remove="todos.splice(index, 1)"
    ></li>
  </ul>
</div>
```

Liaison de Classes HTML

Un besoin classique de la liaison de données est la manipulation de la liste des classes d'un élément, ainsi que ses styles en ligne.

Puisque ce sont tous deux des attributs, il est possible d'utiliser **v-bind** pour les gérer : il faut simplement générer une chaîne de caractère avec nos expressions. Cependant la concaténation de chaîne de caractères est fastidieuse et source d'erreur.

Pour cette raison, Vue fournit des améliorations spécifiques quand **v-bind** est utilisé avec **class** et **style**. En plus des chaînes de caractères, l'expression peut évaluer des objets ou des tableaux.

Syntaxe Objet

Il est possible de passer un objet à **v-bind:class** pour permuter les classes automatiquement :

```
<div v-bind:class="{ active: isActive }"></div>
```

La syntaxe ci-dessus signifie que la classe **active** sera présente si la propriété **isActive** est [considérée comme vraie](#).

Vous pouvez permuter plusieurs classes en ayant plus de champs dans l'objet. De plus, la directive **v-bind:class** peut aussi coexister avec l'attribut **class**. Donc, avec le template suivant :

```
<div  
  class="static"  
  v-bind:class="{ active: isActive, 'text-danger': hasError }"  
></div>
```

Et les données suivantes :

```
data: {  
  isActive: true,  
  hasError: false  
}
```

Le rendu sera :

```
<div class="static active"></div>
```

Quand **isActive** ou **hasError** change, la liste des classes sera mise à jour en conséquence. Par exemple, si **hasError** devient **true**, la liste des classes deviendra "**static active text-danger**".

L'objet lié n'a pas besoin d'être déclaré dans l'attribut :

```
<div v-bind:class="classObject"></div>
```

```
data: {
  classObject: {
    active: true,
    'text-danger': false
  }
}
```

Ceci rendra le même résultat. Il est également possible de lier une [propriété calculée](#) qui retourne un objet. C'est une méthode courante et puissante :

```
<div v-bind:class="classObject"></div>
```

```
data: {
  isActive: true,
  error: null
},
computed: {
  classObject: function () {
    return {
      active: this.isActive && !this.error,
      'text-danger': this.error && this.error.type === 'fatal'
    }
  }
}
```

Syntaxe Tableau

Il est possible de passer un tableau à `v-bind:class` pour appliquer une liste de classes :

```
<div v-bind:class="[activeClass, errorClass]"></div>
```

```
data: {
  activeClass: 'active',
  errorClass: 'text-danger'
}
```

Ce qui rendra :

```
<div class="active text-danger"></div>
```

Si vous voulez permuter une classe de la liste en fonction d'une condition, vous pouvez le faire avec une expression ternaire :

```
<div v-bind:class="[isActive ? activeClass : '', errorClass]"></div>
```

Ceci appliquera toujours la classe `errorClass`, mais appliquera `activeClass` uniquement quand `isActive` vaut `true`.

En revanche, cela peut être un peu verbeux si vous avez plusieurs classes conditionnelles. C'est pourquoi il est aussi possible d'utiliser la syntaxe objet dans la syntaxe tableau :

```
<div v-bind:class="{ active: isActive, errorClass }"></div>
```

Avec des Composants

Quand vous utilisez l'attribut `class` sur un composant personnalisé, ces classes seront ajoutées à l'élément à la racine du composant. Les classes présentes sur cet élément ne seront pas réécrites.

Par exemple, si vous déclarez ce composant :

```
Vue.component('my-component', {  
  template: '<p class="foo bar">Hi</p>'  
})
```

Puis ajoutez quelques classes quand vous l'utilisez :

```
<my-component class="baz boo"></my-component>
```

Le rendu HTML sera :

```
<p class="foo bar baz boo">Hi</p>
```

C'est aussi vrai pour la liaison de classe :

```
<my-component v-bind:class="{ active: isActive }"></my-component>
```

Quand la propriété `isActive` est évaluée à vrai, le rendu HTML sera :

```
<p class="foo bar active">Hi</p>
```

Liaison de Styles HTML

Syntaxe objet

La syntaxe objet pour `v-bind:style` est assez simple - cela ressemble presque à du CSS, sauf que c'est un objet JavaScript. Vous pouvez utiliser camelCase ou kebab-case (utilisez des apostrophes avec kebab-case) pour les noms des propriétés CSS :

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }">  
</div>
```

```
data: {  
  activeColor: 'red',
```

```
    fontSize: 30  
}
```

C'est souvent une bonne idée de lier directement un objet de style, pour que le template soit plus propre :

```
<div v-bind:style="styleObject"></div>
```

```
data: {  
  styleObject: {  
    color: 'red',  
    fontSize: '13px'  
  }  
}
```

Encore une fois, la syntaxe objet est souvent utilisée en conjonction avec des propriétés calculées retournant des objets.

Syntaxe tableau

La syntaxe tableau pour `v-bind:style` permet d'appliquer plusieurs objets de style à un même élément.

```
<div v-bind:style="[baseStyles, overridingStyles]"></div>
```

Filters et computedProps : simplifier l'écriture des templates.

Filters

Vue.js permet de définir des filtres qui peuvent être utilisés pour appliquer des formatages de textes courants.

Les filtres sont utilisables à deux endroits : **les interpolations à moustaches et les expressions de v-bind**

Les filtres doivent être ajoutés à la fin de l'expression JavaScript, indiqués par le symbole de la barre verticale :

```
<!-- dans les moustaches -->
{{ message | capitalize }}

<!-- dans les v-bind -->
<div v-bind:id="rawId | formatId"></div>
```

Vous pouvez définir les filtres locaux dans les options d'un composant :

```
filters: {
  capitalize: function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
  }
}
```

Ou définir un filtre globalement avant de créer l'instance de Vue :

```
Vue.filter('capitalize', function (value) {
  if (!value) return ''
  value = value.toString()
  return value.charAt(0).toUpperCase() + value.slice(1)
})

new Vue({
  // ...
})
```

Quand le filtre global a le même nom que le filtre local, le filtre local va être préféré.

Ci-dessous un exemple du filtre `capitalize` en action :

```
<div id="example_1" class="demo">
  <input type="text" v-model="message">
  <p>{{ message | capitalize }}</p>
</div>
```

```

<script>
new Vue({
  el: '#example_1',
  data: function () {
    return {
      message: 'john'
    }
  },
  filters: {
    capitalize: function (value) {
      if (!value) return ''
      value = value.toString()
      return value.charAt(0).toUpperCase() + value.slice(1)
    }
  }
})
</script>

```

La fonction de filtre reçoit toujours la valeur de l'expression (le résultat de la chaîne) comme premier argument. Dans cet exemple, la fonction de filtre `capitalize` va recevoir la valeur de `message` dans son argument.

Les filtres peuvent être chainés :

```
{{ message | filterA | filterB }}
```

Dans ce cas, `filterA`, défini avec un seul argument, va recevoir la valeur de `message`. Puis la fonction `filterB` va être appelée avec le résultat de `filterA` passé dans `filterB` en tant que simple argument.

Les filtres sont des fonctions JavaScript et peuvent donc recevoir des arguments :

```
{{ message | filterA('arg1', arg2) }}
```

Ici `filterA` est définie comme une fonction prenant trois arguments. La valeur de `message` va être passée en premier argument. La chaîne de caractères '`arg1`' sera passée au filtre `filterA` en tant que second argument, et la valeur de l'expression `arg2` sera évaluée et passée en tant que troisième argument.

Propriétés calculées

Les expressions dans le template sont très pratiques, mais elles sont uniquement destinées à des opérations simples.

Mettre trop de logique dans vos templates peut les rendre trop verbeux et difficiles à maintenir.

Par exemple :

```
<div id="example">
  {{ message.split('').reverse().join('') }}
</div>
```

C'est pourquoi vous devriez utiliser des **propriétés calculées** pour toute logique complexe.

Exemple basique

```
<div id="example" class="demo">
  <p>Message original : "{{ message }}"</p>
  <p>Message inversé calculé : "{{ reversedMessage }}"</p>
</div>
<script>
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Bonjour'
  },
  computed: {
    reversedMessage: function () {
      return this.message.split('').reverse().join('')
    }
  }
})
</script>
```

Ici, nous avons déclaré une propriété calculée `reversedMessage`.

La fonction que nous avons fournie sera utilisée comme une fonction accesseur (getter) pour la propriété `vm.reversedMessage` :

```
console.log(vm.reversedMessage) // => 'ruojnoB'
vm.message = 'Au revoir'
console.log(vm.reversedMessage) // => 'riover uA'
```

Vous pouvez ouvrir la console et jouer vous-même avec l'exemple de `vm`. La valeur de `vm.reversedMessage` dépend toujours de la valeur de `vm.message`.

Mise en cache dans `computed` vs `methods`

Vous avez peut-être remarqué que nous pouvons accomplir le même résultat en invoquant une méthode dans l'expression :

```
<p>Message inversé : "{{ reverseMessage() }}"</p>
```

```
// dans le composant
methods: {
    reverseMessage: function () {
        return this.message.split(' ').reverse().join(' ')
    }
}
```

Au lieu d'une propriété calculée, nous pouvons définir la même fonction en tant que méthode. Pour le résultat final, les deux approches sont en effet exactement les mêmes.

Cependant, la différence est que **les propriétés déclarées dans computed sont mises en cache selon leurs dépendances**.

Une propriété calculée sera réévaluée uniquement quand certaines de ses dépendances auront changé. Cela signifie que tant que `message` n'a pas changé, les multiples accès à la propriété calculée `reversedMessage` retourneront immédiatement le résultat précédemment calculé sans avoir à réexécuter la fonction.

Cela signifie également que la propriété calculée suivante ne sera jamais mise à jour, parce que `Date.now()` n'est pas une dépendance réactive :

```
computed: {
    now: function () {
        return Date.now()
    }
}
```

En comparaison, une invocation de méthode exécutera **toujours** la fonction à chaque fois que se produira un redéclenchement du rendu.

Propriétés calculées vs observées

Vue fournit une façon plus générique d'observer et de réagir aux changements de données sur une instance de Vue : **les propriétés watch**.

Quand vous avez des données qu'il faut changer selon d'autres données, il est tentant d'abuser de `watch`.

Toutefois, il est souvent préférable d'utiliser une propriété calculée et non une fonction de rappel impérative comme `watch`. Considérez cet exemple :

```
<div id="demo">{{ fullName }}</div>
```

```

var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar',
    fullName: 'Foo Bar'
  },
  watch: {
    firstName: function (val) {
      this.fullName = val + ' ' + this.lastName
    },
    lastName: function (val) {
      this.fullName = this.firstName + ' ' + val
    }
  }
})

```

Le code ci-dessus est impératif et répétitif. Comparez-le avec une version de propriété calculée :

```

var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar'
  },
  computed: {
    fullName: function () {
      return this.firstName + ' ' + this.lastName
    }
  }
})

```

Mutateur calculé

Les propriétés calculées ont par défaut uniquement un accesseur, mais vous pouvez également fournir un mutateur (setter) quand vous en avez besoin :

```

// ...
computed: {
  fullName: {
    // accesseur
    get: function () {
      return this.firstName + ' ' + this.lastName
    },
    // mutateur
    set: function (newValue) {
      var names = newValue.split(' ')
      this.firstName = names[0]
      this.lastName = names[names.length - 1]
    }
  }
}

```

```
    }
}
// ...
```

Maintenant, lorsque vous exécutez `vm.fullName = 'John Doe'`, le mutateur sera invoqué, `vm.firstName` et `vm.lastName` seront mis à jour en conséquence.

Observateurs

Bien que les propriétés calculées soient plus appropriées dans la plupart des cas, parfois un observateur personnalisé est nécessaire.

C'est pour cela que Vue fournit une façon plus générique de réagir aux changements de données à travers l'option `watch`.

Ceci est très utile lorsque vous souhaitez exécuter des opérations asynchrones ou couteuses en réponse à des données changeantes.

Par exemple :

```
<div id="watch-example" class="demo">
  <p>
    Posez votre question (réponse par Oui ou Non) :
    <input v-model="question">
  </p>
  <p>{{ answer }}</p>
</div>
<script src="https://cdn.jsdelivr.net/npm/axios@0.12.0/dist/axios.min.js">
</script>
<script src="https://cdn.jsdelivr.net/npm/lodash@4.13.1/lodash.min.js">
</script>
<script>
var watchExampleVM = new Vue({
  el: '#watch-example',
  data: {
    question: '',
    answer: 'Je ne peux pas vous donner une réponse avant que vous ne posiez une question !'
  },
  watch: {
    question: function (newQuestion, oldQuestion) {
      this.answer = "J'attends que vous arrêtez de taper..."
      this.debouncedGetAnswer()
    }
  },
  created: function () {
    this.debouncedGetAnswer = _.debounce(this.getAnswer, 500)
  },
  methods: {
    getAnswer: function () {
```

```

        if (this.question.indexOf('?') === -1) {
            this.answer = "Les questions contiennent généralement un point
d'interrogation. ;-)"
            return
        }
        this.answer = 'Je réfléchis...'
        var vm = this
        axios.get('https://yesno.wtf/api')
            .then(function (response) {
                vm.answer = _.capitalize(response.data.answer)
            })
            .catch(function (error) {
                vm.answer = "Erreur ! Impossible d'accéder à l'API." + error
            })
    }
}
)
</script>

```

Dans ce cas, l'utilisation de l'option `watch` nous permet d'effectuer une opération asynchrone (accéder à une API), de limiter la fréquence d'exécution de cette opération et de définir des états intermédiaires jusqu'à ce que nous obtenions une réponse finale. Rien de tout cela ne serait possible avec une propriété calculée.

En plus de l'option `watch`, vous pouvez aussi utiliser l'API `vm.$watch`.

Templates vs render methods & JSX.

Vue vous recommande l'utilisation de templates pour construire votre HTML dans la grande majorité des cas. Il y a cependant des situations où vous aurez réellement besoin de toute la puissance programmatique de JavaScript.

C'est là que vous pouvez utiliser les **fonctions de rendu**, une alternative aux templates qui est plus proche du compilateur.

Examinons un exemple simple où une fonction `render` serait plus pratique. Imaginons que nous souhaitons générer des titres avec une ancre :

```
<h1>
  <a name="hello-world" href="#hello-world">
    Hello world !
  </a>
</h1>
```

Pour le HTML ci-dessus, vous décidez d'utiliser cette interface de composant :

```
<anchored-heading :level="1">Hello world!</anchored-heading>
```

Quand vous commencez avec un composant se basant sur la prop `level` pour simplement générer des niveaux de titre, vous arrivez rapidement à cela :

```
<script type="text/x-template" id="anchored-heading-template">
  <h1 v-if="level === 1">
    <slot></slot>
  </h1>
  <h2 v-else-if="level === 2">
    <slot></slot>
  </h2>
  <h3 v-else-if="level === 3">
    <slot></slot>
  </h3>
  <h4 v-else-if="level === 4">
    <slot></slot>
  </h4>
  <h5 v-else-if="level === 5">
    <slot></slot>
  </h5>
  <h6 v-else-if="level === 6">
    <slot></slot>
  </h6>
</script>
```

```

Vue.component('anchored-heading', {
  template: '#anchored-heading-template',
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})

```

Ce template ne semble pas génial. Il n'est pas uniquement verbeux, il duplique `<slot></slot>` dans tous les niveaux de titre et nous allons devoir refaire la même chose quand nous ajouterons l'élément ancre.

Alors que les templates fonctionnent bien pour la plupart des composants, il est clair que celui-là n'est pas l'un d'entre eux. Aussi essayons de le réécrire avec une fonction `render` :

```

Vue.component('anchored-heading', {
  render: function (createElement) {
    return createElement(
      'h' + this.level, // nom de balise
      this.$slots.default // tableau des enfants
    ),
    props: {
      level: {
        type: Number,
        required: true
      }
    }
})

```

C'est bien plus simple ! Le code est plus court mais demande une plus grande familiarité avec les propriétés d'une instance de Vue. Dans ce cas, vous devez savoir que lorsque vous passez des enfants sans une directive `v-slot` dans un composant, comme le `Hello world !` à l'intérieur de `anchored-heading`, ces enfants sont stockés dans l'instance du composant via la propriété `$slots.default`.

Nœuds, arbres, et DOM virtuel

Avant de rentrer en profondeur dans les fonctions de rendu, il est important de savoir comment un navigateur fonctionne. Prenons cet HTML comme exemple :

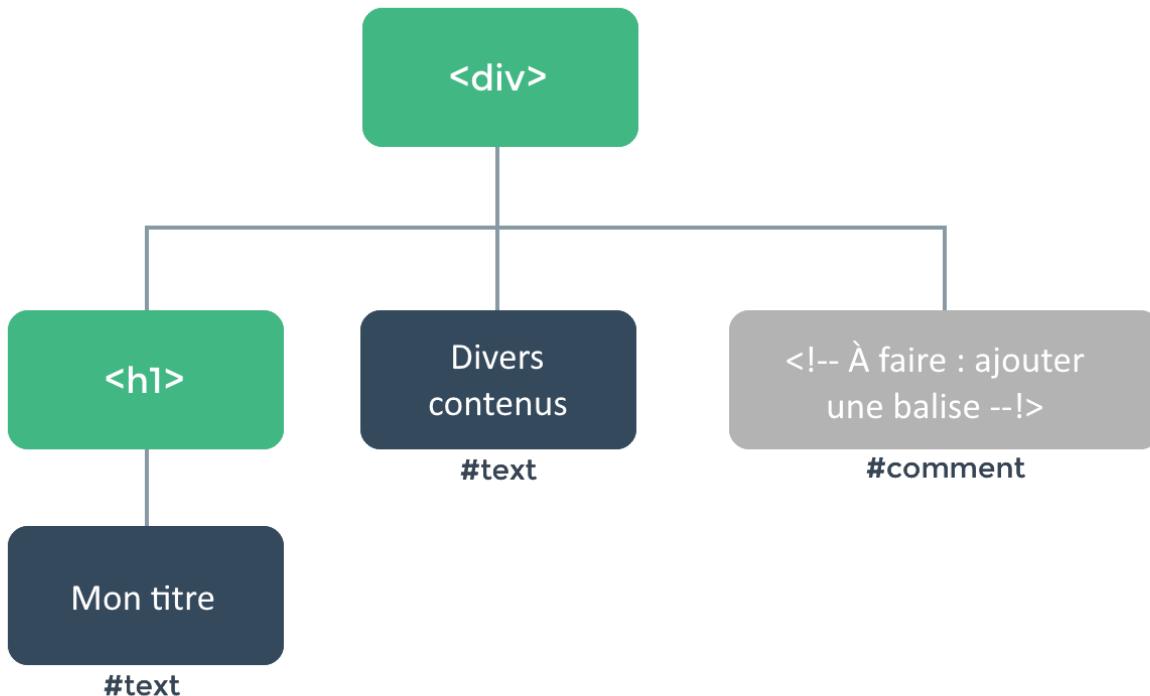
```

<div>
  <h1>Mon titre</h1>
  Divers contenus
  <!-- À faire : ajouter une balise -->
</div>

```

Quand votre navigateur lit ce code, il construit un [arbre de nœud de DOM](#) pour l'aider à garder une trace de tout, comme vous ferriez un arbre généalogique pour garder une trace de votre famille étendue.

L'arbre des nœuds du DOM pour le HTML ci-dessus ressemblerait à cela :



Chaque élément est un nœud. Chaque morceau de texte est un nœud. Même les commentaires sont des nœuds ! Un nœud est juste un morceau de la page. Et comme dans un arbre généalogique, chaque nœud peut avoir des enfants (c.-à-d. que chaque morceau peut contenir d'autres morceaux).

Mettre à jour tous ces nœuds efficacement peut être difficile, mais heureusement, vous n'avez jamais à le faire manuellement. Vous avez juste à dire à Vue quel HTML vous voulez pour votre page dans un template :

```
<h1>{{ blogTitle }}</h1>
```

Ou quelle fonction de rendu :

```
render: function (createElement) {
  return createElement('h1', this.blogTitle)
}
```

Et dans les deux cas, Vue va automatiquement garder la page à jour, même quand **blogTitle** change.

DOM virtuel

Vue arrive à cela grâce à la construction d'un **DOM virtuel** pour garder les traces des changements qui doivent être faits sur le vrai DOM. Prêtons attention à cette ligne :

```
return createElement('h1', this.blogTitle)
```

Qu'est-ce que `createElement` retourne exactement ? Ce n'est pas *réellement* un vrai élément de DOM. Cela pourrait être nommé plus justement `createNodeDescription`, car il contient des informations décrivant à Vue quelle sorte de rendu de nœud il va falloir faire sur la page en incluant les descriptions des nœuds enfants. Nous appelons cette description de nœud un « nœud virtuel », usuellement abrégé en **VNode** (pour « virtual node »). « DOM virtuel » est le nom de l'arbre des VNodes construits par un arbre de composants Vue.

Arguments de `createElement`

La seconde chose à laquelle vous allez devoir vous familiariser est la manière d'utiliser les fonctionnalités des templates avec la fonction `createElement`. Voici les arguments que la fonction `createElement` accepte :

```
// @returns {VNode}
createElement(
  // {String | Object | Function}
  // Un nom de balise HTML, des options de composant, ou une fonction
  // asynchrone retournant l'un des deux. Requis.
  'div',
  // {Object}
  // Un objet de données correspondant aux attributs
  // que vous souhaitez utiliser dans le template. Optionnel.
  {
    // (vu en détail dans la prochaine section)
  },
  // {String | Array}
  // Des VNodes enfants, construit en utilisant `createElement()`,
  // ou en utilisant des chaînes de caractères pour créer des 'text VNodes'.
  // Optionnel.
  [
    'Some text comes first.',
    createElement('h1', 'A headline'),
    createElement(MyComponent, {
      props: {
        someProp: 'foobar'
      }
    })
  ]
)
```

Objet de données dans le détail

Une chose est à noter : de la même manière que `v-bind:class` et `v-bind:style` ont un traitement spécial dans les templates, ils ont leurs propres champs dans les objets de données VNode. Cet objet vous permet également d'insérer des attributs HTML normaux ainsi que des propriétés du DOM comme `innerHTML` (cela remplace la directive `v-html`) :

```

{
  // Même API que `v-bind:class`, acceptant autant
  // une chaîne de caractères, un objet, ou un tableau d'objets.
  class: {
    foo: true,
    bar: false
  },
  // Même API que `v-bind:style`, acceptant autant
  // une chaîne de caractères, un objet, ou un tableau d'objets.
  style: {
    color: 'red',
    fontSize: '14px'
  },
  // Attributs HTML normaux
  attrs: {
    id: 'foo'
  },
  // Props des composants
  props: {
    myProp: 'bar'
  },
  // Propriétés du DOM
  domProps: {
    innerHTML: 'baz'
  },
  // La gestion d'événements est regroupée sous `on` cependant
  // les modificateurs comme `v-on:keyup.enter` ne sont pas
  // supportés. Vous devez vérifier manuellement le code de touche
  // dans le gestionnaire à la place.
  on: {
    click: this.clickHandler
  },
  // Pour les composants seulement. Vous permet d'écouter les
  // événements natifs, plutôt que ceux émis par
  // le composant via `vm.$emit`.
  nativeOn: {
    click: this.nativeClickHandler
  },
  // Directives personnalisées. Notez que la valeur `oldValue`
  // de la liaison ne peut pas être affectée, car Vue la conserve
  // pour vous.
  directives: [
    {
      name: 'my-custom-directive',
      value: '2',
      expression: '1 + 1',
      arg: 'foo',
      modifiers: {
        bar: true
      }
    }
  ],
  // Slots internes sous la forme
  // { name: props => VNode | Array<VNode> }
}

```

```

scopedSlots: {
  default: props => createElement('span', props.text)
},
// Le nom du slot, si ce composant est
// l'enfant d'un autre composant
slot: 'name-of-slot',
// Autres propriétés spéciales de premier niveau
key: 'myKey',
ref: 'myRef',
// Si vous appliquez le même nom de `ref` à plusieurs
// éléments dans la fonction de rendu. Cela va faire de `$refs.myRef` un
// tableau
refInFor: true
}

```

Exemple complet

Avec toutes ces informations, nous pouvons finir le composant que nous avons commencé :

```

var getChildrenTextContent = function (children) {
  return children.map(function (node) {
    return node.children
      ? getChildrenTextContent(node.children)
      : node.text
  }).join('')
}

Vue.component('anchored-heading', {
  render: function (createElement) {
    // créer un identifiant avec la kebab-case
    var headingId = getChildrenTextContent(this.$slots.default)
      .toLowerCase()
      .replace(/\W+/g, '-')
      .replace(/(^-|-$)/g, '')
  }
  return createElement(
    'h' + this.level,
    [
      createElement('a', {
        attrs: {
          name: headingId,
          href: '#' + headingId
        }
      }, this.$slots.default)
    ]
  )
},
props: {
  level: {
    type: Number,
    required: true
  }
}

```

```
    })
```

Contraintes

Les VNodes doivent être uniques

Tous les VNodes dans l'arbre des composants doivent être uniques. Cela signifie que la fonction de rendu suivante est invalide :

```
render: function (createElement) {
  var myParagraphVNode = createElement('p', 'hi')
  return createElement('div', [
    // Aïe – VNodes dupliqués !
    myParagraphVNode, myParagraphVNode
  ])
}
```

Si vous souhaitez réellement dupliquer le même élément/composant plusieurs fois, vous pouvez le faire avec une fonction fabrique. Par exemple, la fonction de rendu suivante est parfaitement valide pour faire le rendu de 20 paragraphes identiques :

```
render: function (createElement) {
  return createElement('div',
    Array.apply(null, { length: 20 }).map(function () {
      return createElement('p', 'hi')
    })
  )
}
```

Remplacer les fonctionnalités de template en pur JavaScript

v-if et v-for

Partout où quelque chose peut être accompli simplement en JavaScript, les fonctions de rendu de Vue ne fournissent pas d'alternative propriétaire. Par exemple, un template utilisant **v-if** et **v-for** :

```
<ul v-if="items.length">
  <li v-for="item in items">{{ item.name }}</li>
</ul>
<p v-else>Aucun élément trouvé.</p>
```

Cela pourrait être réécrit avec les **if/else** et **map** du JavaScript dans une fonction de rendu

```

props: ['items'],
render: function (createElement) {
  if (this.items.length) {
    return createElement('ul', this.items.map(function (item) {
      return createElement('li', item.name)
    }))
  } else {
    return createElement('p', 'Aucun élément trouvé.')
  }
}

```

v-model

Il n'y a pas d'équivalent à **v-model** dans les fonctions de rendu. Vous devez implémenter la logique vous-même :

```

props: ['value'],
render: function (createElement) {
  var self = this
  return createElement('input', {
    domProps: {
      value: self.value
    },
    on: {
      input: function (event) {
        self.$emit('input', event.target.value)
      }
    }
  })
}

```

C'est le prix à payer pour travailler au plus bas niveau, mais cela vous donne un meilleur contrôle sur le détail des interactions comparé à **v-model**.

Modificateurs d'évènement et de code de touche

Pour les modificateurs d'évènement **.passive**, **.capture** et **.once**, Vue offre des préfixes pouvant être utilisés dans **on**:

Modificateur(s)	Préfixes
.passive	&
.capture	!
.once	~
.capture.once ou .once.capture	~!

Par exemple :

```
on: {
  '!click': this.doThisInCapturingMode,
  '~keyup': this.doThisOnce,
  '~~!mouseover': this.doThisOnceInCapturingMode
}
```

Pour tous les autres modificateurs d'évènement et de code de touche, aucun préfixe propriétaire n'est nécessaire car il suffit d'utiliser des méthodes d'évènement dans le gestionnaire :

Modificateur(s)	Équivalence dans le gestionnaire
.stop	event.stopPropagation()
.prevent	event.preventDefault()
.self	if (event.target !== event.currentTarget) return
Touches : .enter, .13	if (event.keyCode !== 13) return (changez 13 en un autre code de touche pour les autres modificateurs de code de touche)

Modificateurs de

Clés : if (!event.ctrlKey) return (changez respectivement **ctrlKey** par **altKey**,
.ctrl, .alt, shiftKey, ou metaKey)
.shift, .meta

Voici un exemple avec tous ces modificateurs utilisés ensemble :

```
on: {
  keyup: function (event) {
    // Annuler si l'élément qui émet l'évènement n'est pas
    // l'élément auquel l'évènement est lié
    if (event.target !== event.currentTarget) return
    // Annuler si la touche relâchée n'est pas la touche
    // Entrée (13) et si la touche `shift` n'est pas maintenue
    // en même temps
    if (!event.shiftKey || event.keyCode !== 13) return
    // Arrêter la propagation d'évènement
    event.stopPropagation()
    // Éviter la gestion par défaut de cet évènement pour cet élément
    event.preventDefault()
    // ...
  }
}
```

Slots

Vous pouvez accéder aux contenus des slots statiques en tant que tableaux de VNodes depuis **this.\$slots** :

```

render: function (createElement) {
  // `<div><slot></slot></div>`
  return createElement('div', this.$slots.default)
}

```

Et accéder aux slots de portée en tant que fonctions qui retournent des VNodes via `this.$scopedSlots` :

```

props: ['message'],
render: function (createElement) {
  // `<div><slot :text="message"></slot></div>`
  return createElement('div', [
    this.$scopedSlots.default({
      text: this.message
    })
  ])
}

```

Pour passer des slots internes à un composant enfant en utilisant des fonctions de rendu, utilisez la propriété `scopedSlots` dans les données du VNode :

```

render: function (createElement) {
  return createElement('div', [
    createElement('child', {
      // passer `scopedSlots` dans l'objet de données
      // in the form of { name: props => VNode | Array<VNode> }
      scopedSlots: {
        default: function (props) {
          return createElement('span', props.text)
        }
      }
    })
  ])
}

```

JSX

Si vous écrivez beaucoup de fonctions `render`, cela pourra vous sembler fatigant d'écrire des choses comme ça :

```

createElement(
  'anchored-heading', {
    props: {
      level: 1
    }
  }, [
    createElement('span', 'Hello'),
    ' world!'
)

```

```
]  
)
```

Et d'autant plus quand la version template est vraiment simple en comparaison :

```
<anchored-heading :level="1">  
  <span>Hello</span> world !  
</anchored-heading>
```

C'est pourquoi il y a un [plugin Babel](#) pour utiliser JSX avec Vue, nous permettant l'utilisation d'une syntaxe plus proche de celle des templates :

```
import AnchoredHeading from './AnchoredHeading.vue'  
  
new Vue({  
  el: '#demo',  
  render: function (h) {  
    return (  
      <AnchoredHeading level={1}>  
        <span>Hello</span> world !  
      </AnchoredHeading>  
    )  
  }  
)
```

Utiliser `h` comme alias de `createElement` est une convention courante que vous verrez dans l'écosystème Vue et qui est en fait requise pour JSX. À partir de la [version 3.4.0](<https://github.com/vuejs/babel-plugin-transform-vue-jsx#h-auto-injection>) du plugin Babel pour Vue, nous injectons automatiquement `const h = this.\$createElement` dans n'importe quelle méthode ou accesseur (pas dans les fonctions ou fonctions fléchées), déclaré avec la syntaxe ES2015 qui a du JSX, vous pouvez ainsi oublier le paramètre `(h)`. Avec les versions précédentes du plugin, si `h` n'est pas disponible dans votre portée courante, votre application va lever une erreur.

Pour plus d'informations sur comment utiliser JSX dans du JavaScript, référez-vous à la [documentation d'utilisation](#).

Composants fonctionnels.

Le composant de titre ancré que nous avons créé plus tôt était relativement simple. Il ne gère aucun état, n'observe aucun état qu'on lui passe, et il n'a pas de méthodes de cycle de vie. Non, ce n'est qu'une simple fonction avec quelques props.

Dans des cas comme celui-ci, nous pouvons marquer les composants comme **functional**, ce qui signifie qu'ils sont sans état (« stateless » c.-à-d. sans **data réactive**) et sans instance (« instanceless » c.-à-d. sans contexte **this**). Un **composant fonctionnel** ressemble à ça :

```
Vue.component('my-component', {
  functional: true,
  // Les props sont optionnelles
  props: {
    // ...
  },
  // Pour compenser le manque d'instance,
  // nous pouvons maintenant fournir en second argument un contexte.
  render: function (createElement, context) {
    // ...
  }
})
```

Note : dans les versions avant 2.3.0, l'option **props** est requise si vous souhaitez accepter des props dans un composant fonctionnel. Dans les versions 2.3.0+ vous pouvez omettre l'option **props** et tous les attributs trouvés dans le nœud composant seront implicitement extraits comme des props.

La référence sera `HTMLElement` quand elle sera utilisé avec des composants fonctionnels car elles sont sans état et sans instance.

Dans la 2.5.0+, si vous utilisez les **composants monofichiers**, les templates fonctionnels basés sur les composants peuvent être déclarés avec :

```
<template functional>
</template>
```

Tout ce dont le composant a besoin est passé dans l'objet **context**, qui est un objet contenant :

- **props** : un objet avec les props fournies,
- **children** : un tableau de VNode enfants,
- **slots** : une fonction retournant un objet de slots,
- **scopedSlots** : (2.6.0+) un objet qui expose les portées passées dans les slots. Expose également les slots normaux comme des fonctions,
- **data** : l'**objet de données** (**data**) complet passé au composant en tant que second argument de **createElement**,
- **parent** : une référence au composant parent,
- **listeners** : (2.3.0+) un objet contenant les écouteurs d'évènement enregistrés dans le parent. C'est un simple alias de **data.on**,

- **injections** : (2.3.0+) si vous utilisez l'option **inject**, cela va contenir les injections résolues.

Après avoir ajouté **functional: true**, mettre à jour la fonction de rendu de notre composant de titres avec ancre va simplement nécessiter d'ajouter l'argument **context**, en remplaçant **this.\$slots.default** par **context.children**, puis **this.level** par **context.props.level**.

Puisque les composants fonctionnels ne sont que des fonctions, leur rendu est plus rapide.

Ils sont également très utiles en tant que composants enveloppes. Par exemple, quand vous avez besoin de :

- Programmatiquement choisir un composant parmi plusieurs autres composants pour de la délégation ou
- Manipuler les enfants, props, ou données avant de les passer au composant enfant.

Voici un exemple d'un composant **smart-list** qui délègue à des composants plus spécifiques en fonction des props qui lui sont passées :

```
var EmptyList = { /* ... */ }
var TableList = { /* ... */ }
var OrderedList = { /* ... */ }
var UnorderedList = { /* ... */ }

Vue.component('smart-list', {
  functional: true,
  props: {
    items: {
      type: Array,
      required: true
    },
    isOrdered: Boolean
  },
  render: function (createElement, context) {
    function appropriateListComponent () {
      var items = context.props.items

      if (items.length === 0) return EmptyList
      if (typeof items[0] === 'object') return TableList
      if (context.props.isOrdered) return OrderedList

      return UnorderedList
    }

    return createElement(
      appropriateListComponent(),
      context.data,
      context.children
    )
  }
})
```

Passer des attributs et évènements aux éléments / composants enfants

Sur les composants normaux, les attributs qui ne sont pas définis comme props sont automatiquement ajoutés à l'élément racine du composant en remplaçant ou [en étant intelligemment rajouté sur](#) n'importe quel attribut existant avec le même nom.

Cependant les composants fonctionnels vous demande de définir explicitement ce comportement :

```
Vue.component('my-functional-button', {
  functional: true,
  render: function (createElement, context) {
    // Passer de manière transparente n'importe quel attribut, écouteur
    // d'évènement, enfant, etc.
    return createElement('button', context.data, context.children)
  }
})
```

En passant `context.data` en tant que second paramètre à `createElement`, nous transferrons à l'enfant racine n'importe quel attribut ou écouteur d'évènement utilisé sur `my-functional-button`. C'est même tellement transparent que les évènements n'ont même pas besoin du modificateur `.native`.

Si vous utilisez un composant fonctionnel basé sur un template, vous devrez aussi ajouter manuellement les attributs et les écouteurs. Puisque nous avons accès au contenus individuels du contexte, nous pouvons utiliser `data.attrs` en le passant avec n'importe quel attribut HTML ou `listeners` (*l'alias pour `data.on`*) en le passant avec n'importe quel écouteur d'évènement.

```
<template functional>
  <button
    class="btn btn-primary"
    v-bind="data.attrs"
    v-on="listeners"
  >
    <slot/>
  </button>
</template>
```

slots() vs. children

Vous vous demandez peut-être l'utilité d'avoir `slot()` et `children` en même temps. `slots().default` n'est-il pas la même chose que `children` ? Dans la majorité des cas, oui. Mais que faire si vous avez un composant fonctionnel avec les enfants suivants ?

```
<my-functional-component>
  <p v-slot:foo>
    premier
  </p>
  <p>second</p>
</my-functional-component>
```

Pour ce composant, `children` va vous donner les deux paragraphes, `slots().default` ne vous donnera que le second, et `slots().foo` ne vous donnera que le premier. Avoir le choix entre `children` et `slots()` vous permet donc de choisir ce que le composant sait à propos du système de slot ou si vous délégez peut-être la responsabilité à un autre composant en passant simplement `children`.

Animations & transitions : améliorer l'expérience utilisateur.

Vue d'ensemble

Vue fournit plusieurs façons d'appliquer des effets de transition lorsque des éléments sont insérés, mis à jour ou supprimés du DOM. Cela inclut des outils pour :

- appliquer automatiquement des classes pour les transitions et les animations CSS
- intégrer des bibliothèques d'animation CSS tierces, comme Animate.css
- utiliser JavaScript pour manipuler directement le DOM durant les hooks de transition
- intégrer des bibliothèques d'animation JavaScript tierces, comme Velocity.js

Sur cette page, nous ne traiterons que des transitions entrantes, sortantes et de liste, mais vous pouvez consulter la section suivante pour la [gestion des transitions d'état](#).

Transition d'éléments/composants simples

Vue fournit un composant conteneur **transition**, vous permettant d'ajouter des transitions entrantes/sortantes pour n'importe quel élément ou composant dans les contextes suivants :

- Le rendu conditionnel (en utilisant **v-if**)
- L'affichage conditionnel (en utilisant **v-show**)
- Les composants dynamiques
- Les noeuds racines de composant

Voilà à quoi ressemble un exemple très simple en action :

```
<div id="demo">
  <button v-on:click="show = !show">
    Permuter
  </button>
  <transition name="demo-transition">
    <p v-if="show">bonjour</p>
  </transition>
</div>
<script>
new Vue({
  el: '#demo',
  data: {
    show: true
  }
})
</script>
<style>
.demo-transition-enter-active, .demo-transition-leave-active {
  transition: opacity .5s
}
.demo-transition-enter, .demo-transition-leave-to {
  opacity: 0
}
</style>
```

Quand un élément, encapsulé dans un composant **transition**, est inséré ou enlevé, voilà ce qui arrive :

1. Vue recherchera automatiquement si l'élément cible a des transitions CSS ou des animations appliquées. Si c'est le cas, les classes de transition CSS seront ajoutées/supprimées aux moments appropriés.
2. Si le composant de transition possède des [hooks JavaScript](#), ces hooks seront appelés aux moments appropriés.
3. Si aucune transition/animation CSS n'est détectée et qu'aucun hook JavaScript n'est fourni, les opérations du DOM pour l'insertion et/ou la suppression seront exécutées immédiatement à la frame suivante (Remarque : il s'agit d'une frame d'animation du navigateur, différente du concept de **nextTick**).

Classes de transition

Il y a six classes appliquées pour les transitions entrantes/sortantes.

1. **v-enter**: c'est l'état de départ pour **enter**. Il est ajouté avant que l'élément soit inséré, il est supprimé une fois que l'élément est inséré.
2. **v-enter-active**: c'est l'état actif pour **enter**. Il est appliqué pendant toute la phase **enter**. Il est ajouté avant que l'élément soit inséré, il est supprimé lorsque la transition/animation est terminée. Cette classe peut être utilisée pour définir la durée, le retard et la courbe d'accélération pour la transition entrante.
3. **v-enter-to**: **seulement disponible pour les versions 2.1.8+**. C'est l'état de fin pour **enter**. Il est ajouté une fois que l'élément est inséré (au même moment que **v-enter** est supprimé), il est supprimé lorsque la transition/animation est terminée.
4. **v-leave**: c'est l'état de départ pour **leave**. Il est ajouté dès qu'une transition sortante est déclenchée, il est supprimé après une frame.
5. **v-leave-active**: c'est l'état actif pour **leave**. Il est appliqué pendant toute la phase **leave**. Il est ajouté dès qu'une transition sortante est déclenchée, il est supprimé lorsque la transition/animation est terminée. Cette classe peut être utilisée pour définir la durée, le retard et la courbe d'accélération pour la transition de sortie.
6. **v-leave-to**: **seulement disponible pour les versions 2.1.8+**. C'est l'état de fin pour **leave**. Il est ajouté après que la transition sortante soit déclenchée (au même moment que **v-leave** est supprimé), il est supprimé lorsque la transition/animation est terminée.

Diagramme de transition

Chacune de ces classes sera préfixée avec le nom de la transition. Ici le préfixe **v-** est celui par défaut lorsque vous utilisez l'élément `<transition>` sans nom. Si vous utilisez par exemple `<transition name="ma-transition">`, alors la classe **v-enter** sera nommée **ma-transition-enter**.

v-enter-active et **v-leave-active** vous donnent la possibilité de spécifier des courbes d'accélération pour les transitions entrantes/sortantes, ce que nous verrons en exemple dans la section suivante.

Transitions CSS

L'un des types de transition les plus courants utilise les transitions CSS. Voici un exemple simple :

```

<div id="example-1" class="demo">
  <button @click="show = !show">
    Permuter
  </button>
  <transition name="slide-fade">
    <p v-if="show">bonjour</p>
  </transition>
</div>
<script>
new Vue({
  el: '#example-1',
  data: {
    show: true
  }
})
</script>
<style>
.slide-fade-enter-active {
  transition: all .3s ease;
}
.slide-fade-leave-active {
  transition: all .8s cubic-bezier(1.0, 0.5, 0.8, 1.0);
}
.slide-fade-enter, .slide-fade-leave-to {
  transform: translateX(10px);
  opacity: 0;
}
</style>

```

CSS Animations

Les animations CSS sont appliquées de la même manière que les transitions CSS, la différence étant que `v-enter` n'est pas supprimé immédiatement après l'insertion de l'élément, mais par un évènement `animationend`.

Voici un exemple, en supprimant les règles CSS préfixées pour des raisons de concision :

```

<div id="example-2" class="demo">
  <button @click="show = !show">Permuter l'affichage</button>
  <transition name="bounce">
    <p v-show="show">Lorem ipsum dolor sit amet, consectetur adipiscing
    elit. Mauris facilisis enim libero, at lacinia diam fermentum id.
    Pellentesque habitant morbi tristique senectus et netus.</p>
  </transition>
</div>

<style>
.bounce-enter-active {
  -webkit-animation: bounce-in .5s;
  animation: bounce-in .5s;
}

```

```

.bounce-leave-active {
  -webkit-animation: bounce-in .5s reverse;
  animation: bounce-in .5s reverse;
}
@keyframes bounce-in {
  0% {
    -webkit-transform: scale(0);
    transform: scale(0);
  }
  50% {
    -webkit-transform: scale(1.5);
    transform: scale(1.5);
  }
  100% {
    -webkit-transform: scale(1);
    transform: scale(1);
  }
}
@-webkit-keyframes bounce-in {
  0% {
    -webkit-transform: scale(0);
    transform: scale(0);
  }
  50% {
    -webkit-transform: scale(1.5);
    transform: scale(1.5);
  }
  100% {
    -webkit-transform: scale(1);
    transform: scale(1);
  }
}

```

</style>

<script>

```

new Vue({
  el: '#example-2',
  data: {
    show: true
  }
})

```

</script>

Classes de transition personnalisées

Vous pouvez également spécifier des classes de transition personnalisées en fournissant les attributs suivants :

- `enter-class`
- `enter-active-class`
- `enter-to-class` (2.1.8+)
- `leave-class`
- `leave-active-class`
- `leave-to-class` (2.1.8+)

Celles-ci remplacent les noms de classes habituelles. C'est surtout utile quand vous voulez combiner le système de transition de Vue avec une bibliothèque d'animation CSS existante, comme [Animate.css](#).

Voici un exemple :

```
<link href="https://cdn.jsdelivr.net/npm/animate.css@3.5.1"
      rel="stylesheet" type="text/css">
<div id="example-3" class="demo">
  <button @click="show = !show">
    Permuter
  </button>
  <transition
    name="custom-classes-transition"
    enter-active-class="animated tada"
    leave-active-class="animated bounceOutRight">
    <p v-if="show">bonjour</p>
  </transition>
</div>
<script>
new Vue({
  el: '#example-3',
  data: {
    show: true
  }
})
</script>
```

Utilisation simultanée des transitions et animations

Vue a besoin d'attacher des écouteurs d'évènement pour savoir quand une transition est terminée. Cela peut être **transitionend** ou **animationend**, selon le type de règles CSS appliquées. Si vous utilisez seulement l'une ou l'autre, Vue peut automatiquement déterminer le type correct.

Toutefois, dans certains cas, vous pouvez les avoir tous les deux sur le même élément, par exemple avoir une animation CSS déclenchée par Vue, ainsi qu'un effet de transition CSS lors du survol. Dans ces cas, vous devez explicitement déclarer le type dont vous voulez que Vue se soucie dans un attribut **type**, avec une valeur à **animation** ou **transition**.

Durées de transition explicites

Nouveau dans 2.2.0+

Dans la plupart des cas, Vue peut automatiquement déterminer quand la transition est terminée. Par défaut, Vue attend le premier évènement **transitionend** ou **animationend** sur l'élément de transition racine. Cependant, cela peut ne pas toujours être souhaité (par exemple, nous pouvons avoir une séquence de transition chorégraphiée où certains éléments internes imbriqués ont une transition retardée ou une durée de transition plus longue que l'élément de transition racine).

Dans ce cas, vous pouvez spécifier une durée de transition explicite (en millisecondes) en utilisant le prop `duration` sur le composant `<transition>` :

```
<transition :duration="1000">...</transition>
```

Vous pouvez également spécifier des valeurs séparées pour les durées de `enter` et `leave` :

```
<transition :duration="{ enter: 500, leave: 800 }">...</transition>
```

hooks JavaScript

Vous pouvez définir des hooks JavaScript dans les attributs :

```
<transition
  v-on:before-enter="beforeEnter"
  v-on:enter="enter"
  v-on:after-enter="afterEnter"
  v-on:enter-cancelled="enterCancelled"

  v-on:before-leave="beforeLeave"
  v-on:leave="leave"
  v-on:after-leave="afterLeave"
  v-on:leave-cancelled="leaveCancelled"
>
  <!-- ... -->
</transition>
```

```
// ...
methods: {
  // -----
  // EN ENTRANT
  // -----

  beforeEnter: function (el) {
    // ...
  },
  // la fonction de rappel done est facultative quand
  // utilisée en combinaison avec du CSS
  enter: function (el, done) {
    // ...
    done()
  },
  afterEnter: function (el) {
    // ...
  },
  enterCancelled: function (el) {
    // ...
  }
}
```

```

},
// -----
// EN SORTANT
// -----

beforeLeave: function (el) {
    // ...
},
// la fonction de rappel done est facultative quand
// utilisée en combinaison avec du CSS
leave: function (el, done) {
    // ...
    done()
},
afterLeave: function (el) {
    // ...
},
// leaveCancelled est uniquement disponible avec v-show
leaveCancelled: function (el) {
    // ...
}
}

```

Ces hooks peuvent être utilisés en combinaison avec des transitions/animations CSS ou sur eux-mêmes.

Lors de l'utilisation de transitions uniquement JavaScript, **les fonctions de rappel `done` sont obligatoires pour les hooks `enter` et `leave`**. Dans le cas contraire, elles seront appelées de façon synchrone et la transition se terminera immédiatement.

C'est aussi une bonne idée d'ajouter explicitement `v-bind:css="false"` pour les transitions uniquement JavaScript afin que Vue puisse ignorer la détection CSS. Cela empêche également les règles CSS d'interférer accidentellement avec la transition.

Maintenant, nous allons plonger dans un exemple. Voici une simple transition JavaScript à l'aide de Velocity.js :

```

<!--
Velocity fonctionne aussi bien que jQuery.animate et est
une excellente option pour les animations
-->

<div id="example-4" class="demo">
    <button @click="show = !show">
        Permuter
    </button>
    <transition
        v-on:before-enter="beforeEnter"
        v-on:enter="enter"
        v-on:leave="leave"
    >
        <p v-if="show">
            Démo
    
```

```

        </p>
    </transition>
</div>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"
"></script>
<script>
new Vue({
  el: '#example-4',
  data: {
    show: false
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
      el.style.transformOrigin = 'left'
    },
    enter: function (el, done) {
      Velocity(el, { opacity: 1, fontSize: '1.4em' }, { duration: 300 })
      Velocity(el, { fontSize: '1em' }, { complete: done })
    },
    leave: function (el, done) {
      Velocity(el, { translateX: '15px', rotateZ: '50deg' }, { duration:
600 })
      Velocity(el, { rotateZ: '100deg' }, { loop: 2 })
      Velocity(el, {
        rotateZ: '45deg',
        translateY: '30px',
        translateX: '30px',
        opacity: 0
      }, { complete: done })
    }
  }
})
</script>

```

Transitions sur le rendu initial

Si vous souhaitez également appliquer une transition sur le rendu initial d'un nœud, vous pouvez ajouter l'attribut `appear` :

```

<transition appear>
  <!-- ... -->
</transition>

```

Par défaut, cela utilisera les transitions spécifiées pour l'entrée et la sortie. Si vous le souhaitez, vous pouvez également spécifier des classes CSS personnalisées :

```

<transition
  appear

```

```

appear-class="custom-appear-class"
appear-to-class="custom-appear-to-class" (2.1.8+)
appear-active-class="custom-appear-active-class"
>
<!-- ... -->
</transition>

```

et des hooks JavaScript personnalisés :

```

<transition
  appear
  v-on:before-appear="customBeforeAppearHook"
  v-on:appear="customAppearHook"
  v-on:after-appear="customAfterAppearHook"
  v-on:appear-cancelled="customAppearCancelledHook"
>
<!-- ... -->
</transition>

```

Dans l'exemple ci-dessus, l'attribut `appear` ou le hook `v-on:appear` vont tous deux déclencher une transition d'apparition.

Transition entre éléments

Plus loin, nous parlons de [transition entre les composants](#), mais vous pouvez également faire une transition entre des éléments bruts en utilisant `v-if/v-else`. L'une des transitions les plus courantes sur deux éléments est entre un conteneur de liste et un message décrivant une liste vide :

```

<transition>
  <table v-if="items.length > 0">
    <!-- ... -->
  </table>
  <p v-else>Désolé, aucun élément trouvé.</p>
</transition>

```

Cela fonctionne bien, mais il y a une mise en garde à connaître :

Lors de la permutation entre des éléments qui ont **le même nom de balise**, vous devez indiquer à Vue qu'ils sont des éléments distincts en lui donnant des attributs `key` uniques. Sinon, le compilateur de Vue ne remplacera que le contenu de l'élément dans le but d'être efficace. Cependant, même si c'est techniquement inutile, **c'est considéré comme une bonne pratique de toujours avoir une clé pour chaque élément dans un composant**.

Par exemple :

```

<transition>
  <button v-if="isEditing" key="save">

```

```

        Sauver
    </button>
    <button v-else key="edit">
        Modifier
    </button>
</transition>
```

Dans ces cas, vous pouvez aussi utiliser l'attribut `key` pour effectuer une transition entre différents états du même élément. Au lieu d'utiliser `v-if` et `v-else`, l'exemple ci-dessus pourrait être réécrit ainsi :

```

<transition>
    <button v-bind:key="isEditing">
        {{ isEditing ? 'Sauver' : 'Modifier' }}
    </button>
</transition>
```

Il est effectivement possible de faire une transition entre un nombre indéfini d'éléments, soit en utilisant plusieurs `v-if` ou soit en liant un élément unique à une propriété dynamique. Par exemple :

```

<transition>
    <button v-if="docState === 'saved'" key="saved">
        Modifier
    </button>
    <button v-if="docState === 'edited'" key="edited">
        Sauver
    </button>
    <button v-if="docState === 'editing'" key="editing">
        Annuler
    </button>
</transition>
```

Qui pourrait aussi s'écrire ainsi :

```

<transition>
    <button v-bind:key="docState">
        {{ buttonMessage }}
    </button>
</transition>
```

```

// ...
computed: {
    buttonMessage: function () {
        switch (this.docState) {
            case 'saved': return 'Modifier'
            case 'edited': return 'Sauver'
            case 'editing': return 'Annuler'
```

```
    }
}
}
```

Modes de transition

Cependant, il existe encore un problème. Essayez de cliquer sur le bouton ci-dessous :

```
<div id="no-mode-demo" class="demo">
  <transition name="no-mode-fade">
    <button v-if="on" key="on" @click="on = false">
      on
    </button>
    <button v-else key="off" @click="on = true">
      off
    </button>
  </transition>
</div>
<script>
new Vue({
  el: '#no-mode-demo',
  data: {
    on: false
  }
})
</script>
<style>
.no-mode-fade-enter-active, .no-mode-fade-leave-active {
  transition: opacity .5s
}
.no-mode-fade-enter, .no-mode-fade-leave-active {
  opacity: 0
}
</style>
```

Comme c'est une transition entre le bouton « on » et le bouton « off », les deux boutons sont rendus (l'un est en transition sortante pendant que l'autre est en transition entrante). Il s'agit du comportement par défaut de `<transition>` (l'entrée et la sortie se font simultanément).

Parfois, cela fonctionne très bien, comme lorsque des éléments de transition sont absolument positionnés l'un sur l'autre :

```
<div id="no-mode-absolute-demo" class="demo">
  <div class="no-mode-absolute-demo-wrapper">
    <transition name="no-mode-absolute-fade">
      <button v-if="on" key="on" @click="on = false">
        on
      </button>
      <button v-else key="off" @click="on = true">
        off
      </button>
    </transition>
  </div>
</div>
```

```

        </button>
    </transition>
</div>
</div>
<script>
new Vue({
  el: '#no-mode-absolute-demo',
  data: {
    on: false
  }
})
</script>
<style>
.no-mode-absolute-demo-wrapper {
  position: relative;
  height: 18px;
}
.no-mode-absolute-demo-wrapper button {
  position: absolute;
}
.no-mode-absolute-fade-enter-active, .no-mode-absolute-fade-leave-active {
  transition: opacity .5s;
}
.no-mode-absolute-fade-enter, .no-mode-absolute-fade-leave-active {
  opacity: 0;
}
</style>

```

Et puis cela peut être interprété comme des transitions de diapositives.

```

<div id="no-mode-translate-demo" class="demo">
<div class="no-mode-translate-demo-wrapper">
  <transition name="no-mode-translate-fade">
    <button v-if="on" key="on" @click="on = false">
      on
    </button>
    <button v-else key="off" @click="on = true">
      off
    </button>
  </transition>
</div>
</div>
<script>
new Vue({
  el: '#no-mode-translate-demo',
  data: {
    on: false
  }
})
</script>
<style>
.no-mode-translate-demo-wrapper {
  position: relative;

```

```

height: 18px;
}
.no-mode-translate-demo-wrapper button {
  position: absolute;
}
.no-mode-translate-fade-enter-active, .no-mode-translate-fade-leave-active {
  transition: all 1s;
}
.no-mode-translate-fade-enter, .no-mode-translate-fade-leave-active {
  opacity: 0;
}
.no-mode-translate-fade-enter {
  transform: translateX(31px);
}
.no-mode-translate-fade-leave-active {
  transform: translateX(-31px);
}

```

Les transitions simultanées d'entrée et de sortie ne sont pas toujours désirées, donc Vue propose des **modes de transition** alternatifs :

- **in-out**: La transition entrante du nouvel élément s'effectue en premier, puis une fois terminée, déclenche la transition sortante de l'élément courant.
- **out-in**: La transition sortante de l'élément courant s'effectue en premier, puis une fois terminée, déclenche la transition entrante du nouvel élément.

Maintenant, mettons à jour la transition pour nos boutons on/off avec **out-in** :

```

<transition name="fade" mode="out-in">
  <!-- ... les boutons ... -->
</transition>

```

```

<div id="with-mode-demo" class="demo">
  <transition name="with-mode-fade" mode="out-in">
    <button v-if="on" key="on" @click="on = false">
      on
    </button>
    <button v-else key="off" @click="on = true">
      off
    </button>
  </transition>
</div>
<script>
new Vue({
  el: '#with-mode-demo',
  data: {
    on: false
  }
})</script>

```

```

        }
    })
</script>
<style>
.with-mode-fade-enter-active, .with-mode-fade-leave-active {
    transition: opacity .5s
}
.with-mode-fade-enter, .with-mode-fade-leave-active {
    opacity: 0
}
</style>

```

Avec l'ajout d'un simple attribut, nous avons corrigé cette transition originale sans devoir ajouter un style spécial.

Le mode `in-out` n'est pas utilisé aussi souvent, mais peut parfois être utile pour un effet de transition un peu différent. Essayons de le combiner avec la transition diapositive sur laquelle nous avons travaillé précédemment.

```

<div id="in-out-translate-demo" class="demo">
    <div class="in-out-translate-demo-wrapper">
        <transition name="in-out-translate-fade" mode="in-out">
            <button v-if="on" key="on" @click="on = false">
                on
            </button>
            <button v-else key="off" @click="on = true">
                off
            </button>
        </transition>
    </div>
</div>
<script>
new Vue({
    el: '#in-out-translate-demo',
    data: {
        on: false
    }
})
</script>
<style>
.in-out-translate-demo-wrapper {
    position: relative;
    height: 18px;
}
.in-out-translate-demo-wrapper button {
    position: absolute;
}
.in-out-translate-fade-enter-active, .in-out-translate-fade-leave-active {
    transition: all .5s;
}
.in-out-translate-fade-enter, .in-out-translate-fade-leave-active {
    opacity: 0;
}
.in-out-translate-fade-enter {
    opacity: 1;
}

```

```

    transform: translateX(31px);
}
.in-out-translate-fade-leave-active {
  transform: translateX(-31px);
}

```

Transition entre les composants

Faire une transition entre les composants est encore plus simple (nous n'avons même pas besoin de l'attribut **key**). Au lieu de cela, nous les enveloppons simplement d'un composant dynamique.

```

<transition name="component-fade" mode="out-in">
  <component v-bind:is="view"></component>
</transition>

```

```

<div id="transition-components-demo" class="demo">
  <input v-model="view" type="radio" value="v-a" id="a" name="view"><label
for="a">A</label>
  <input v-model="view" type="radio" value="v-b" id="b" name="view"><label
for="b">B</label>
  <transition name="component-fade" mode="out-in">
    <component v-bind:is="view"></component>
  </transition>
</div>
<style>
.component-fade-enter-active, .component-fade-leave-active {
  transition: opacity .3s ease;
}
.component-fade-enter, .component-fade-leave-to {
  opacity: 0;
}
</style>
<script>
new Vue({
  el: '#transition-components-demo',
  data: {
    view: 'v-a'
  },
  components: {
    'v-a': {
      template: '<div>Composant A</div>'
    },
    'v-b': {
      template: '<div>Composant B</div>'
    }
  }
})
</script>

```

Transitions de liste

Jusqu'à présent, nous avons réalisé des transitions pour :

- des nœuds individuels
- des nœuds multiples où un seul est rendu à la fois

Alors, qu'en est-il lorsque nous avons une liste complète d'éléments où nous voulons faire un rendu simultané, par exemple avec **v-for** ? Dans ce cas, nous allons utiliser le composant **<transition-group>**. Cependant avant de plonger dans un exemple, il y a quelques éléments importants à connaître sur ce composant :

- Contrairement à **<transition>**, il rend un élément réel : par défaut un ****. Vous pouvez modifier l'élément rendu avec l'attribut **tag**.
- **Les modes de transition** ne sont pas disponibles car nous ne pouvons plus alterner entre des éléments mutuellement exclusifs.
- Les éléments à l'intérieur **doivent toujours avoir** un attribut **key** unique.
- Les classes de transition CSS seront appliquées sur les éléments internes et non pas sur les groupes/conteneurs eux-mêmes.

Transitions de liste entrantes/sortantes

Maintenant, nous allons nous plonger dans un exemple simple, faire des transitions entrantes et sortantes en utilisant les mêmes classes CSS que celles utilisées précédemment :

```
<div id="list-demo" class="demo">
  <button v-on:click="add">Ajouter</button>
  <button v-on:click="remove">Enlever</button>
  <transition-group name="list" tag="p">
    <span v-for="item in items" :key="item" class="list-item">
      {{ item }}
    </span>
  </transition-group>
</div>
<script>
new Vue({
  el: '#list-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9],
    nextNum: 10
  },
  methods: {
    randomIndex: function () {
      return Math.floor(Math.random() * this.items.length)
    },
    add: function () {
      this.items.splice(this.randomIndex(), 0, this.nextNum++)
    },
    remove: function () {
      this.items.splice(this.randomIndex(), 1)
    },
  }
})
```

```

</script>
<style>
.list-item {
  display: inline-block;
  margin-right: 10px;
}
.list-enter-active, .list-leave-active {
  transition: all 1s;
}
.list-enter, .list-leave-to {
  opacity: 0;
  transform: translateY(30px);
}
</style>

```

Il y a un problème avec cet exemple. Quand vous ajoutez ou enlevez un élément, ceux qui l'entourent se placent instantanément dans leur nouvel emplacement au lieu de se déplacer en douceur.

Transitions de déplacement de liste

Le composant `<transition-group>` a un autre tour dans son sac. Il peut non seulement animer l'entrée et la sortie, mais aussi faire des changements de position. Le seul nouveau concept que vous devez connaître pour utiliser cette fonctionnalité, c'est l'addition de **la classe `v-move`**, qui est ajoutée quand les éléments changent de position. Comme les autres classes, son préfixe correspondra à la valeur d'un attribut `name` fourni et vous pouvez également spécifier manuellement une classe avec l'attribut `move-class`.

Cette classe est surtout utile pour spécifier le temps de la transition et la courbe d'accélération, comme vous pourrez le voir ci-dessous :

```

<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"
"></script>
<div id="flip-list-demo" class="demo">
  <button v-on:click="shuffle">Mélanger</button>
  <transition-group name="flip-list" tag="ul">
    <li v-for="item in items" :key="item">
      {{ item }}
    </li>
  </transition-group>
</div>
<script>
new Vue({
  el: '#flip-list-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9]
  },
  methods: {
    shuffle: function () {
      this.items = _.shuffle(this.items)
    }
  }
})

```

```

})
</script>
<style>
.flip-list-move {
  transition: transform 1s;
}
</style>

```

Cela peut sembler magique, mais sous le capot, Vue utilise une technique simple d'animation appelée [FLIP](#) pour transiter en douceur les éléments de leur ancienne position vers la nouvelle à l'aide de transformations.

Nous pouvons combiner cette technique avec notre implémentation précédente pour animer chaque changement possible dans notre liste !

```

<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"
"></script>
<div id="list-complete-demo" class="demo">
  <button v-on:click="shuffle">Mélanger</button>
  <button v-on:click="add">Ajouter</button>
  <button v-on:click="remove">Enlever</button>
  <transition-group name="list-complete" tag="p">
    <span v-for="item in items" :key="item" class="list-complete-item">
      {{ item }}
    </span>
  </transition-group>
</div>
<script>
new Vue({
  el: '#list-complete-demo',
  data: {
    items: [1,2,3,4,5,6,7,8,9],
    nextNum: 10
  },
  methods: {
    randomIndex: function () {
      return Math.floor(Math.random() * this.items.length)
    },
    add: function () {
      this.items.splice(this.randomIndex(), 0, this.nextNum++)
    },
    remove: function () {
      this.items.splice(this.randomIndex(), 1)
    },
    shuffle: function () {
      this.items = _.shuffle(this.items)
    }
  }
})
</script>
<style>
.list-complete-item {

```

```

        transition: all 1s;
        display: inline-block;
        margin-right: 10px;
    }
.list-complete-enter, .list-complete-leave-to {
    opacity: 0;
    transform: translateY(30px);
}
.list-complete-leave-active {
    position: absolute;
}
</style>

```

Un point important est à noter : ces transitions FLIP ne fonctionnent pas si des éléments sont configurés avec `display: inline`. Comme alternative, vous pouvez utiliser `display: inline-block` où placer des éléments dans un contexte flexible.

Échelonnage des transitions de liste

En communiquant avec des transitions JavaScript via des attributs de données, il est également possible d'échelonner les transitions dans une liste :

```

<script
src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"
"></script>
<div id="example-5" class="demo">
    <input v-model="query">
    <transition-group
        name="staggered-fade"
        tag="ul"
        v-bind:css="false"
        v-on:before-enter="beforeEnter"
        v-on:enter="enter"
        v-on:leave="leave"
    >
        <li
            v-for="(item, index) in computedList"
            v-bind:key="item.msg"
            v-bind:data-index="index"
            >{{ item.msg }}</li>
    </transition-group>
</div>
<script>
new Vue({
    el: '#example-5',
    data: {
        query: '',
        list: [
            { msg: 'Bruce Lee' },
            { msg: 'Jackie Chan' },
            { msg: 'Chuck Norris' },
            { msg: 'Jet Li' },

```

```

        { msg: 'Kung Fury' }
    ]
},
computed: {
    computedList: function () {
        var vm = this
        return this.list.filter(function (item) {
            return item.msg.toLowerCase().indexOf(vm.query.toLowerCase()) !==
-1
        })
    }
},
methods: {
    beforeEnter: function (el) {
        el.style.opacity = 0
        el.style.height = 0
    },
    enter: function (el, done) {
        var delay = el.dataset.index * 150
        setTimeout(function () {
            Velocity(
                el,
                { opacity: 1, height: '1.6em' },
                { complete: done }
            )
        }, delay)
    },
    leave: function (el, done) {
        var delay = el.dataset.index * 150
        setTimeout(function () {
            Velocity(
                el,
                { opacity: 0, height: 0 },
                { complete: done }
            )
        }, delay)
    }
}
})
</script>

```

Transitions réutilisables

Les Transitions peuvent être réutilisées par le biais du système de composant de Vue. Pour créer une transition réutilisable, il suffit de placer un composant `<transition>` ou `<transition-group>` à la racine, puis passer les enfants dans le composant de transition.

Voici un exemple utilisant un composant template :

```

Vue.component('my-special-transition', {
    template: `\
        <transition\`

```

```

    name="very-special-transition"\n
    mode="out-in"\n
    v-on:before-enter="beforeEnter"\n
    v-on:after-enter="afterEnter"\n
  >\n
    <slot></slot>\n
  </transition>\n
',
methods: {
  beforeEnter: function (el) {
    // ...
  },
  afterEnter: function (el) {
    // ...
  }
}
})

```

Et les composants fonctionnels sont particulièrement adaptés à cette tâche :

```

Vue.component('my-special-transition', {
  functional: true,
  render: function (createElement, context) {
    var data = {
      props: {
        name: 'very-special-transition',
        mode: 'out-in'
      },
      on: {
        beforeEnter: function (el) {
          // ...
        },
        afterEnter: function (el) {
          // ...
        }
      }
    }
    return createElement('transition', data, context.children)
  }
})

```

Transitions dynamiques

Oui, même les transitions dans Vue sont pilotées par les données ! Le plus simple des exemples d'une transition dynamique lie l'attribut `name` à une propriété dynamique.

```

<transition v-bind:name="transitionName">
  <!-- ... -->
</transition>

```

Cela peut être utile quand vous avez défini des transitions/animations CSS à l'aide des conventions de classes de transition de Vue et que vous souhaitez simplement basculer entre elles.

En vérité, tout attribut de transition peut être dynamiquement lié. Et il ne s'agit pas seulement des attributs. Étant donné que les hooks d'évènements ne sont que des méthodes, ils ont accès à toutes les données dans le contexte. Cela signifie que selon l'état de votre composant, vos transitions JavaScript peuvent se comporter différemment.

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js
"></script>
<div id="dynamic-fade-demo" class="demo">
  Fondu entrant : <input type="range" v-model="fadeInDuration" min="0" v-
  bind:max="maxFadeDuration">
  Fondu sortant : <input type="range" v-model="fadeOutDuration" min="0" v-
  bind:max="maxFadeDuration">
  <transition
    v-bind:css="false"
    v-on:before-enter="beforeEnter"
    v-on:enter="enter"
    v-on:leave="leave"
  >
    <p v-if="show">bonjour</p>
  </transition>
  <button
    v-if="stop"
    v-on:click="stop = false; show = false"
  >Commencer l'animation</button>
  <button
    v-else
    v-on:click="stop = true"
  >Arrêtez ça !</button>
</div>
<script>
new Vue({
  el: '#dynamic-fade-demo',
  data: {
    show: true,
    fadeInDuration: 1000,
    fadeOutDuration: 1000,
    maxFadeDuration: 1500,
    stop: true
  },
  mounted: function () {
    this.show = false
  },
  methods: {
    beforeEnter: function (el) {
      el.style.opacity = 0
    },
    enter: function (el, done) {
      var vm = this
      Velocity(el,
```

```

        { opacity: 1 },
        {
            duration: this.fadeInDuration,
            complete: function () {
                done()
                if (!vm.stop) vm.show = false
            }
        }
    ),
    leave: function (el, done) {
        var vm = this
        Velocity(el,
            { opacity: 0 },
            {
                duration: this.fadeOutDuration,
                complete: function () {
                    done()
                    vm.show = true
                }
            }
        )
    }
)
</script>

```

Finalement, le meilleur moyen de créer des transitions dynamiques, c'est par le biais de composants qui acceptent des props pour changer la nature des transitions à utiliser. Cela peut sembler mieux, mais la seule limite est votre imagination.

Le système de transition de Vue offre de nombreuses façons simples d'animer l'entrée, la sortie et les listes, mais qu'en est-il de l'animation de vos propres données ? Par exemple :

- les nombres et les calculs
- les couleurs affichées
- les positions des nœuds SVG
- les tailles et les autres propriétés des éléments

Tous ces éléments sont déjà stockés sous forme de nombres bruts, ou peuvent être convertis en nombres. Dès lors, nous pouvons animer ces modifications de l'état à l'aide de bibliothèques tierces vers un état intermédiaire, en combinaison avec la réactivité de Vue et les systèmes de composants.

Animation de l'état avec des observateurs

Les observateurs nous permettent d'animer les changements de toute propriété numérique dans une autre propriété. Cela peut paraître compliqué dans l'abstrait, donc plongeons-nous dans un exemple en utilisant [GreenSock](#):

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/gsap/1.20.3/TweenMax.min.js">
</script>
<div id="animated-number-demo" class="demo">
  <input v-model.number="number" type="number" step="20">
  <p>{{ animatedNumber }}</p>
</div>
<script>
new Vue({
  el: '#animated-number-demo',
  data: {
    number: 0,
    tweenedNumber: 0
  },
  computed: {
    animatedNumber: function() {
      return this.tweenedNumber.toFixed(0);
    }
  },
  watch: {
    number: function(newValue) {
      TweenLite.to(this.$data, 0.5, { tweenedNumber: newValue });
    }
  }
})
</script>
```

Lorsque vous mettez à jour le nombre, la modification est animée en dessous de l'input. Cela fait une belle démonstration, mais qu'en est-il de quelque chose qui n'est pas directement stocké comme un nombre, comme n'importe quelle couleur CSS valide par exemple ? Voici comment nous pourrions accomplir cela avec l'ajout de [Tween.js](#) et [Color.js](#):

```

<script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>
<script src="https://cdn.jsdelivr.net/npm/color.js@1.0.3"></script>
<div id="example-7" class="demo">
  <input
    v-model="colorQuery"
    v-on:keyup.enter="updateColor"
    placeholder="Entrer une couleur"
  >
  <button v-on:click="updateColor">Mettre à jour</button>
  <p>Aperçu :</p>
  <span
    v-bind:style="{ backgroundColor: tweenedCSSColor }"
    class="example-7-color-preview"
  ></span>
  <p>{{ tweenedCSSColor }}</p>
</div>
<script>
var Color = net.brehaut.Color
new Vue({
  el: '#example-7',
  data: {
    colorQuery: '',
    color: {
      red: 0,
      green: 0,
      blue: 0,
      alpha: 1
    },
    tweenedColor: {}
  },
  created: function () {
    this.tweenedColor = Object.assign({}, this.color)
  },
  watch: {
    color: function () {
      function animate () {
        if (TWEEN.update()) {
          requestAnimationFrame(animate)
        }
      }
      new TWEEN.Tween(this.tweenedColor)
        .to(this.color, 750)
        .start()

      animate()
    }
  },
  computed: {
    tweenedCSSColor: function () {
      return new Color({
        red: this.tweenedColor.red,
        green: this.tweenedColor.green,
        blue: this.tweenedColor.blue,
      })
    }
  }
})

```

```

        alpha: this.tweenedColor.alpha
    }).toCSS()
}
},
methods: {
    updateColor: function () {
        this.color = new Color(this.colorQuery).toRGB()
        this.colorQuery = ''
    }
}
})
</script>
<style>
.example-7-color-preview {
    display: inline-block;
    width: 50px;
    height: 50px;
}
</style>

```

Transitions d'état dynamiques

Tout comme pour les composants de transition de Vue, les transitions d'état de données peuvent être mises à jour en temps réel, ce qui est particulièrement utile pour le prototypage !

Même en utilisant un polygone SVG simple, vous pouvez obtenir de nombreux effets qu'il serait difficile de concevoir tant que vous n'avez pas un peu joué avec les variables.

Consulter [ce fiddle](#) pour voir le code complet derrière la démo ci-dessus.

Organisation des transitions dans les composants

La gestion de nombreuses transitions d'états peut augmenter rapidement la complexité d'une instance ou d'un composant Vue. Heureusement, de nombreuses animations peuvent être extraites dans des composants enfants dédiés. Faisons cela avec l'entier animé de notre exemple précédent :

```

<script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>

<div id="example-8">
    <input v-model.number="firstNumber" type="number" step="20"> +
    <input v-model.number="secondNumber" type="number" step="20"> =
    {{ result }}
    <p>
        <animated-integer v-bind:value="firstNumber"></animated-integer> +
        <animated-integer v-bind:value="secondNumber"></animated-integer> =
        <animated-integer v-bind:value="result"></animated-integer>
    </p>
</div>

```

```

// Cette logique d'interpolation complexe peut maintenant être réutilisée
entre
// les entiers que nous souhaitons animer dans notre application.
// Les composants offrent également une interface propre pour configurer
// des transitions plus dynamiques et des stratégies complexes
// de transition.
Vue.component('animated-integer', {
  template: '<span>{{ tweeningValue }}</span>',
  props: {
    value: {
      type: Number,
      required: true
    }
  },
  data: function () {
    return {
      tweeningValue: 0
    }
  },
  watch: {
    value: function (newValue, oldValue) {
      this.tween(oldValue, newValue)
    }
  },
  mounted: function () {
    this.tween(0, this.value)
  },
  methods: {
    tween: function (startValue, endValue) {
      var vm = this
      function animate () {
        if (TWEEN.update()) {
          requestAnimationFrame(animate)
        }
      }
      new TWEEN.Tween({ tweeningValue: startValue })
        .to({ tweeningValue: endValue }, 500)
        .onUpdate(function () {
          vm.tweeningValue = this.tweeningValue.toFixed(0)
        })
        .start()

      animate()
    }
  }
})

// Toute la complexité a été supprimée de l'instance principale de Vue !
new Vue({
  el: '#example-8',
  data: {
    firstNumber: 20,
    secondNumber: 40
  }
})

```

```
},  
computed: {  
    result: function () {  
        return this.firstNumber + this.secondNumber  
    }  
}  
)
```



Gérer la navigation

Gérer la navigation

Créer une application monopage avec Vue + Vue Router est vraiment simple. Avec Vue.js, nous concevons déjà notre application avec des composants. En ajoutant vue-router dans notre application, tout ce qu'il nous reste à faire est de relier nos composants aux routes, et de laisser vue-router faire le rendu. Voici un exemple de base :

HTML

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

<div id="app">
  <h1>Bonjour l'application !</h1>
  <p>
    <!-- utilisez le composant router-link pour la navigation. -->
    <!-- spécifiez le lien en le passant à la prop `to` -->
    <!-- `<router-link>` sera rendu en tag `<a>` par défaut -->
    <router-link to="/foo">Aller à Foo</router-link>
    <router-link to="/bar">Aller à Bar</router-link>
  </p>
  <!-- balise pour le composant router-view -->
  <!-- le composant correspondant à la route sera rendu ici -->
  <router-view></router-view>
</div>
```

JavaScript

```
// 0. Si vous utilisez un système de module (par ex. via vue-cli), il faut
// importer Vue et Vue Router et ensuite appeler `Vue.use(VueRouter)`.

// 1. Définissez les composants de route.
// Ces derniers peuvent être importés depuis d'autre fichier
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 2. Définissez des routes.
// Chaque route doit correspondre à un composant. Le « composant » peut
// soit être un véritable composant créé via `Vue.extend()`, ou juste un
// objet d'options.
// Nous parlerons plus tard des routes imbriquées.
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]

// 3. Créez l'instance du routeur et passez l'option `routes`.
// Vous pouvez également passer des options supplémentaires,
// mais nous allons faire simple pour l'instant.
const router = new VueRouter({
```

```

routes // raccourci pour `routes: routes`
})

// 5. Créez et montez l'instance de Vue.
// Soyez sûr d'injecter le routeur avec l'option `router` pour
// permettre à l'application tout entière d'être à l'écoute du routeur.
const app = new Vue({
  router
}).$mount('#app')

// L'application est maintenant en marche !

```

En injectant le routeur, nous y avons accès à travers `this.$router`. Nous avons également accès à la route courante derrière `this.$route` depuis n'importe quel composant :

```

// Home.vue
export default {
  computed: {
    username () {
      // Nous verrons ce que représente `params` dans un instant.
      return this.$route.params.username
    }
  },
  methods: {
    goBack () {
      window.history.length > 1
        ? this.$router.go(-1)
        : this.$router.push('/')
    }
  }
}

```

Dans les documentations, nous allons souvent utiliser l'instance `router`. Gardez à l'esprit que l'utilisation de `this.$router` est exactement la même chose que celle de `router`. La raison pour laquelle nous utilisons `this.$router` est la possibilité ainsi offerte de ne pas avoir à importer le routeur dans chaque fichier de composant ayant besoin d'accéder au routage.

Système de routing.

Routeur officiel

Pour la plupart des applications monopages, il est recommandé d'utiliser la bibliothèque officiellement supportée [vue-router](#). Pour plus de détails, voir la [documentation](#) de vue-router.

Du routage simple en partant de zéro

Si vous avez simplement besoin d'un routage très simple et ne souhaitez pas ajouter une bibliothèque riche en fonctionnalités, vous pouvez le faire en déclenchant dynamiquement le rendu d'un composant de page de cette manière :

```
const NotFound = { template: '<p>Page not found</p>' }
const Home = { template: '<p>home page</p>' }
const About = { template: '<p>about page</p>' }

const routes = {
  '/': Home,
  '/about': About
}

new Vue({
  el: '#app',
  data: {
    currentRoute: window.location.pathname
  },
  computed: {
    ViewComponent () {
      return routes[this.currentRoute] || NotFound
    }
  },
  render (h) { return h(this.ViewComponent) }
})
```

Combiné à l'API HTML5 History, vous pouvez construire un routeur client basique mais parfaitement fonctionnel. Pour en voir une démonstration, faites un checkout [de cette application d'exemple](#).

Intégrer des routeurs tierce-partie

S'il y a un routeur tierce-partie que vous préférez utiliser, tel que [Page.js](#) ou [Director](#), l'intégration est [tout aussi simple](#). Voici [un exemple complet](#) qui utilise Page.js.

Principes de Vue router.

Vue Router est le router officiel pour Vue.js. Il s'intègre aisément avec Vue.js pour faire des applications mono page avec Vue.js. Fonctionnalités incluses:

- Vues et routes imbriquées
- Modulaire, configuration basée sur les composants
- Paramètres de route, de requête
- Effets de transition de vues basées sur le système de transition de Vue.js
- Gestion fine de la navigation
- Classes CSS pour liens actifs
- Mode HTML5 de la gestion de l'historique ou mode par hash, avec solution de repli automatique pour IE9
- Gestion du scroll

Pour un guide complet sur l'utilisation du nouveau Vue Router, consultez [la documentation Vue Router](#).

Initialisation du Routeur

Vous avez juste à passer la propriété **router** à l'instance de Vue :

```
new Vue({  
  el: '#app',  
  router: router,  
  template: '<router-view></router-view>'  
})
```

Ou, si vous utilisez le build runtime de Vue :

```
new Vue({  
  el: '#app',  
  router: router,  
  render: h => h('router-view')  
})
```

Configurer des routes.

Les routes sont définies dans un tableau dans une option `routes` lors de l'instanciation du routeur. Donc ces routes par exemple :

```
var router = new VueRouter({
  routes: [
    { path: '/foo', component: Foo },
    { path: '/bar', component: Bar }
  ]
})
```

La syntaxe par tableau permet une plus grande prédictibilité de la concordance des routes, puisque l'itération sur un objet ne garantit pas le même ordre d'affichage sur tous les navigateurs.

```
<div class="upgrade-path">
  <h4>Comment procéder ?</h4>
  <p>Lancez l'<a href="https://github.com/vuejs/vue-migration-helper">outil d'aide à la migration</a> sur votre code pour trouver des exemples d'appel de <code>router.map</code>.</p>
</div>
```

Si vous avez besoin programmatiquement de générer les routes au démarrage de votre application, vous pouvez le faire dynamiquement en ajoutant les définitions de route dans votre tableau.

Par exemple :

```
// Base de routes statiques
var routes = [
  // ...
]

// Routes générées dynamiquement
marketingPages.forEach(function (page) {
  routes.push({
    path: '/marketing/' + page.slug
    component: {
      extends: MarketingComponent
      data: function () {
        return { page: page }
      }
    }
  })
}

var router = new Router({
  routes: routes
})
```

Si vous avez besoin d'ajouter une nouvelle route après que le routeur soit instancié, vous pouvez remplacer l'objet de concordance des routes du routeur par un nouveau contenant la route que vous souhaitez ajouter :

```
router.match = createMatcher(  
  [{  
    path: '/my/new/path',  
    component: MyComponent  
  }].concat(router.options.routes)  
)
```

router.beforeEach

`router.beforeEach` fonctionne maintenant de manière asynchrone et prend une fonction de rappel `next` en tant que troisième argument.

```
router.beforeEach(function (transition) {  
  if (transition.to.path === '/forbidden') {  
    transition.abort()  
  } else {  
    transition.next()  
  }  
})
```

```
router.beforeEach(function (to, from, next) {  
  if (to.path === '/forbidden') {  
    next(false)  
  } else {  
    next()  
  }  
})
```

Redirection

Par une définition comme ci-dessous dans votre configuration de `routes` :

```
{  
  path: '/tos',  
  redirect: '/terms-of-service'  
}
```

Alias

Par une définition comme ci-dessous dans votre configuration de `routes` :

```
{  
  path: '/admin',  
  component: AdminPanel,  
  alias: '/manage'  
}
```

Si vous avez besoin de plusieurs alias, vous pouvez aussi utiliser la syntaxe de tableau :

```
alias: ['/manage', '/administer', '/administrate']
```

Propriétés de route personnalisées

Les propriétés de route personnalisées sont imbriquées dans une nouvelle propriété meta pour éviter les conflits avec les fonctionnalités futures.

```
{  
  path: '/admin',  
  component: AdminPanel,  
  meta: {  
    requiresAuth: true  
  }  
}
```

Puis quand vous accédez à cette propriété pour une route, vous pourrez toujours y accéder via **meta**. Par exemple :

```
if (route.meta.requiresAuth) {  
  // ...  
}
```

Concordance de routes

La concordance de routes utilise [path-to-regexp](#) pour fonctionner, ce qui la rend plus flexible que précédemment.

La syntaxe a quelque peu changé, ainsi [/category/*tags](#) par exemple, doit être mis à jour pour [/category/:tags+](#).

Liens

routerLink

Le composant [`<router-link>`](#) permet de créer des liens.

```
<router-link to="/about">À propos</router-link>
```

Notez que `target="_blank"` n'est pas supporté sur `<router-link>` donc si vous avez besoin d'ouvrir un lien dans un nouvel onglet, vous devez utiliser `<a>` à la place.

```
<router-link tag="li" to="/about">
  <a>À propos</a>
</router-link>
```

Le `<a>` sera dans ce cas le lien (et amènera sur l'adresse correcte), mais la classe active sera appliquée sur le `` extérieur.

Navigation par programmation

router.go

`router.go` est utilisé pour la navigation en arrière ou en avant alors que `router.push` est utilisé pour naviguer vers une page spécifique.

Sécuriser une application : les Navigation Guards ou Hooks de Route

activate

Utilisez `beforeRouteEnter` dans le composant à la place.

canActivate

Utilisez `beforeEnter` dans le composant à la place.

deactivate

Utilisez le `beforeDestroy` du composant ou le hook `destroyed` à la place.

canDeactivate

Utilisez `beforeRouteLeave` dans le composant à la place.

data

La propriété `$route` est maintenant réactive donc vous pouvez juste utiliser un observateur pour réagir au changement de route comme ceci :

```
watch: {
  '$route': 'fetchData'
},
methods: {
  fetchData: function () {
    // ...
  }
}
```

Transitions entre écrans.

Vu que `<router-view>` est essentiellement un composant dynamique, on peut lui appliquer certains effets de transitions en utilisant le composant `<transition>` :

```
<transition>
  <router-view></router-view>
</transition>
```

Tout à propos de `<transition>` fonctionne également ici de la même manière.

Transition par route

L'utilisation du dessus applique la même transition pour chaque route. Si vous voulez que les composants de route aient des transitions différentes, vous pouvez utiliser à la place `<transition>` avec des noms différents à l'intérieur de chaque composant de route :

```
const Foo = {
  template: `
    <transition name="slide">
      <div class="foo">...</div>
    </transition>
  `
}

const Bar = {
  template: `
    <transition name="fade">
      <div class="bar">...</div>
    </transition>
  `
}
```

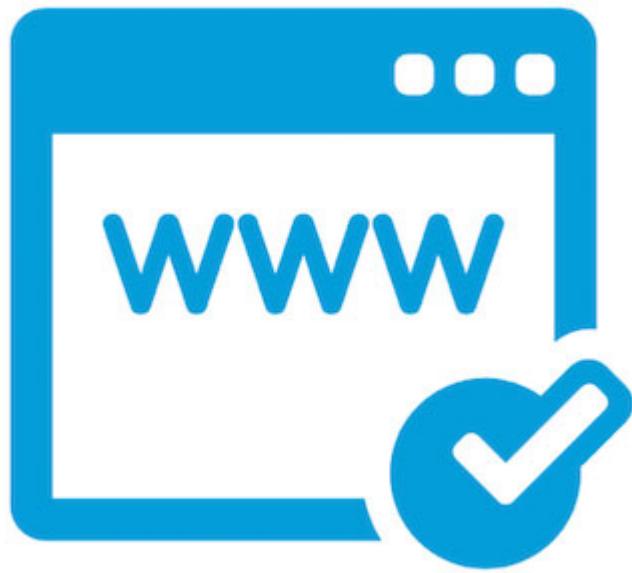
Transition dynamique basée sur la route

Il est aussi possible de déterminer la transition à utiliser en se basant sur la relation entre la route cible et la route actuelle :

```
<!-- utiliser un nom de transition dynamique -->
<transition :name="transitionName">
  <router-view></router-view>
</transition>
```

```
// et dans le composant parent,
// observer la `$route` pour déterminer la transition à utiliser
```

```
watch: {
  '$route' (to, from) {
    const toDepth = to.path.split('/').length
    const fromDepth = from.path.split('/').length
    this.transitionName = toDepth < fromDepth ? 'slide-right' : 'slide-left'
  }
}
```



Gestion des données avec Vuex

Gestion des données avec Vuex

Vuex est un **gestionnaire d'état** (« **state management pattern** ») et une bibliothèque pour des applications Vue.js. Il sert de zone de stockage de données centralisée pour tous les composants dans une application, avec des règles pour s'assurer que l'état ne puisse subir de mutations que d'une manière prévisible. Il s'intègre également avec [l'extension officielle](#) de Vue afin de fournir des fonctionnalités avancées comme la visualisation d'état dans le temps et des exports et imports d'instantanés (« **snapshot** ») d'état.

Un « gestionnaire d'état », qu'est-ce que c'est ?

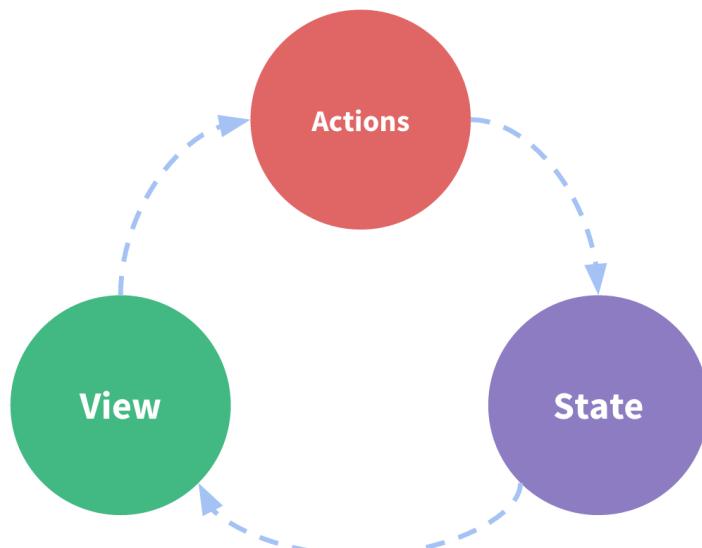
Commençons par une simple application de comptage avec Vue :

```
new Vue({
  // état
  data () {
    return {
      count: 0
    }
  },
  // vue
  template: `
    <div>{{ count }}</div>
  `,
  // actions
  methods: {
    increment () {
      this.count++
    }
  }
})
```

C'est une application autosuffisante avec les parties suivantes :

- L'**état**, qui est la source de vérité qui pilotant votre application,
- La **vue**, qui est une réflexion déclarative de l'**état**,
- Les **actions**, qui sont les façons possibles pour l'état de changer en réaction aux actions utilisateurs depuis la **vue**.

Voici une représentation extrêmement simple du concept de « flux de donnée unidirectionnel » :



Cependant, la simplicité s'évapore rapidement lorsque nous avons **de multiples composants qui partagent un même état global** :

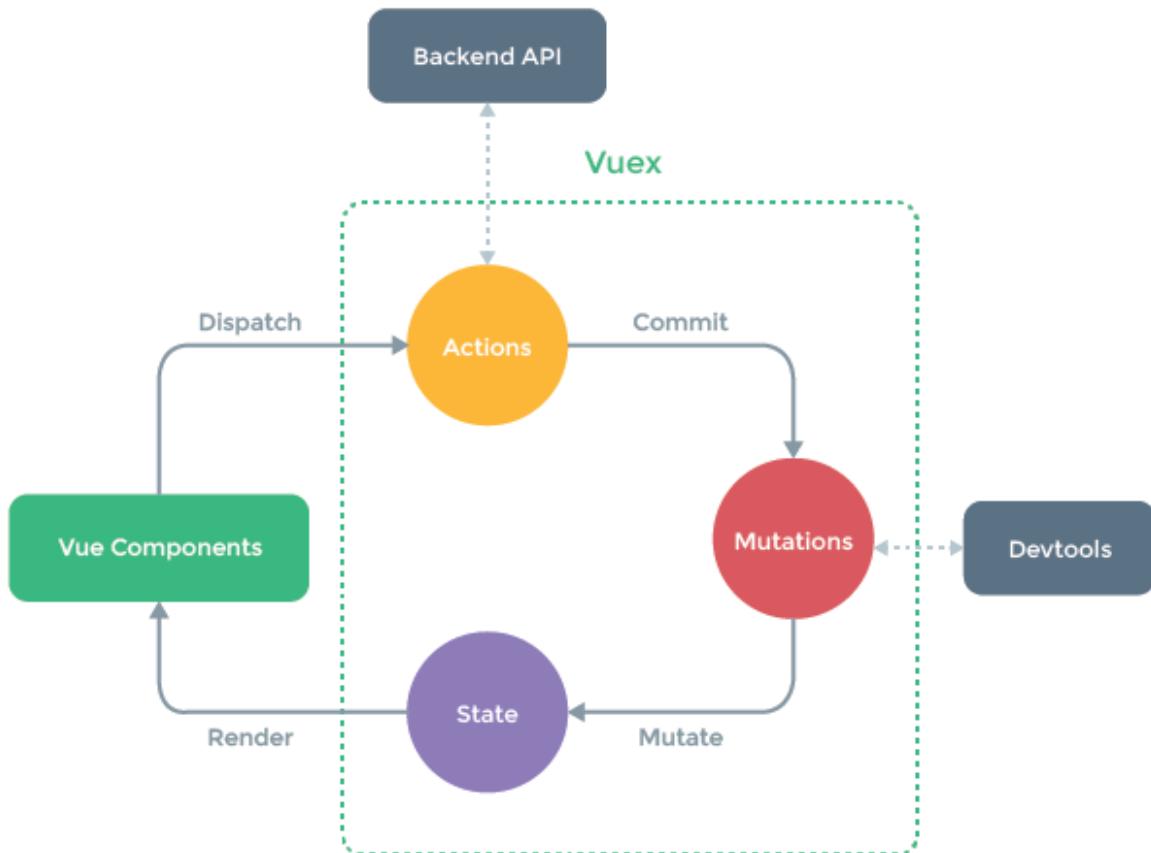
- Plusieurs vues peuvent dépendre de la même partie de l'état global.
- Des actions dans différentes vues peuvent avoir besoin de muter la même partie de l'état global.

Pour le premier problème, passer des props peut être fastidieux pour les composants profondément imbriqués, et ça ne fonctionne tout simplement pas pour les composants d'un même parent. Pour le deuxième problème, on se retrouve souvent à se rabattre sur des solutions telles qu'accéder aux références d'instance du parent/enfant direct ou essayer de muter et synchroniser de multiples copies de l'état via des événements. Ces deux modèles sont fragiles et posent rapidement des problèmes de maintenabilité du code.

Alors pourquoi ne pas extraire l'état global partagé des composants, et le gérer dans un singleton global ? De cette manière, notre arbre de composant devient une grosse « vue », et n'importe quel composant peut accéder à l'état global ou déclencher des actions, peu importe où il se trouve dans l'arbre !

De plus, en définissant et en séparant les concepts impliqués dans la gestion de l'état global et en appliquant certaines règles, on donne aussi une structure et une maintenabilité à notre code.

Voilà l'idée de base derrière Vuex, inspiré par [Flux](#), [Redux](#) et [l'architecture Elm](#). À l'inverse des autres modèles, Vuex est aussi une bibliothèque d'implémentation conçue spécialement pour Vue.js afin de bénéficier de son système de réactivité granulaire pour des modifications efficaces.



Quand l'utiliser ?

Bien que Vuex nous aide à gérer un état global partagé, il apporte aussi le cout de nouveaux concepts et *abstraction de code* (« boilerplate »). C'est un compromis entre la productivité à court terme et à long terme.

Si vous n'avez jamais créé une *application monopage* à grande échelle et que vous sautez directement dans Vuex, cela peut paraître verbeux et intimidant. C'est parfaitement normal ; si votre application est simple, vous vous en sortirez sans doute très bien sans Vuex.

Un simple [canal d'évènement global](#) pourrait très bien vous suffire. Mais si vous devez créer une application monopage à moyenne ou grande échelle, il y a des chances que vous vous trouviez dans des situations qui vous feront vous interroger sur une meilleure gestion de l'état global, détaché de votre composant Vue, et Vuex sera naturellement la prochaine étape pour vous. Voici une bonne citation de Dan Abramov, l'auteur de Redux :

State

Arbre d'état unique

Vuex utilise un **arbre d'état unique**, c'est-à-dire que cet unique objet contient tout l'état au niveau applicatif et sert de « source de vérité unique ». Cela signifie également que vous n'aurez qu'un seul store pour chaque application. Un arbre d'état unique rend rapide la localisation d'une partie spécifique de l'état et permet de facilement prendre des instantanés de l'état actuel de l'application à des fins de débogage.

L'arbre d'état unique n'entre pas en conflit avec la modularité. Dans les prochains chapitres, nous examinerons comment séparer votre état et vos mutations dans des sous-modules.

Récupération d'état Vuex dans des composants Vue

Alors, comment affichons-nous l'état du store dans nos composants Vue ? Puisque les stores Vuex sont réactifs, la façon la plus simple d'y « récupérer » l'état est tout simplement de retourner une partie de l'état depuis une [une propriété calculée](#) :

```
// créons un composant Counter
const Counter = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {
      return store.state.count
    }
  }
}
```

Lorsque `store.state.count` change, cela entraînera la réévaluation de la propriété calculée, et déclenchera les actions associées au DOM.

Cependant, ce modèle oblige le composant à compter sur le singleton global du store. Lorsqu'on utilise un système de module, il est nécessaire d'importer le store dans tous les composants qui utilisent l'état du store, et il est également nécessaire de le simuler lorsque l'on teste le composant.

Vuex fournit un mécanisme pour « injecter » le store dans tous les composants enfants du composant racine avec l'option `store` (activée par `Vue.use(Vuex)`) :

```
const app = new Vue({
  el: '#app',
  // fournit le store avec l'option `store`.
  // cela injectera l'instance du store dans tous les composants enfants.
  store,
  components: { Counter },
  template:
    <div class="app">
      <counter></counter>
    </div>
})
```

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

En fournissant l'option `store` à l'instance racine, le store sera injecté dans tous les composants enfants de la racine et sera disponible dans ces derniers avec `this.$store`. Mettons à jour notre implémentation de `Counter` :

```
const Counter = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {
      return this.$store.state.count
    }
  }
}
```

La fonction utilitaire `mapState`

Lorsqu'un composant a besoin d'utiliser plusieurs accesseurs ou propriétés de l'état du store, déclarer toutes ces propriétés calculées peut devenir répétitif et verbeux. Afin de pallier à ça, nous pouvons utiliser la fonction utilitaire `mapState` qui génère des fonctions d'accès pour nous et nous épargne quelques coups de clavier :

```
// dans la version complète, des fonctions utilitaires sont exposées
// telles que `Vuex.mapState`
import { mapState } from 'vuex'

export default {
  // ...
  computed: mapState({
    // les fonctions fléchées peuvent rendre le code très succinct !
    count: state => state.count,

    // passer la valeur littérale 'count' revient à écrire `state =>
    state.count`
    countAlias: 'count',

    // pour accéder à l'état local avec `this`, une fonction normale doit
    // être utilisée
    countPlusLocalStorage (state) {
      return state.count + this.localCount
    }
  })
}
```

Il est également possible de fournir un tableau de chaînes de caractères à `mapState` lorsque le nom de la propriété calculée associée est le même que le nom de l'état du store.

```
computed: mapState([
  // attacher `this.count` à `store.state.count`
  'count'
])
```

Opérateur de décomposition

Notez que `mapState` renvoie un objet. Comment l'utiliser en complément des autres propriétés calculées locales ? Normalement, il faudrait utiliser un outil pour fusionner les multiples objets en un seul afin de passer cet objet final à `computed`. Cependant avec l'[opérateur de décomposition](#) (qui est une proposition stage-4 ECMAScript), nous pouvons grandement simplifier la syntaxe :

```
computed: {
  localComputed () { /* ... */ },
  // rajouter cet objet dans l'objet `computed` avec l'opérateur de
  // décomposition
  ...mapState({
    // ...
  })
}
```

Les composants peuvent toujours avoir un état local

Utiliser Vuex ne signifie pas que vous devez mettre **tout** votre état dans Vuex. Bien que le fait de mettre plus d'états dans Vuex rende vos mutations d'état plus explicites et plus débogable, parfois il peut aussi rendre le code plus verbeux et indirect. Si une partie de l'état appartient directement à un seul composant, il est parfaitement sain de la laisser dans l'état local. Assurez-vous de prendre en compte les avantages et inconvénients d'une telle décision afin de vous adapter au mieux aux besoins de votre application.

Accesseurs

Parfois nous avons besoin de calculer des valeurs basées sur l'état du store, par exemple pour filtrer une liste d'éléments et les compter :

```
computed: {
  doneTodosCount () {
    return this.$store.state.todos.filter(todo => todo.done).length
  }
}
```

Si plus d'un composant a besoin d'utiliser cela, il nous faut ou bien dupliquer cette fonction, ou bien l'extraire dans une fonction utilitaire séparée et l'importer aux endroits nécessaires. Les deux idées sont loin d'être idéales.

Vuex nous permet de définir des accesseurs (« getters ») dans le store. Voyez-les comme les propriétés calculées des stores. Comme pour les propriétés calculées, le résultat de l'accesseur est mis en cache en se basant sur ses dépendances et il ne sera réévalué que lorsque l'une de ses dépendances aura changée.

Les accesseurs prennent l'état en premier argument :

```
const store = new Vuex.Store({
  state: {
    todos: [
      { id: 1, text: '...', done: true },
      { id: 2, text: '...', done: false }
    ]
  },
  getters: {
    doneTodos: state => {
      return state.todos.filter(todo => todo.done)
    }
  }
})
```

Accès par propriété

Les accesseurs seront exposés sur l'objet `store.getters` et vous accéderez aux valeurs comme des propriétés :

```
store.getters.doneTodos // -> [{ id: 1, text: '...', done: true }]
```

Les accesseurs recevront également les autres accesseurs en second argument :

```
getters: {
  // ...
  doneTodosCount: (state, getters) => {
    return getters.doneTodos.length
  }
}
```

```
store.getters.doneTodosCount // -> 1
```

Nous pouvons maintenant facilement les utiliser dans n'importe quel composant :

```
computed: {
  doneTodosCount () {
    return this.$store.getters.doneTodosCount
  }
}
```

Notez que les accesseurs accédés par propriétés sont mis en cache par le système de réactivité de Vue.

Accès par méthode

Vous pouvez aussi passer des arguments aux accesseurs en retournant une fonction. Cela est particulièrement utile quand vous souhaitez interroger un tableau dans le store :

```
getters: {
  // ...
  getTodoById: (state) => (id) => {
    return state.todos.find(todo => todo.id === id)
  }
}
```

```
store.getters.getTodoById(2) // -> { id: 2, text: '...', done: false }
```

Notez que les accesseur accédés par méthodes vont être exécuté chaque fois qu'il seront appelés. Le résultat ne sera donc pas mis en cache.

La fonction utilitaire `mapGetters`

La fonction utilitaire `mapGetters` attache simplement vos accesseurs du store aux propriétés calculées locales :

```
import { mapGetters } from 'vuex'
```

```
export default {
  // ...
  computed: {
    // rajouter les accesseurs dans `computed` avec l'opérateur de
    // décomposition
    ...mapGetters([
      'doneTodosCount',
      'anotherGetter',
      // ...
    ])
  }
}
```

Si vous voulez attacher un accesseur avec un nom différent, utilisez un objet :

```
...mapGetters({
  // attacher `this.doneCount` à `this.$store.getters.doneTodosCount`
  doneCount: 'doneTodosCount'
})
```

Mutations

La seule façon de vraiment modifier l'état dans un store Vuex est d'acter une mutation. Les mutations Vuex sont très similaires aux évènements : chaque mutation a un **type** sous forme de chaîne de caractères et un **gestionnaire**. La fonction de gestion est en charge de procéder aux véritables modifications de l'état, et elle reçoit l'état en premier argument :

```
const store = new Vuex.Store({
  state: {
    count: 1
  },
  mutations: {
    increment (state) {
      // muter l'état
      state.count++
    }
  }
})
```

Vous ne pouvez pas appeler directement un gestionnaire de mutation. Le parti-pris ici est proche de l'abonnement à un évènement : « Lorsqu'une mutation du type **increment** est déclenchée, appelle ce gestionnaire. » Pour invoquer un gestionnaire de mutation, il faut appeler **store.commit** avec son type :

```
store.commit('increment')
```

Acter avec un argument additionnel

Vous pouvez donner un argument additionnel (« payload ») à la fonction **store.commit** lors de la mutation :

```
// ...
mutations: {
  increment (state, n) {
    state.count += n
  }
}
```

```
store.commit('increment', 10)
```

Dans la plupart des cas, l'argument additionnel devrait être un objet, ainsi il peut contenir plusieurs champs, et les mutations enregistrées seront également plus descriptives :

```
// ...
mutations: {
  increment (state, payload) {
    state.count += payload.amount
  }
}
```

```
store.commit('increment', {
  amount: 10
})
```

Acter avec un objet

Une méthode alternative pour acter une mutation est d'utiliser directement un objet qui a une propriété **type** :

```
store.commit({
  type: 'increment',
  amount: 10
})
```

Lors de l'utilisation d'un objet pour acter, c'est l'objet lui-même qui fera office d'argument pour aux gestionnaires de mutation, le gestionnaire reste donc inchangé :

```
mutations: {
  increment (state, payload) {
    state.count += payload.amount
  }
}
```

Les mutations suivent les règles de réactivité de Vue

Puisqu'un état de store de Vuex est rendu réactif par Vue, lorsque nous mutons l'état, les composants Vue observant cet état seront automatiquement mis à jour. Cela signifie également que les mutations Vuex sont sujettes aux mêmes limitations qu'avec l'utilisation de Vue seul :

1. Initialisez de préférence le store initial de votre état avec tous les champs désirés auparavant.
2. Lorsque vous ajoutez de nouvelles propriétés à un objet, vous devriez soit :
 - Utiliser **Vue.set(obj, 'newProp', 123)**, ou
 - Remplacer cet objet par un nouvel objet. Par exemple, en utilisant [opérateur de décomposition \(stage-2\)](#), il est possible d'écrire :

```
state.obj = { ...state.obj, newProp: 123 }
```

Utilisation de constante pour les types de mutation

C'est une façon de faire régulière que d'utiliser des constantes pour les types de mutations dans diverses implémentations de Flux. Cela permet au code de bénéficier d'outils comme les linters (des outils d'aide à l'analyse syntaxique), et écrire toutes ces constantes dans un seul fichier permet à vos collaborateurs d'avoir un aperçu de quelles mutations sont possibles dans toute l'application :

```
// mutation-types.js
export const SOME_MUTATION = 'SOME_MUTATION'
```

```
// store.js
import Vuex from 'vuex'
import { SOME_MUTATION } from './mutation-types'

const store = new Vuex.Store({
  state: { ... },
  mutations: {
    // nous pouvons utiliser la fonctionnalité de nom de propriété
    // calculée
    [SOME_MUTATION] (state) {
      // muter l'état
    }
  }
})
```

Utiliser les constantes ou non relève de la préférence personnelle. Cela peut être bénéfique sur un gros projet avec beaucoup de développeurs, mais c'est totalement optionnel si vous n'aimez pas cette pratique.

Les mutations doivent être synchrones

Une règle importante à retenir est que **les fonctions de gestion des mutations doivent être synchrones**. Pourquoi ? Considérons l'exemple suivant :

```
mutations: {
  someMutation (state) {
    api.callAsyncMethod(()) => {
      state.count++
    })
  }
}
```

Maintenant imaginons que nous déboguons l'application et que nous regardons dans les logs de mutation des outils de développement (« devtools »). Pour chaque mutation enregistrée, le devtool aura besoin de capturer un instantané de l'état « avant » et un instantané « après ». Cependant, la fonction de rappel asynchrone de

l'exemple ci-dessus rend l'opération impossible : la fonction de rappel n'est pas encore appelée lorsque la mutation est actée, et il n'y a aucun moyen pour le devtool de savoir quand la fonction de rappel sera véritablement appelée. Toute mutation d'état effectuée dans la fonction de rappel est essentiellement intraçable !

Acter des mutations dans les composants

Vous pouvez acter des mutations dans les composants avec `this.$store.commit('xxx')`, ou en utilisant la fonction utilitaire `mapMutations` qui attache les méthodes du composant aux appels de `store.commit` (nécessite l'injection de `store` à la racine) :

```
import { mapMutations } from 'vuex'

export default {
  // ...
  methods: {
    ...mapMutations([
      'increment' // attacher `this.increment()` à
      `this.$store.commit('increment')` 

      // `mapMutations` supporte également les paramètres additionnels :
      'incrementBy' // attacher `this.incrementBy(amount)` à
      `this.$store.commit('incrementBy', amount)`
    ]),
    ...mapMutations({
      add: 'increment' // attacher `this.add()` à
      `this.$store.commit('increment')`
    })
  }
}
```

En avant vers les actions

L'asynchronisme combiné à la mutation de l'état peut rendre votre programme très difficile à comprendre. Par exemple, lorsque vous appelez deux méthodes avec toutes les deux des fonctions de rappel asynchrones qui changent l'état, comment savez-vous quelle fonction de rappel est appelée en première ? C'est exactement la raison pour laquelle nous voulons séparer les deux concepts. Avec Vuex, **les mutations sont des transactions synchrones** :

```
store.commit('increment')
// n'importe quel changement d'état de « increment » par mutation
// devrait être faite de manière synchrone.
```

Pour gérer les opérations asynchrones, tournons-nous vers les [Actions](#).

Actions

Les actions sont similaires aux mutations, à la différence que :

- Au lieu de modifier l'état, les actions actent des mutations.
- Les actions peuvent contenir des opérations asynchrones.

Enregistrons une simple action :

```
const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      state.count++
    }
  },
  actions: {
    increment (context) {
      context.commit('increment')
    }
  }
})
```

Les gestionnaires d'action reçoivent un objet contexte qui expose le même ensemble de méthodes et propriétés que l'instance du store, donc vous pouvez appeler `context.commit` pour acter une mutation, ou accéder à l'état et aux accesseurs via `context.state` et `context.getters`. Nous verrons pourquoi cet objet contexte n'est pas l'instance du store elle-même lorsque nous présenterons les [Modules](#) plus tard.

En pratique, nous utilisons souvent la [déstructuration d'argument](#) pour simplifier quelque peu le code (particulièrement si nous avons besoin d'appeler `commit` plusieurs fois) :

```
actions: {
  increment ({ commit }) {
    commit('increment')
  }
}
```

Propager des actions

Les actions sont déclenchées par la méthode `store.dispatch` :

```
store.dispatch('increment')
```

Cela peut sembler idiot au premier abord : si nous avons besoin d'incrémenter le compteur, pourquoi ne pas simplement appeler `store.commit('increment')` directement ? Vous rappelez-vous que **les mutations doivent être synchrones** ? Les actions ne suivent pas cette règle. Il est possible de procéder à des opérations **asynchrones** dans une action :

```
actions: {
  incrementAsync ({ commit }) {
    setTimeout(() => {
      commit('increment')
    }, 1000)
  }
}
```

Les actions prennent également en charge les paramètres additionnels (« payload ») et les objets pour propager :

```
// propager avec un paramètre additionnel
store.dispatch('incrementAsync', {
  amount: 10
})

// propager avec un objet
store.dispatch({
  type: 'incrementAsync',
  amount: 10
})
```

Un exemple concret d'application serait une action pour vider un panier d'achats, ce qui implique **d'appeler une API asynchrone et d'acter de multiples mutations** :

```
actions: {
  checkout ({ commit, state }, products) {
    // sauvegarder les articles actuellement dans le panier
    const savedCartItems = [...state.cart.added]
    // envoyer la requête de checkout,
    // et vider le panier
    commit(types.CHECKOUT_REQUEST)
    // l'API de la boutique en ligne prend une fonction de rappel en cas
    de succès et une autre en cas d'échec
    shop.buyProducts(
      products,
      // gérer le succès
      () => commit(types.CHECKOUT_SUCCESS),
      // gérer l'échec
      () => commit(types.CHECKOUT_FAILURE, savedCartItems)
    )
  }
}
```

Notez que nous procédons à un flux d'opérations asynchrones, et enregistrons les effets de bord (mutation de l'état) de l'action en les actant.

Propager des actions dans les composants

Vous pouvez propager des actions dans les composants avec `this.$store.dispatch('xxx')`, ou en utilisant la fonction utilitaire `mapActions` qui attache les méthodes du composant aux appels de `store.dispatch` (nécessite l'injection de `store` à la racine) :

```
import { mapActions } from 'vuex'

export default {
  // ...
  methods: {
    ...mapActions([
      'increment' // attacher `this.increment()` à
      `this.$store.dispatch('increment')` 

      // `mapActions` supporte également les paramètres additionnels :
      'incrementBy' // attacher `this.incrementBy(amount)` à
      `this.$store.dispatch('incrementBy', amount)`
    ]),
    ...mapActions({
      add: 'increment' // attacher `this.add()` à
      `this.$store.dispatch('increment')`
    })
  }
}
```

Composer les actions

Les actions sont souvent asynchrones, donc comment savoir lorsqu'une action est terminée ? Et plus important, comment composer plusieurs actions ensemble pour manipuler des flux asynchrones plus complexes ?

La première chose à savoir est que `store.dispatch` peut gérer la Promesse (« Promise ») retournée par le gestionnaire d'action déclenché et par conséquent vous pouvez également retourner une Promesse :

```
actions: {
  actionA ({ commit }) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        commit('someMutation')
        resolve()
      }, 1000)
    })
  }
}
```

Maintenant vous pouvez faire :

```
store.dispatch('actionA').then(() => {
  // ...
})
```

Et également dans une autre action :

```
actions: {
  // ...
  actionB ({ dispatch, commit }) {
    return dispatch('actionA').then(() => {
      commit('someOtherMutation')
    })
  }
}
```

Pour finir, si nous utilisons `async / await`, nous pouvons composer nos actions ainsi :

```
// sachant que `getData()` et `getOtherData()` retournent des Promesses.

actions: {
  async actionA ({ commit }) {
    commit('gotData', await getData())
  },
  async actionB ({ dispatch, commit }) {
    await dispatch('actionA') // attendre que `actionA` soit finie
    commit('gotOtherData', await getOtherData())
  }
}
```

Il est possible pour un `store.dispatch` de déclencher plusieurs gestionnaires d'action dans différents modules. Dans ce genre de cas, la valeur renvoyée sera une Promesse qui se résout quand tous les gestionnaires déclenchés ont été résolus.

Modules

Du fait de l'utilisation d'un arbre d'état unique, tout l'état de notre application est contenu dans un seul et même gros objet. Cependant, au fur et à mesure que notre application grandit, le store peut devenir très engorgé.

Pour y remédier, Vuex nous permet de diviser notre store en **modules**. Chaque module peut contenir ses propres états, mutations, actions, accesseurs. Il peut même contenir ses propres modules internes.

```
const moduleA = {
  state: { ... },
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: { ... },
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> l'état du `moduleA`
store.state.b // -> l'état du `moduleB`
```

État local d'un module

Dans les mutations et accesseurs d'un module, le premier argument reçu sera **l'état local du module**.

```
const moduleA = {
  state: { count: 0 },
  mutations: {
    increment (state) {
      // `state` est l'état du module local
      state.count++
    }
  },
  getters: {
    doubleCount (state) {
      return state.count * 2
    }
  }
}
```

```
}
```

De façon similaire, dans les actions du module, `context.state` exposera l'état local, et l'état racine sera disponible avec `context.rootState` :

```
const moduleA = {
  // ...
  actions: {
    incrementIfOddOnRootSum ({ state, commit, rootState }) {
      if ((state.count + rootState.count) % 2 === 1) {
        commit('increment')
      }
    }
  }
}
```

Également, dans les accesseurs du module, l'état racine sera exposé en troisième argument :

```
const moduleA = {
  // ...
  getters: {
    sumWithRootCount (state, getters, rootState) {
      return state.count + rootState.count
    }
  }
}
```

Espace de nom

Par défaut, les actions, mutations et accesseurs à l'intérieur d'un module sont toujours enregistrés sous l'**espace de nom global**. Cela permet à de multiples modules d'être réactifs au même type de mutation et d'action.

Si vous souhaitez que votre module soit autosuffisant et réutilisable, vous pouvez le ranger sous un espace de nom avec `namespaced: true`. Quand le module est enregistré, tous ses accesseurs, actions et mutations seront automatiquement basés sur l'espace de nom du module dans lesquels ils sont rangés. Par exemple :

```
const store = new Vuex.Store({
  modules: {
    account: {
      namespaced: true,
      // propriétés du module
      state: { ... }, // l'état du module est déjà imbriqué et n'est pas
      affecté par l'option `namespace`
      getters: {
        ...
      }
    }
  }
})
```

```

        isAdmin () { ... } // -> getters['account/isAdmin']
    },
    actions: {
        login () { ... } // -> dispatch('account/login')
    },
    mutations: {
        login () { ... } // -> commit('account/login')
    },

    // modules imbriqués
    modules: {
        // hérite de l'espace de nom du module parent
        myPage: {
            state: { ... },
            getters: {
                profile () { ... } // -> getters['account/profile']
            }
        },
        // utilise un espace de nom imbriqué
        posts: {
            namespaced: true,
            state: { ... },
            getters: {
                popular () { ... } // -> getters['account/posts/popular']
            }
        }
    }
}
)

```

Les accesseurs et actions sous espace de nom reçoivent des `getters`, `dispatch` et `commit` localisés. En d'autres termes, vous pouvez utiliser les paramètres de module sans écrire de préfixe dans ce même module. Permuter entre un espace de nom ou non n'affecte pas le code à l'intérieur du module.

Accéder aux propriétés globales dans les modules à espace de nom

Si vous voulez utiliser des états et accesseurs globaux, `rootState` et `rootGetters` sont passés en 3^e et 4^e arguments des fonctions d'accès et sont également exposés en tant que propriété de l'objet `context` passé aux fonctions d'action.

Pour propager les actions ou les mutations actées dans l'espace de nom global, passez `{ root: true }` en 3^e argument à `dispatch` et `commit`.

```

modules: {
    foo: {
        namespaced: true,
        getters: {
            // Les `getters` sont localisés dans le module des accesseurs
        }
    }
}

```

```

    // vous pouvez utiliser `rootGetters` via le 4e argument des
    accessseurs
    someGetter (state, getters, rootState, rootGetters) {
      getters.someOtherGetter // -> 'foo/someOtherGetter'
      rootGetters.someOtherGetter // -> 'someOtherGetter'
    },
    someOtherGetter: state => { ... }
  },

  actions: {
    // les actions et mutations sont aussi localisées pour ce module
    // elles vont accepter une option `root` pour la racine des actions
    et mutations.
    someAction ({ dispatch, commit, getters, rootGetters }) {
      getters.someGetter // -> 'foo/someGetter'
      rootGetters.someGetter // -> 'someGetter'

      dispatch('someOtherAction') // -> 'foo/someOtherAction'
      dispatch('someOtherAction', null, { root: true }) // ->
      'someOtherAction'

      commit('someMutation') // -> 'foo/someMutation'
      commit('someMutation', null, { root: true }) // -> 'someMutation'
    },
    someOtherAction (ctx, payload) { ... }
  }
}
}

```

Enregistrer des actions globales dans des modules avec espace de nom

Si vous voulez enregistrer des actions globales dans des modules avec espace de nom, vous pouvez les marquer avec `root: true` et placer la définition de l'action dans une fonction `handler`. Par exemple :

```

{
  actions: {
    someOtherAction ({dispatch}) {
      dispatch('someAction')
    }
  },
  modules: {
    foo: {
      namespaced: true,

      actions: {
        someAction: {
          root: true,
          handler (namespacedContext, payload) { ... } // -> 'someAction'
        }
      }
    }
  }
}

```

```
}
```

Fonctions utilitaires liées avec espace de nom

Quand nous lions un module sous espace de nom à un composant avec les fonctions utilitaires `mapState`, `mapGetters`, `mapActions` and `mapMutations`, cela peut être légèrement verbeux :

```
computed: {
  ...mapState({
    a: state => state.some.nested.module.a,
    b: state => state.some.nested.module.b
  })
},
methods: {
  ...mapActions([
    'some/nested/module/foo', // -> this['some/nested/module/foo']()
    'some/nested/module/bar' // -> this['some/nested/module/bar']()
  ])
}
```

Dans ces cas-là, vous pouvez passer une chaîne de caractère représentant le nom d'espace en tant que premier argument aux fonctions utilitaires ainsi toutes les liaisons seront faites en utilisant le module comme contexte. Cela peut être simplifié comme ci-dessous :

```
computed: {
  ...mapState('some/nested/module', {
    a: state => state.a,
    b: state => state.b
  })
},
methods: {
  ...mapActions('some/nested/module', [
    'foo', // -> this.foo()
    'bar' // -> this.bar()
  ])
}
```

De plus, vous pouvez créer des fonctions utilitaires liées avec espace de nom en utilisant `createNamespacedHelpers`. Cela retourne un objet qui a les nouvelles fonctions utilitaires rattachées à la valeur d'espace de nom fournie :

```
import { createNamespacedHelpers } from 'vuex'

const { mapState, mapActions } =
createNamespacedHelpers('some/nested/module')
```

```

export default {
  computed: {
    // vérifie dans `some/nested/module`
    ...mapState({
      a: state => state.a,
      b: state => state.b
    })
  },
  methods: {
    // vérifie dans `some/nested/module`
    ...mapActions([
      'foo',
      'bar'
    ])
  }
}

```

Limitations pour les plugins des développeurs

Vous devez faire attention au nom d'espace imprévisible pour vos modules quand vous créez un [plugin](#) qui fournit les modules et laisser les utilisateurs les ajouter au store de Vuex. Vos modules seront également sous espace de nom si l'utilisateur du plugin l'ajoute sous un module sous espace de nom. Pour vous adapter à la situation, vous devez recevoir la valeur de l'espace de nom via vos options de plugin :

```

// passer la valeur d'espace de nom via une option du plugin
// et retourner une fonction de plugin Vuex
export function createPlugin (options = {}) {
  return function (store) {
    // ajouter l'espace de nom aux types de module
    const namespace = options.namespace || ''
    store.dispatch(namespace + 'pluginAction')
  }
}

```

Enregistrement dynamique de module

Vous pouvez enregistrer un module **après** que le store ait été créé avec la méthode [store.registerModule](#) :

```

// enregistrer un module `myModule`
store.registerModule('myModule', {
  // ...
})

// enregistrer un module imbriqué `nested/myModule`
store.registerModule(['nested', 'myModule'], {
  // ...
})

```

L'état des modules est disponible dans `store.state.myModule` et `store.state.nested.myModule`.

L'enregistrement dynamique de module permet aux autres plugins Vue de bénéficier de la gestion de l'état de Vuex en attachant un module au store de l'application. Par exemple, la bibliothèque `vuex-router-sync` intègre vue-router avec vuex en gérant l'état de la route d'application dans un module enregistré dynamiquement.

Vous pouvez aussi supprimer un module enregistré dynamiquement avec `store.unregisterModule(moduleName)`. Notez que vous ne pouvez pas supprimer des modules statiques (déclarés à la création du store) avec cette méthode.

Il est possible que vous souhaitiez préserver un état précédent quand vous abonnez un nouveau module. Par exemple préserver l'état depuis l'application rendue côté serveur. Vous pouvez réaliser ceci avec l'option `preserveState: store.registerModule('a', module, { preserveState: true })`.

Réutiliser un module

Parfois nous devrons créer de multiples instances d'un module pour, par exemple :

- créer plusieurs stores qui utilisent le même module (par ex. pour éviter les singltons d'état avec du SSR quand l'option `runInNewContext` est à `false` ou `'once'`) ou
- enregistrer le même module plusieurs fois dans le même store.

Si nous utilisons un objet pour déclarer l'état du module, alors cet objet d'état sera partagé par référence et causera de contamination inter store/module quand il sera muté.

C'est exactement le même problème qu'avec `data` dans un composant Vue. Ainsi la solution est là même, utiliser une fonction pour déclarer notre état de module (supporté par la 2.3.0+) :

```
const MyReusableModule = {
  state () {
    return {
      foo: 'bar'
    }
  },
  // mutations, actions, accesseurs...
}
```

Installation et mise en œuvre.

Téléchargement direct / CDN

<https://unpkg.com/vuex>

Unpkg.com fournit des liens CDN basés sur npm. Le lien ci-dessus pointera toujours vers la dernière release sur npm. Vous pouvez aussi utiliser un tag ou une version spécifique via un URL comme

<https://unpkg.com/vuex@2.0.0>.

Incluez `vuex` après Vue et l'installation sera automatique :

```
<script src="/path/to/vue.js"></script>
<script src="/path/to/vuex.js"></script>
```

npm

```
npm install vuex --save
```

Yarn

```
yarn add vuex
```

Lorsqu'il est utilisé avec un système de module, vous devez explicitement installer Vuex via `Vue.use()`:

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)
```

Vous n'avez pas besoin de faire cela lors de l'utilisation des balises de script globales (`<script>`).

Promesse

Vuex nécessite les [promesses](#). Si vous supportez des navigateurs qui n'implémentent pas les promesses (par ex. IE), vous devez utiliser une bibliothèque polyfill, comme [es6-promise](#).

Vous pouvez l'inclure par CDN :

```
<script src="https://cdn.jsdelivr.net/npm/es6-promise@4/dist/es6-
promise.auto.js"></script>
```

Puis `window.Promise` sera disponible automatiquement.

Si vous préférez utiliser un gestionnaire de package comme npm ou Yarn, installez le avec les commandes suivantes :

```
npm install es6-promise --save # NPM  
yarn add es6-promise # Yarn
```

Puis, ajoutez la ligne ci-dessous partout dans votre code juste avant l'utilisation de Vuex :

```
import 'es6-promise/auto'
```

Build de développement

Vous aurez besoin de cloner directement vuex depuis GitHub et le compiler vous-même si vous souhaitez utiliser le dernier build de développement.

```
git clone https://github.com/vuejs/vuex.git node_modules/vuex  
cd node_modules/vuex  
npm install  
npm run build
```

Étendre VueX avec les plugins.

Plugins

Les stores Vuex prennent une option `plugins` qui expose des hooks pour chaque mutation. Un plugin Vuex est simplement une fonction qui reçoit un store comme unique argument :

```
const myPlugin = store => {
  // appelé quand le store est initialisé
  store.subscribe((mutation, state) => {
    // appelé après chaque mutation.
    // Les mutations arrivent au format `{ type, payload }`.
  })
}
```

Et peut être utilisé ainsi :

```
const store = new Vuex.Store({
  // ...
  plugins: [myPlugin]
})
```

Acter des mutations dans des plugins

Les plugins ne sont pas autorisés à muter directement l'état. Tout comme vos composants, ils peuvent simplement déclencher des changements en actant des mutations.

En actant des mutations, un plugin peut être utilisé pour synchroniser la source de données avec le store. Par exemple, pour synchroniser la source de données d'une websocket vers le store (c'est juste un exemple artificiel, en réalité la fonction `createPlugin` peut prendre des options additionnelles pour des tâches plus complexes) :

```
export default function createWebSocketPlugin (socket) {
  return store => {
    socket.on('data', data => {
      store.commit('receiveData', data)
    })
    store.subscribe(mutation => {
      if (mutation.type === 'UPDATE_DATA') {
        socket.emit('update', mutation.payload)
      }
    })
  }
}
```

```
const plugin = createWebSocketPlugin(socket)

const store = new Vuex.Store({
  state,
  mutations,
  plugins: [plugin]
})
```

Prendre des instantanés de l'état

Parfois un plugin peut vouloir recevoir des « instantanés » de l'état, et également comparer l'état post mutation avec l'état prémutation. Pour faire ceci, vous aurez besoin d'effectuer une copie complète de l'état :

```
const myPluginWithSnapshot = store => {
  let prevState = _.cloneDeep(store.state)
  store.subscribe((mutation, state) => {
    let nextState = _.cloneDeep(state)

    // comparer `prevState` et `nextState` ...

    // sauver l'état pour la prochaine mutation
    prevState = nextState
  })
}
```

Les plugins qui peuvent prendre des instantanés ne devraient être utilisés que pendant le développement. Lorsqu'on utilise webpack ou Browserify, on peut laisser nos outils de développement (« devtools ») s'occuper de ça pour nous :

```
const store = new Vuex.Store({
  // ...
  plugins: process.env.NODE_ENV !== 'production'
    ? [myPluginWithSnapshot]
    : []
})
```

Le plugin sera utilisé par défaut. Pour la production, vous aurez besoin de [DefinePlugin](#) pour webpack ou de [envify](#) pour Browserify pour convertir la valeur de `process.env.NODE_ENV !== 'production'` à `false` pour le build final.

Plugin de logs intégré

Si vous utilisez [vue-devtools](#) vous n'avez probablement pas besoin de ça.

Vuex fournit un plugin de logs à des fins de débogage :

```
import createLogger from 'vuex/dist/logger'

const store = new Vuex.Store({
  plugins: [createLogger()]
})
```

La fonction `createLogger` prend quelques options :

```
const logger = createLogger({
  collapsed: false, // auto-expand logged mutations
  filter (mutation, stateBefore, stateAfter) {
    // retourne `true` si une mutation devrait être logguée
    // `mutation` est un `{ type, payload }`
    return mutation.type !== "aBlacklistedMutation"
  },
  transformer (state) {
    // transforme l'état avant de le logguer.
    // retourne par exemple seulement un sous-arbre spécifique
    return state.subTree
  },
  mutationTransformer (mutation) {
    // les mutations sont logguées au format `{ type, payload }`
    // nous pouvons les formater comme nous le souhaitons.
    return mutation.type
  },
  logger: console, // implementation de l'API `console`, par défaut
  `console`
})
```

Le fichier de logs peut aussi être inclus directement via une balise `script`, et exposera la fonction `createVuexLogger` globalement.

Notez que le plugin de logs peut prendre des instantanés de l'état, ne l'utilisez donc que durant le développement.

Librairies alternatives.

- Flux
- Redux
- MobX
- Unstated
- Pullstate



Développer une application connectée

Développer une application connectée

Charger et envoyer des données avec AJAX : les différents scénarios.

Gestion des formulaires.

Usage basique

Vous pouvez utiliser la directive `v-model` pour créer une liaison de données bidirectionnelle sur les champs de formulaire (input, select ou textarea). Elle choisira automatiquement la bonne manière de mettre à jour l'élément en fonction du type de champ. Bien qu'un peu magique, `v-model` est essentiellement du sucre syntaxique pour mettre à jour les données lors des évènements de saisie utilisateur sur les champs, ainsi que quelques traitements spéciaux pour certains cas particuliers.

«v-model» ne prend pas en compte la valeur initiale des attributs `value`, `checked` ou `selected` fournis par un champ. Elle traitera toujours les données de l'instance de vue comme la source de vérité. Vous devez déclarer la valeur initiale dans votre JavaScript, dans l'option `data` de votre composant.

`v-model` utilise en interne différentes propriétés et émetteurs d'évènement pour différents éléments de saisie :

- Les éléments `text` et `textarea` utilisent la propriété `value` et évènement `input`;
- Les éléments `checkboxes` et `radiobuttons` utilisent la propriété `checked` et l'évènement `change`;
- Les éléments `select` utilisent `value` comme une prop et `change` comme un évènement.

Pour les langues qui requièrent une [méthode de saisie (IME)]

([https://fr.wikipedia.org/wiki/M%C3%A9thode_de_saisie_\(IME\)](https://fr.wikipedia.org/wiki/M%C3%A9thode_de_saisie_(IME))) (chinois, japonais, coréen, etc.), vous remarquerez que «v-model» ne sera pas mise à jour durant l'exécution de la méthode de saisie. Si vous souhaitez également prendre en compte ces mises à jour, utilisez plutôt l'évènement «input».

Texte

```
<input v-model="message" placeholder="modifiez-moi">
<p>Le message est : {{ message }}</p>
```

```
<div id="example-1" class="demo">
  <input v-model="message" placeholder="modifiez-moi">
  <p>Le message est : {{ message }}</p>
</div>
<script>
new Vue({
  el: '#example-1',
  data: {
    message: ''
  }
})</script>
```

Texte multiligne

```
<div id="example-textarea" class="demo">
  <span>Le message multiligne est :</span>
```

```

<p style="white-space: pre-line;">{{ message }}</p>
<br>
<textarea v-model="message" placeholder="ajoutez plusieurs lignes">
</textarea>
</div>
<script>
new Vue({
  el: '#example-textarea',
  data: {
    message: ''
  }
})
</script>

```

Checkbox

Checkbox seule, valeur booléenne :

```

<div id="example-2" class="demo">
  <input type="checkbox" id="checkbox" v-model="checked">
  <label for="checkbox">{{ checked }}</label>
</div>
<script>
new Vue({
  el: '#example-2',
  data: {
    checked: false
  }
})
</script>

```

Checkboxes multiples, liées au même tableau (Array) :

```

<div id="example-3" class="demo">
  <input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
  <label for="jack">Jack</label>
  <input type="checkbox" id="john" value="John" v-model="checkedNames">
  <label for="john">John</label>
  <input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
  <label for="mike">Mike</label>
  <br>
  <span>Noms cochés : {{ checkedNames }}</span>
</div>
<script>
new Vue({
  el: '#example-3',
  data: {
    checkedNames: []
  }
})
</script>

```

```
}
```

```
)
```

```
</script>
```

Radio

```
<div id="example-4" class="demo">
  <input type="radio" id="one" value="Un" v-model="picked">
  <label for="one">Un</label>
  <br>
  <input type="radio" id="two" value="Deux" v-model="picked">
  <label for="two">Deux</label>
  <br>
  <span>Choisi : {{ picked }}</span>
</div>
<script>
new Vue({
  el: '#example-4',
  data: {
    picked: ''
  }
})
</script>
```

Select

Select à choix unique :

```
<div id="example-5" class="demo">
  <select v-model="selected">
    <option disabled value="">Choisissez</option>
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <span>Sélectionné : {{ selected }}</span>
</div>
<script>
new Vue({
  el: '#example-5',
  data: {
    selected: ''
  }
})
</script>
```

Si la valeur initiale de votre expression dans `v-model` ne correspond à aucune des options, l'élément `<select>` va faire le rendu dans un état « non sélectionné ».

Select à choix multiples (lié à un tableau) :

```
<div id="example-7" class="demo">
  <select v-model="selected">
    <option v-for="option in options" v-bind:value="option.value">
      {{ option.text }}
    </option>
  </select>
  <span>Sélectionné : {{ selected }}</span>
</div>
<script>
new Vue({
  el: '#example-7',
  data: {
    selected: 'A',
    options: [
      { text: 'Un', value: 'A' },
      { text: 'Deux', value: 'B' },
      { text: 'Trois', value: 'C' }
    ]
  }
})
</script>
```

Liaisons des attributs value

Pour les boutons radio, les cases à cocher et les listes d'options, les valeurs de liaison de `v-model` sont habituellement des chaînes de caractères statiques (ou des booléens pour une case à cocher) :

```
<!-- `picked` sera une chaîne de caractères "a" quand le bouton radio sera
sélectionné -->
<input type="radio" v-model="picked" value="a">

<!-- `toggle` est soit true soit false -->
<input type="checkbox" v-model="toggle">

<!-- `selected` sera une chaîne de caractères "abc" quand la première
option sera sélectionnée -->
<select v-model="selected">
  <option value="abc">ABC</option>
</select>
```

Mais parfois nous pouvons souhaiter lier la valeur à une propriété dynamique de l'instance de Vue. Nous pouvons réaliser cela en utilisant `v-bind`. De plus, utiliser `v-bind` nous permet de lier la valeur de l'input à des valeurs qui ne sont pas des chaînes de caractères.

Checkbox

```
<input  
  type="checkbox"  
  v-model="toggle"  
  true-value="oui"  
  false-value="non"  
>
```

```
// lorsque c'est coché :  
vm.toggle === 'oui'  
// lorsque que c'est décoché :  
vm.toggle === 'non'
```

Radio

```
<input type="radio" v-model="pick" v-bind:value="a">
```

```
// lorsque c'est choisi :  
vm.pick === vm.a
```

Options de select

```
<select v-model="selected">  
  <!-- objet littéral en ligne -->  
  <option v-bind:value="{ number: 123 }">123</option>  
</select>
```

```
// lorsque c'est sélectionné :  
typeof vm.selected // => 'object'  
vm.selected.number // => 123
```

Modificateurs

.lazy

Par défaut, `v-model` synchronise le champ avec les données après chaque évènement `input` (à l'exception de l'exécution d'une méthode de saisie comme [mentionné plus haut](#)). Vous pouvez ajouter le modificateur `lazy` pour synchroniser après les évènements `change` à la place :

```
<!-- synchronisé après le "change" au lieu du "input" -->
<input v-model.lazy=msg >
```

.number

Si vous voulez que la saisie utilisateur soit automatiquement convertie en nombre, vous pouvez ajouter le modificateur **number** à vos champs gérés par **v-model** :

```
<input v-model.number="age" type="number">
```

C'est souvent utile, parce que même avec **type="number"**, la valeur des éléments de saisie HTML retourne toujours une chaîne de caractères. Si la valeur ne peut pas être transformée avec **parseFloat()**, alors la valeur originale est retournée.

.trim

Si vous voulez que les espaces superflus des saisies utilisateur soient automatiquement retirés, vous pouvez ajouter le modificateur **trim** à vos champs gérés par **v-model** :

```
<input v-model.trim=msg>
```

'two-way data binding' vs 'one way data flow'.

v-model avec les composants

Les types de champ standards HTML ne couvriront pas toujours vos besoins. Heureusement, les composants de Vue vous permettent de construire des champs avec un comportement complètement personnalisé. Ces champs fonctionnent même avec v-model !

Propriété calculée bidirectionnelle

Admettons tout de même que l'exemple ci-dessus est plus verbeux que le v-model couplé à l'état local (tout en perdant quelques fonctionnalités pratiques de v-model au passage). Une approche alternative consiste à utiliser une propriété calculée bidirectionnelle avec un mutateur :

```
<input v-model="message">
```

```
// ...
computed: {
  message: {
    get () {
      return this.$store.state.obj.message
    },
    set (value) {
      this.$store.commit('updateMessage', value)
    }
  }
}
```

VueX et les formulaires.

Lorsque l'on utilise Vuex en mode strict, il peut être compliqué d'utiliser `v-model` sur une partie de l'état qui appartient à Vuex :

```
<input v-model="obj.message">
```

Supposons que `obj` est une propriété calculée qui retourne un objet depuis le store, le `v-model` tentera de muter directement `obj.message` lorsque l'utilisateur saisit du texte dans le champ. En mode strict, cela produira une erreur car la mutation n'est pas effectuée dans un gestionnaire de mutation Vuex explicite.

La « méthode Vuex » pour gérer ça est de lier la valeur de l'`input` et d'appeler une action sur l'évènement `input` ou `change` :

```
<input :value="message" @input="updateMessage">
```

```
// ...
computed: {
  ...mapState({
    message: state => state.obj.message
  })
},
methods: {
  updateMessage (e) {
    this.$store.commit('updateMessage', e.target.value)
  }
}
```

Et voici le gestionnaire de mutation :

```
// ...
mutations: {
  updateMessage (state, message) {
    state.obj.message = message
  }
}
```

Validation de la saisie utilisateur.

[Regarder un cours gratuit sur Vue School](#)

La validation des formulaires est supportée nativement par le navigateur. Parfois on va observer des différences sur la manière de gérer la validation en fonction des navigateurs ce qui fait que se reposer sur cette validation supportée nativement est des plus délicat. Même quand la validation est supportée parfaitement, il se peut que quand des validations personnalisées ou plus « manuelles » sont nécessaires, les solutions basées sur Vue soient plus appropriées. Commençons avec un exemple simple.

Pour un formulaire avec trois champs, considérons que deux sont obligatoires. Regardons le HTML d'abord:

```
<form
  id="app"
  @submit="checkForm"
  action="https://vuejs.org/"
  method="post"
>

  <p v-if="errors.length">
    <b>Please correct the following error(s):</b>
    <ul>
      <li v-for="error in errors">{{ error }}</li>
    </ul>
  </p>

  <p>
    <label for="name">Name</label>
    <input
      id="name"
      v-model="name"
      type="text"
      name="name"
    >
  </p>

  <p>
    <label for="age">Age</label>
    <input
      id="age"
      v-model="age"
      type="number"
      name="age"
      min="0"
    >
  </p>

  <p>
    <label for="movie">Favorite Movie</label>
    <select
      id="movie"
      v-model="movie"
      name="movie"
    >
  </p>
```

```

>
<option>Star Wars</option>
<option>Vanilla Sky</option>
<option>Atomic Blonde</option>
</select>
</p>

<p>
<input
  type="submit"
  value="Submit"
>
</p>

</form>

```

Analysons cela à partir en partant du haut. La balise `<form>` a un id que nous utiliserons pour le composant Vue. Il y a un gestionnaire d'évènement à la soumission du formulaire que vous verrez dans un moment, et l'attribut `action` correspond à une URL temporaire qui devrait pointer vers quelque chose de réel sur un serveur (sur lequel vous avez une validation côté serveur bien entendu).

En dessous il y a un paragraphe qui s'affiche ou non en fonction de la présence d'erreurs. C'est une simple liste d'erreurs au-dessus du formulaire. Notez aussi que l'on déclenche la validation à la soumission du formulaire plutôt qu'à la modification de chaque champ.

La dernière chose à remarquer est que chacun des trois champs possède un `v-model` correspondant afin de les connecter aux valeurs sur lesquelles nous travaillerons en JavaScript.

```

const app = new Vue({
  el: '#app',
  data: {
    errors: [],
    name: null,
    age: null,
    movie: null
  },
  methods: {
    checkForm: function (e) {
      if (this.name && this.age) {
        return true;
      }

      this.errors = [];

      if (!this.name) {
        this.errors.push('Name required.');
      }
      if (!this.age) {
        this.errors.push('Age required.');
      }
    }
  }
});

```

```
    }
}
})
```

Relativement court et simple, on définit un tableau pour contenir les erreurs et les valeurs des trois champs du formulaire sont initialisées à `null`. La logique de `checkForm` (qui est activée à la soumission du formulaire) vérifie seulement que name et age ont des valeurs puisque movie est optionnel. Si ce n'est pas le cas, on vérifie chacune d'elles et on ajoute une erreur spécifique quand elles sont nulles. Et c'est tout. Vous pouvez lancer la démo ci-dessous. N'oubliez pas que pour une soumission réussie, cela va générer une requête POST à une URL temporaire.

Utiliser une validation personnalisée.

Pour le second exemple, le deuxième champ de texte (age) est remplacé par un champ d'email qui sera validé par un peu de logique personnalisée. Le code vient de la question StackOverflow , [Comment valider une adresse email en JavaScript?](#). C'est une très bonne question puisqu'elle fait passer votre plus intense discussion politique ou religieuse sur Facebook pour un simple désaccord sur qui fait la meilleure bière. Sérieusement, c'est délirant. Voici le HTML, même si il est très proche du premier exemple.

```
<form
  id="app"
  @submit="checkForm"
  action="https://vuejs.org/"
  method="post"
  novalidate="true"
>

<p v-if="errors.length">
  <b>Please correct the following error(s):</b>
  <ul>
    <li v-for="error in errors">{{ error }}</li>
  </ul>
</p>

<p>
  <label for="name">Name</label>
  <input
    id="name"
    v-model="name"
    type="text"
    name="name"
  >
</p>

<p>
  <label for="email">Email</label>
  <input
    id="email"
    v-model="email"
    type="email"
    name="email"
  >
```

```

>
</p>

<p>
  <label for="movie">Favorite Movie</label>
  <select
    id="movie"
    v-model="movie"
    name="movie"
  >
    <option>Star Wars</option>
    <option>Vanilla Sky</option>
    <option>Atomic Blonde</option>
  </select>
</p>

<p>
  <input
    type="submit"
    value="Submit"
  >
</p>

</form>

```

Bien qu'il y ait peu de différence, remarquez le `novalidate="true"` au début. C'est important car le navigateur va essayer de valider l'adresse email dans le champ quand `type=email` est spécifié. Honnêtement, il est plus logique de faire confiance au navigateur dans ce cas, mais comme nous voulions un exemple personnalisé de validation, nous le désactivons. Voici le JavaScript mis à jour.

```

const app = new Vue({
  el: '#app',
  data: {
    errors: [],
    name: null,
    email: null,
    movie: null
  },
  methods: {
    checkForm: function (e) {
      this.errors = [];

      if (!this.name) {
        this.errors.push("Name required.");
      }
      if (!this.email) {
        this.errors.push('Email required.');
      } else if (!this.validEmail(this.email)) {
        this.errors.push('Valid email required.');
      }

      if (!this.errors.length) {

```

```
        return true;
    }

    e.preventDefault();
},
validEmail: function (email) {
    var re = /^(([^<>()\\[\\]\\\\.,,:\\s@"]+(\.([^\<>()\\[\\]\\\\.,,:\\s@"]+)*|("."+"))@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.)|(([a-zA-Z\\-0-9]+\\.)+[a-zA-Z]{2,})))$/;
    return re.test(email);
}
})
```

Comme vous pouvez le voir, nous avons ajouté une nouvelle méthode `validEmail` qui est simplement appelée par `checkForm`. Vous pouvez jouer avec l'exemple ici:



Autre exemple de validation personnalisée

Pour le troisième exemple, nous avons construit quelque chose que vous avez sûrement déjà vu dans des applications de sondage. L'utilisateur se voit demander de dépenser un budget pour un ensemble de propriétés pour un nouveau modèle de Star Destroyer. Le total doit être de 100. Tout d'abord le HTML.

```
<form
  id="app"
  @submit="checkForm"
  action="https://vuejs.org/"
  method="post"
  novalidate="true"
>
<p v-if="errors.length">
  <b>Please correct the following error(s):</b>
  <ul>
    <li v-for="error in errors">{{ error }}</li>
  </ul>
</p>
```

```

<p>
    Given a budget of 100 dollars, indicate how much
    you would spend on the following features for the
    next generation Star Destroyer. Your total must sum up to 100.
</p>

<p>
    <input
        v-model.number="weapons"
        type="number"
        name="weapons"
    > Weapons <br/>
    <input
        v-model.number="shields"
        type="number"
        name="shields"
    > Shields <br/>
    <input
        v-model.number="coffee"
        type="number"
        name="coffee"
    > Coffee <br/>
    <input
        v-model.number="ac"
        type="number"
        name="ac"
    > Air Conditioning <br/>
    <input
        v-model.number="mousedroids"
        type="number"
        name="mousedroids"
    > Mouse Droids <br/>
</p>

<p>
    Current Total: {{total}}
</p>

<p>
    <input
        type="submit"
        value="Submit"
    >
</p>

</form>

```

Notez l'ensemble des champs pour les cinq propriétés. Remarquez l'ajout de `.number` à la suite de l'attribut `v-model`. Cela dit à Vue de caster la valeur en un nombre quand vous l'utilisez. Il y a cependant un bug avec cette fonctionnalité qui fait que quand la valeur est nulle, cela retourne une chaîne de caractère. Vous verrez comment contourner cela plus bas. Pour faciliter la tâche à l'utilisateur, nous avons ajouté le total en cours juste en bas afin qu'ils puissent le visualiser en temps réel. Maintenant regardons le JavaScript.

```

const app = new Vue({
  el: '#app',
  data: {
    errors: [],
    weapons: 0,
    shields: 0,
    coffee: 0,
    ac: 0,
    mousedroids: 0
  },
  computed: {
    total: function () {
      // must parse because Vue turns empty value to string
      return Number(this.weapons) +
        Number(this.shields) +
        Number(this.coffee) +
        Number(this.ac+this.mousedroids);
    }
  },
  methods: {
    checkForm: function (e) {
      this.errors = [];

      if (this.total != 100) {
        this.errors.push('Total must be 100!');
      }

      if (!this.errors.length) {
        return true;
      }

      e.preventDefault();
    }
  }
})

```

Nous avons défini le total comme une valeur calculée et la méthode checkForm doit maintenant juste vérifier si le total est 100 et c'est tout. Vous pouvez jouer avec ici:

Validation côté serveur

Dans mon dernier exemple, nous allons construire une application vuejs qui utilise Ajax pour valider des données via le serveur. Le formulaire va vous demander de nommer un nouveau produit et ensuite s'assurer que ce nom est unique. Nous avons écrit une rapide [OpenWhisk](#) action sans serveur pour gérer la validation, voici la logique de cette action.

```

function main(args) {
  return new Promise((resolve, reject) => {
    // bad product names: vista, empire, mbp
    const badNames = ['vista', 'empire', 'mbp'];
  })
}

```

```

        if (badNames.includes(args.name)) {
            reject({error: 'Existing product'});
        }

        resolve({status: 'ok'});
    );
}

```

En gros, tous les noms exceptés "vista", "empire", and "mbp" sont valides. Bien, regardons donc le formulaire.

```

<form
  id="app"
  @submit="checkForm"
  method="post"
>

<p v-if="errors.length">
  <b>Please correct the following error(s):</b>
  <ul>
    <li v-for="error in errors">{{ error }}</li>
  </ul>
</p>

<p>
  <label for="name">New Product Name: </label>
  <input
    id="name"
    v-model="name"
    type="text"
    name="name"
  >
</p>

<p>
  <input
    type="submit"
    value="Submit"
  >
</p>

</form>

```

Il n'y a rien de bien spécial ici. Voyons maintenant le JavaScript.

```

const apiUrl =
'https://openwhisk.ng.bluemix.net/api/v1/web/r Camden%40us.ibm.com_My%20Space/safeToDelete/productName.json?name=';

const app = new Vue({

```

```

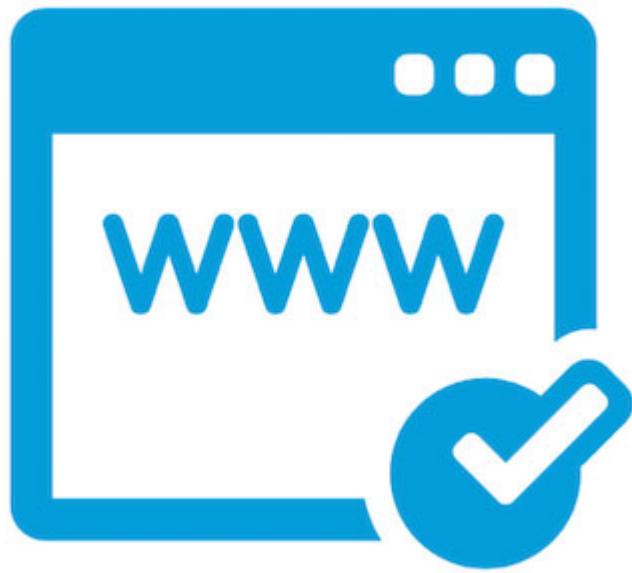
el: '#app',
data: {
  errors: [],
  name: ''
},
methods:{
  checkForm: function (e) {
    e.preventDefault();

    this.errors = [];

    if (this.name === '') {
      this.errors.push('Product name is required.');
    } else {
      fetch(apiUrl + encodeURIComponent(this.name))
        .then(res => res.json())
        .then(res => {
          if (res.error) {
            this.errors.push(res.error);
          } else {
            // redirect to a new URL, or do something on success
            alert('ok!');
          }
        });
    }
  }
}
)

```

On commence par une variable pour l'URL de l'API qui est exécuté sur OpenWhisk. Maintenant, voyons `checkForm`. Dans cette version, nous empêchons le formulaire d'être soumis (ce qui, par ailleurs, pourrait être fait en HTML par Vue). Vous pouvez voir une vérification basique sur la nullité de `this.name` puis on attaque l'API. Si c'est un mauvais nom, on ajoute une erreur comme précédemment. Si c'est bon, dans cet exemple nous ne faisons rien à part une alerte JavaScript, mais vous pouvez renvoyer l'utilisateur vers une nouvelle page avec le nom du produit dans l'URL, ou effectuer d'autres actions.



Aller plus loin avec Vue.js, bonnes pratiques

Aller plus loin avec Vue.js, bonnes pratiques

Consultez les annexes pour découvrir les recommandations et convention d'optimisation de votre code applicatif.

Mixins et plugins.

Mixins

Les mixins offrent une manière flexible de créer des fonctionnalités réutilisables pour les composants de Vue. Un objet mixin peut contenir toute option valide pour un composant. Quand un composant utilise un mixin, toutes les options du mixin seront "fusionnées" avec les options du composant.

Exemple:

```
// définir un objet mixin
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}

// définition d'un composant qui utilise ce mixin
var Component = Vue.extend({
  mixins: [myMixin]
})

var component = new Component() // => "hello from mixin!"
```

Fusion des options

Quand un mixin et un composant définissent les mêmes options, elles seront fusionnées en utilisant la stratégie appropriée.

Par exemple, les données d'un mixin subissant une fusion (une propriété profonde) avec les données d'un composant vont prendre la priorité en cas de conflits.

```
var mixin = {
  data: function () {
    return {
      message: 'bonjour',
      foo: 'abc'
    }
  }
}
```

```

    }

new Vue({
  mixins: [mixin],
  data: function () {
    return {
      message: 'au revoir',
      bar: 'def'
    }
  },
  created: function () {
    console.log(this.$data)
    // => { message: "au revoir", foo: "abc", bar: "def" }
  }
})

```

Les fonctions de hook avec le même nom seront fusionnées dans un tableau afin qu'elles soient toutes appelées. De plus, les hooks des mixins seront appelés **avant** les propres hooks du composant.

```

var mixin = {
  created: function () {
    console.log('hook appelé du mixin')
  }
}

new Vue({
  mixins: [mixin],
  created: function () {
    console.log('hook appelé du composant')
  }
})

// => "hook appelé du mixin"
// => "hook appelé du composant"

```

Les options qui attendent un objet, par exemple **methods**, **components** et **directives**, seront fusionnées dans le même objet. Les options du composant auront la priorité en cas de conflit sur une ou plusieurs clés de ces objets.

```

var mixin = {
  methods: {
    foo: function () {
      console.log('foo')
    },
    conflicting: function () {
      console.log('du mixin')
    }
  }
}

```

```

var vm = new Vue({
  mixins: [mixin],
  methods: {
    bar: function () {
      console.log('bar')
    },
    conflicting: function () {
      console.log('de lui-même')
    }
  }
})

vm.foo() // => "foo"
vm.bar() // => "bar"
vm.conflicting() // => "de lui-même"

```

Notez que les mêmes stratégies de fusion sont utilisées par `Vue.extend()`.

Mixin global

Vous pouvez aussi appliquer un mixin de manière globale. À utiliser avec prudence ! Une fois que vous appliquez un mixin globalement, il modifiera **toutes** les instances de vues créées ensuite. Bien utilisé, cela peut être exploité pour injecter une logique de traitement pour des options personnalisées :

```

// injection d'une fonction pour l'option personnalisée `myOption`
Vue.mixin({
  created: function () {
    var myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

new Vue({
  myOption: 'bonjour !'
})
// => "bonjour !"

```

Utilisez les mixins globaux prudemment et rarement, parce qu'ils affectent chacune des Vue créées, y compris celles des librairies tierces. Dans la plupart des cas, vous devriez uniquement vous en servir pour la gestion des options personnalisées comme illustré dans l'exemple ci-dessus. C'est aussi une bonne idée de les encapsuler dans des [Plugins](plugins.html) pour éviter de les appliquer plusieurs fois par erreur.

Stratégie de fusion des options personnalisées

Quand les options personnalisées sont fusionnées, elles utilisent la stratégie par défaut, qui est simplement d'écraser la valeur existante. Si vous souhaitez appliquer une logique personnalisée pour la fusion d'une option personnalisée, vous devez attacher une nouvelle fonction à `Vue.config.optionMergeStrategies`:

```
Vue.config.optionMergeStrategies.myOption = function (toVal, fromVal) {
  // return mergedVal
}
```

Pour la plupart des options qui attendent des objets, vous pouvez simplement utiliser la stratégie de fusion utilisée par `methods`:

```
var strategies = Vue.config.optionMergeStrategies
strategies.myOption = strategies.methods
```

Un exemple plus avancé peut être trouvé dans la stratégie de fusion de [Vuex](#) 1.x :

```
const merge = Vue.config.optionMergeStrategies.computed
Vue.config.optionMergeStrategies.vuex = function (toVal, fromVal) {
  if (!toVal) return fromVal
  if (!fromVal) return toVal
  return {
    getters: merge(toVal.getters, fromVal.getters),
    state: merge(toVal.state, fromVal.state),
    actions: merge(toVal.actions, fromVal.actions)
  }
}
```

Plugins.

Les plugins sont habituellement ajoutés au niveau des fonctionnalités globales de Vue. Il y a un cadre strictement défini pour un plugin et il y a divers types de plugins que vous pouvez écrire pour :

1. Ajouter plusieurs méthodes globales ou propriétés. Par ex. [vue-custom-element](#)
2. Ajouter une ou plusieurs ressources globales : directives, filters, transitions, etc. Par ex. [vue-touch](#)
3. Ajouter plusieurs options de composant avec un mixin global. Par ex. [vue-router](#)
4. Ajouter des méthodes d'instance de Vue en les reliant au prototype de Vue.
5. Fournir une bibliothèque qui fournit sa propre API, et qui en même temps injecte une combinaison des points précédents. Par ex. [vue-router](#)

Utiliser un plugin

Utiliser un plugin en appelant la méthode globale `Vue.use()` :

```
// appelle `MyPlugin.install(Vue)`
Vue.use(MyPlugin)
```

Vous pouvez optionnellement passer certaines options :

```
Vue.use(MyPlugin, { someOption: true })
```

`Vue.use` empêchera automatiquement l'utilisation du même plugin plusieurs fois. Ainsi, appeler cette fonction plusieurs fois sur le même plugin n'installera le plugin qu'une seule fois.

Certains plugins fournis officiellement par Vue.js comme `vue-router` appellent automatiquement `Vue.use()` si `Vue` est disponible en tant que variable globale. Cependant, dans un environnement modulaire comme avec CommonJS, vous devrez toujours appeler explicitement `Vue.use()` :

```
// En utilisant CommonJS depuis Browserify ou webpack
var Vue = require('vue')
var VueRouter = require('vue-router')

// N'oubliez pas de l'appeler
Vue.use(VueRouter)
```

Consultez [awesome-vue](#) pour une large collection de plugins et bibliothèques fournis par la contribution de la communauté.

Écrire un plugin

Un plugin Vue.js devrait exposer une méthode `install`. Cette méthode sera appelée avec le constructeur de `Vue` en tant que premier argument, avec les options possibles suivantes :

```
MyPlugin.install = function (Vue, options) {
    // 1. ajouter une méthode globale ou une propriété
    Vue.myGlobalMethod = function () {
        // de la logique de code...
    }

    // 2. ajouter une ressource globale
    Vue.directive('my-directive', {
        bind (el, binding, vnode, oldVnode) {
            // de la logique de code...
        }
        ...
    })

    // 3. injecter des options de composant
    Vue.mixin({
        created: function () {
            // de la logique de code...
        }
        ...
    })
}

// 4. ajouter une méthode d'instance
Vue.prototype.$myMethod = function (methodOptions) {
    // de la logique de code...
}
```

Regrouper son code dans des Single File Components .vue.

Dans beaucoup de projets Vue, des composants globaux seront définis en utilisant `Vue.component`, suivi de `new Vue({ el: '#container' })` pour cibler un élément conteneur dans le corps de chaque page.

Cela peut très bien fonctionner pour des petits projets ou des projets de taille moyenne, pour lesquels JavaScript est utilisé uniquement pour améliorer certaines vues. Cependant, pour des projets plus complexes, ou bien quand votre front-end est entièrement généré par JavaScript, des désavantages se manifestent :

- **Les définitions globales** forcent à avoir un nom unique pour chaque composant
- **Les templates sous forme de chaines de caractères** ne bénéficient pas de la coloration syntaxique et requièrent l'usage de slashes disgracieux pour le HTML multiligne.
- **L'absence de support pour le CSS** signifie que le CSS ne peut pas être modularisé comme HTML et JavaScript
- **L'absence d'étape de build** nous restreint au HTML et à JavaScript ES5, sans pouvoir utiliser des préprocesseurs tels que Babel ou Pug (anciennement Jade).

Tous ces désavantages sont résolus par les **composants monofichiers** avec une extension `.vue`, rendus possibles par les outils de *build* tels que webpack ou Browserify.

Voici un exemple simple de fichier que nous appellerons `Hello.vue` :

```
<template>
<p>{{ greeting }} World!</p>
</template>

<script>
module.exports = {
  data: function () {
    return {
      greeting: 'Hello'
    }
  }
}
</script>

<style scoped>
p {
  font-size: 2em;
  text-align: center;
}
</style>
```

Maintenant nous avons :

- Une coloration syntaxique complète
- Des modules CommonJS
- Du CSS dont la portée est limitée au composant

Et comme promis, nous pouvons aussi utiliser des préprocesseurs tels que Pug, Babel (avec les modules ES2015), et Stylus pour obtenir des composants plus lisibles et plus riches en fonctionnalités.

```
< > Hello.vue x ...<template lang="jade">
  div
    p {{ greeting }} World!
    other-component
</template>

<script>
  import OtherComponent from './OtherComponent.vue'

  export default {
    data () {
      return {
        greeting: 'Hello'
      }
    },
    components: {
      OtherComponent
    }
  }
</script>

<style lang="stylus" scoped>
  p
    font-size 2em
    text-align center
</style>
```

Line 27, Column 1 Spaces: 2 Vue Component

Ces langages spécifiques ne sont que des exemples; vous pourriez tout aussi aisément utiliser Bublé, Typescript, SCSS, PostCSS - ou tout autre préprocesseur qui vous aide à être productif. Si vous utilisez webpack avec vue-loader, vous aurez aussi un outil de premier choix pour les modules CSS.

Qu'en est-il de la séparation des préoccupations ?

Une chose importante à souligner est que **la séparation des préoccupations** (« Separation of concerns ») **n'est pas identique à la séparation des fichiers**. Dans le développement des interfaces utilisateur modernes, nous avons constaté que plutôt que de diviser tout notre code en trois grosses couches distinctes interdépendantes, il était plus intuitif de le diviser en petits composants faiblement couplés, et de les combiner. Au sein d'un composant, son template, sa logique et ses styles sont intrinsèquement couplés, et les réunir rend en réalité le composant plus cohérent et facile à maintenir.

Si vous n'aimez pas l'idée des composants monofichiers, vous pouvez toujours tirer parti du rechargement à chaud et la précompilation pour mettre le CSS et le JavaScript dans des fichiers séparés.

```
<!-- my-component.vue -->
<template>
  <div>This will be pre-compiled</div>
</template>
<script src="./my-component.js"></script>
<style src="./my-component.css"></style>
```

Pour les utilisateurs qui ne connaissent pas les systèmes de *build* de modules en JavaScript

Avec les composants **.vue**, nous entrons de plain-pied dans le domaine des applications JavaScript avancées. Cela implique d'apprendre à utiliser quelques nouveaux outils si vous ne les connaissez pas déjà :

- **Node Package Manager (npm)**: lisez le guide npm [Getting Started guide](#) section *10: Uninstalling global packages*.
- **JavaScript moderne avec ES2015/16**: lisez le guide Babel [Learn ES2015 guide](#). Vous n'avez pas besoin de mémoriser chacune des fonctionnalités maintenant, mais gardez cette page en référence pour pouvoir y revenir.

Une fois que vous aurez pris une journée pour vous plonger dans ces ressources, nous vous recommandons d'essayer [Vue CLI 3](#). Suivez les instructions et vous devriez avoir en un clin d'œil un projet Vue avec des composants **.vue**, ES2015 et le rechargement à chaud !

Pour les utilisateurs avancés

La CLI prend soin de la configuration de la plupart des outils pour vous, mais vous permet tout de même une optimisation granulaire avec vos propres [options de configuration](#).

Dans le cas où vous préfériez créer votre installation depuis le départ, vous pourrez configurer manuellement webpack avec [vue-loader](#). Pour en apprendre plus à propos de webpack en lui-même, consultez la [documentation officielle](#) et la [Webpack Academy](#).

Créer des custom directives.

En supplément de l'ensemble de directives fournies par défaut (`v-model` et `v-show`), Vue vous permet également d'enregistrer vos propres directives. Notez qu'avec Vue 2.0, les composants sont la forme principale de réutilisabilité et d'abstraction du code. Il y a cependant des cas où vous aurez juste besoin d'un accès de bas niveau aux éléments du DOM, et c'est là que les directives personnalisées vous seraient utiles. Un exemple pourrait être la prise du focus sur un élément de champ, comme celui-ci :

```
<div id="simplest-directive-example" class="demo">
  <input v-focus>
</div>
<script>
Vue.directive('focus', {
  inserted: function (el) {
    el.focus()
  }
})
new Vue({
  el: '#simplest-directive-example'
})
</script>
```

Quand la page se charge, cet élément prend le focus (notez que `autofocus` ne fonctionne pas sur Safari mobile). En fait, si vous n'avez cliqué sur rien du tout depuis votre arrivée sur la page, le champ ci-dessus devrait avoir le focus. À présent, jetons un œil à la directive qui pourrait accomplir cela :

```
// Enregistrer une directive globale appelée `v-focus`
Vue.directive('focus', {
  // Quand l'élément lié est inséré dans le DOM...
  inserted: function (el) {
    // L'élément prend le focus
    el.focus()
  }
})
```

Si vous préférez enregistrer à la place la directive en local, les composants acceptent également l'option `directives` :

```
directives: {
  focus: {
    // définition de la directive
    inserted: function (el) {
      el.focus()
    }
  }
}
```

Puis dans un template, vous pouvez utiliser le nouvel attribut **v-focus** sur n'importe quel élément, comme celui-ci :

```
<input v-focus>
```

Fonctions de hook

Un objet de définition de directive peut fournir plusieurs fonctions de hook (toutes optionnelles) :

- **bind** : appelée une seule fois quand la directive est attachée à l'élément. C'est ici que vous pouvez effectuer les actions uniques d'initialisation.
- **inserted**: appelée quand l'élément lié a été inséré dans son nœud parent (cela garantit uniquement sa présence dans le nœud parent, mais pas nécessairement dans le document principal).
- **update**: appelée après que le composant conteneur VNode ait été mis à jour, **mais possiblement avant que ses enfants aient été mis à jour**. La valeur de la directive peut ou pas avoir changé, mais vous pouvez ignorer les mises à jour inutiles en comparant les valeurs actuelles et anciennes de la liaison (voir plus bas les arguments de hook).

Nous couvrirons VNodes plus en détail [plus tard](./render-function.html#DOM-virtuel), quand nous discuterons des [fonctions de rendu](./render-function.html).

- **componentUpdated**: appelée après que le composant conteneur VNode **et les VNode de ses enfants** aient été mis à jour.
- **unbind**: appelée uniquement une fois, quand la directive est déliée de l'élément.

Nous allons explorer les arguments passés à ces hooks (c.-à-d. **el**, **binding**, **vnode**, et **oldVnode**) dans la prochaine section.

Arguments des hooks d'une directive

Les hooks d'une directive ont accès à ces arguments :

- **el** : l'élément sur lequel la directive est liée. Cela peut être utilisé pour directement manipuler le DOM.
- **binding** : un objet contenant les propriétés suivantes.
 - **name** : le nom de la directive, sans le préfixe **v-**.
 - **value** : la valeur passée à la directive. Par exemple dans **v-my-directive="1 + 1"**, la valeur serait **2**.
 - **oldValue** : la valeur précédente, seulement disponible dans **update** et **componentUpdated**. Elle est disponible, que la valeur ait changé ou non.
 - **expression** : l'expression liée en tant que chaîne de caractères. Par exemple dans **v-my-directive="1 + 1"**, l'expression serait **"1 + 1"**.
 - **arg** : l'argument passé à la directive, s'il y en a une. Par exemple dans **v-my-directive:foo**, l'argument serait **"foo"**.
 - **modifiers** : un objet contenant les modificateurs, s'il y en a. Par exemple dans **v-my-directive.foo.bar**, l'objet des modificateurs serait **{ foo: true, bar: true }**.
- **vnode** : le nœud virtuel produit par le compilateur Vue. Voir l'[API VNode](#) pour tous les détails.

- **oldVnode** : le nœud virtuel précédent, seulement disponible dans les hooks **update** et **componentUpdated**.

À l'exception de `el`, vous devez traiter ces arguments comme étant en lecture seule (« read-only ») et ne jamais les modifier. Si vous souhaitez partager des informations entre les hooks, il est recommandé de le faire à travers les attributs de données sur mesure de ses éléments (voir [dataset](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dataset)).

Un exemple de directive personnalisée utilisant plusieurs de ces propriétés :

```
<div id="hook-arguments-example" v-demo:foo.a.b="message" class="demo">
</div>
<script>
Vue.directive('demo', {
  bind: function (el, binding, vnode) {
    var s = JSON.stringify
    el.innerHTML =
      'name: ' + s(binding.name) + '<br>' +
      'value: ' + s(binding.value) + '<br>' +
      'expression: ' + s(binding.expression) + '<br>' +
      'argument: ' + s(binding.arg) + '<br>' +
      'modifiers: ' + s(binding.modifiers) + '<br>' +
      'vnode keys: ' + Object.keys(vnode).join(', ')
  }
})
new Vue({
  el: '#hook-arguments-example',
  data: {
    message: 'bonjour !'
  }
})
</script>
```

Arguments de directive dynamiques

Les arguments de directive peuvent être dynamiques. Par exemple, dans **v-mydirective:[argument]="value"**, l'**argument** peut-être mis à jour en se basant sur la propriété de donnée dans notre instance de composant ! Cela rend nos directives personnalisées flexibles à travers notre application.

Imaginons que vous vouliez créer une directive personnalisée qui vous permet d'attacher des éléments sur votre page en utilisant le positionnement fixe. Nous pourrions créer une directive personnalisée ou la valeur met à jour le positionnement vertical en nombre de pixel, comme ceci :

```
<div id="baseexample">
  <p>Faite défiler la page vers le bas ↓</p>
  <p v-pin="200">Je suis attaché à 200px depuis le haut de page.</p>
</div>
```

```

Vue.directive('pin', {
  bind: function (el, binding, vnode) {
    el.style.position = 'fixed'
    el.style.top = binding.value + 'px'
  }
})

new Vue({
  el: '#baseexample'
})

```

Cela va attacher l'élément à 200px depuis le haut de la page. Mais que ce passe t-il si nous sommes dans un scénario où nous avons besoin d'attacher l'élément sur la gauche à la place du haut ? Ici nous avons un argument dynamique qui peut être mis à jour pour chaque instance du composant de manière très pratique :

```

<div id="dynamicexample">
  <h3>Faite défiler la page vers le bas ↓</h3>
  <p v-pin:[direction]="200">Je suis attaché à 200px depuis la gauche de
la page.</p>
</div>

```

```

Vue.directive('pin', {
  bind: function (el, binding, vnode) {
    el.style.position = 'fixed'
    var s = (binding.arg == 'left' ? 'left' : 'top')
    el.style[s] = binding.value + 'px'
  }
})

new Vue({
  el: '#dynamicexample',
  data: function () {
    return {
      direction: 'left'
    }
  }
})

```

Fonction abrégée

Dans de nombreux cas, vous pourriez vouloir le même comportement pour les hooks **bind** et **update**, sans avoir besoin des autres hooks. Par exemple :

```

Vue.directive('color-swatch', function (el, binding) {
  el.style.backgroundColor = binding.value
})

```

Objets littéraux

Si votre directive a besoin de plusieurs valeurs, vous pouvez également passer un objet JavaScript. Souvenez-vous, une directive peut accepter n'importe quelle expression JavaScript.

```
<div v-demo="{ color: 'white', text: 'bonjour !' }"></div>
```

```
Vue.directive('demo', function (el, binding) {
  console.log(binding.value.color) // => "white"
  console.log(binding.value.text) // => "bonjour !"
})
```

Typage flow/TypeScript.

Vue CLI fournit des outils de support à TypeScript.

Déclaration officielle dans les packages npm

Un système de typage statique peut aider à prévenir des erreurs d'exécutions potentielles, et particulièrement quand les applications grandissent. C'est pourquoi Vue est fourni avec des [déclarations de types officielles](#) pour [TypeScript](#), et pas seulement pour le cœur de Vue, mais aussi pour [vue-router](#) et [vuex](#).

Puisque ceux-ci sont [publiés sur npm](#), et que la dernière version de TypeScript sait comment résoudre des déclarations de type dans des packages npm, cela signifie qu'installer ceux-ci via npm ne requiert aucun outil supplémentaire pour utiliser TypeScript avec Vue.

Configuration recommandée

```
// tsconfig.json
{
  "compilerOptions": {
    // alignement avec le support navigateur de Vue
    "target": "es5",
    // activation de la déduction stricte pour les propriétés de données
    // sur `this`
    "strict": true,
    // si vous utilisez webpack 2+ ou rollup, permettre le tree shaking :
    "module": "es2015",
    "moduleResolution": "node"
  }
}
```

Notez que vous devez inclure `strict: true` (ou au moins `noImplicitThis: true` qui est une partie de `strict`) pour activer la vérification de type de `this` dans les méthodes de composant, autrement il sera toujours traité comme un type `any`.

Voir [les options de compilation TypeScript](#) pour plus de détails.

Outils de développement

Création de projet

Vue CLI 3 peut générer de nouveaux projets qui utilisent TypeScript. Pour commencer :

```
# 1. Installer Vue CLI s'il n'est pas déjà installé
npm install --global @vue/cli

# 2. Créer un nouveau projet et choisir l'option "Manually select
# features"
vue create my-project-name
```

Support d'édition

Pour développer des applications Vue avec TypeScript, nous recommandons fortement d'utiliser [Visual Studio Code](#) qui fournit un support de TypeScript nativement. Si vous utilisez des [composants monofichiers](#), utilisez la super [extension Vetur](#) qui fournit des déductions TypeScript à l'intérieur de vos composants monofichiers et bien d'autres fonctionnalités extras.

[WebStorm](#) fournit également un support de base pour TypeScript et Vue.js.

Utilisation de base

Pour laisser TypeScript déduire proprement les types dans les options des composants Vue, vous devez définir vos composants avec [Vue.component](#) ou [Vue.extend](#) :

```
import Vue from 'vue'

const Component = Vue.extend({
  // déduction de type activée
})

const Component = {
  // ceci N'aura PAS la déduction de type,
  // car TypeScript ne peut pas savoir qu'il s'agit d'options pour un
  // composant Vue.
}
```

Composants Vue basés sur les classes

Si vous préférez une API basée sur les classes quand vous déclarez des composants, vous pouvez utiliser le décorateur officiel [vue-class-component](#) :

```
import Vue from 'vue'
import Component from 'vue-class-component'

// Le décorateur @Component indique que la classe est un composant Vue
@Component({
  // Toutes les options de composant sont autorisées ici.
  template: '<button @click="onClick">Click!</button>'
})
export default class MyComponent extends Vue {
  // Les données initiales peuvent être déclarées comme des propriétés de
  // l'instance
  message: string = 'Bonjour !'

  // Les méthodes peuvent être déclarées comme des méthodes d'instance
  onClick (): void {
    window.alert(this.message)
  }
}
```

Déclaration des types des plugins Vue

Les plugins peuvent ajouter des propriétés d'instance de Vue, des propriétés globales de Vue et des options de composant de Vue. Dans ces cas, les déclarations de type sont nécessaires pour permettre aux plugins de compiler en TypeScript. Fort heureusement, il y a une fonctionnalité TypeScript pour augmenter les types existants appelée [module d'augmentation](#).

Par exemple, pour déclarer une propriété d'instance `$myProperty` avec le type `string` :

```
// 1. Assurez-vous d'importer `vue` avant de déclarer les types augmentés
import Vue from 'vue'

// 2. Spécifiez un fichier avec les types que vous voulez augmenter
//     Vue a le type de constructeur dans types/vue.d.ts
declare module 'vue/types/vue' {
    // 3. Déclarez l'augmentation pour Vue
    interface Vue {
        $myProperty: string
    }
}
```

Après inclusion du code ci-dessus en tant que déclaration de fichier (comme `my-property.d.ts`) dans votre projet, vous pouvez utiliser `$myProperty` dans une instance de Vue.

```
var vm = new Vue()
console.log(vm.$myProperty) // Ceci sera compilé avec succès
```

Vous pouvez aussi déclarer des propriétés globales additionnelles et des options de composant :

```
import Vue from 'vue'

declare module 'vue/types/vue' {
    // Les propriétés globales peuvent être déclarées
    // sur l'interface `VueConstructor`
    interface VueConstructor {
        $myGlobal: string
    }
}

// `ComponentOptions` est déclarée dans types/options.d.ts
declare module 'vue/types/options' {
    interface ComponentOptions<V extends Vue> {
        myOption?: string
    }
}
```

La déclaration ci-dessus permet au code suivant de compiler :

```
// Propriété globale
console.log(Vue.$myGlobal)

// Option additionnelle de composant
var vm = new Vue({
  myOption: 'Hello'
})
```

Annotation des types de retour

Du fait de la nature circulaire de la déclaration des fichiers Vue, TypeScript peut avoir des difficultés à deviner les types de certaines méthodes. Pour ces raisons, vous devriez annoter les types de retour des méthodes comme `render` et ceux dans `computed`.

```
import Vue, { VNode } from 'vue'

const Component = Vue.extend({
  data () {
    return {
      msg: 'Bonjour'
    }
  },
  methods: {
    // Besoin d'une annotation car `this` fait parti du type de retour
    greet (): string {
      return this.msg + ' world'
    }
  },
  computed: {
    // Besoin d'une annotation
    greeting(): string {
      return this.greet() + '!'
    }
  },
  // `createElement` est deviné, mais `render` à besoin d'une annotation
  // de type de retour
  render (createElement): VNode {
    return createElement('div', this.greeting)
  }
})
```

Si vous vous apercevez que l'autocomplétion ne fonctionne pas, annoter certaines méthodes peut aider à résoudre ces problèmes. Utiliser l'option `--noImplicitAny` aidera à trouver bon nombre de ces méthodes non annotées.

Rendre son code robuste grâce aux Prop types.

Validation des props

Les composants peuvent spécifier des conditions requises pour leurs props, comme les types que nous avons déjà vus. Si les conditions requises ne sont pas validées, Vue va vous en avertir dans la console JavaScript du navigateur. Cela est particulièrement utile quand on développe un composant destiné à être utilisé par les autres.

Pour spécifier des validations de prop, vous pouvez fournir un objet avec les conditions de validation pour les valeurs des `props` plutôt qu'un tableau de chaîne de caractère.

Par exemple :

```
Vue.component('my-component', {
  props: {
    // Vérification de type basique (`null` et `undefined` valide
    // n'importe quel type)
    propA: Number,
    // Multiple types possibles
    propB: [String, Number],
    // Chaîne de caractères nécessaire
    propC: {
      type: String,
      required: true
    },
    // Nombre avec une valeur par défaut
    propD: {
      type: Number,
      default: 100
    },
    // Objet avec une valeur par défaut
    propE: {
      type: Object,
      // Les objets et tableaux par défaut doivent être retournés depuis
      // une fonction de fabrique
      default: function () {
        return { message: 'hello' }
      }
    },
    // Fonction de validation personnalisée
    propF: {
      validator: function (value) {
        // La valeur passée doit être l'une de ces chaînes de caractères
        return ['success', 'warning', 'danger'].indexOf(value) !== -1
      }
    }
})
```

Quand la validation de prop échoue, Vue va lancer un avertissement dans la console (si vous utilisez le Build de développement)

Notez que les props sont validées **avant** que l'instance du composant soit créée. Donc les propriétés d'instances (par ex. `data`, `computed`, etc) ne seront pas disponibles dans les fonctions `default` ou `validator`.

Vérification de types

Le **type** peut être l'un des constructeurs natifs suivants :

- String
- Number
- Boolean
- Array
- Object
- Date
- Function
- Symbol

De plus, **type** peut aussi être une fonction constructeur personnalisée et l'assertion pourra être testée avec **instanceof**. Par exemple, avec la fonction constructeur suivante :

```
function Person (firstName, lastName) {  
  this.firstName = firstName  
  this.lastName = lastName  
}
```

Vous pourrez utiliser :

```
Vue.component('blog-post', {  
  props: {  
    author: Person  
  }  
})
```

pour valider que la valeur de la prop **author** est bien créée avec un **new Person**.

Compilation & optimisation des performances.

La plupart des tips ci-dessous sont activé par défaut si vous utilisez [Vue CLI](#). Cette section est seulement pertinente si vous utilisez des installations personnalisées.

Passer en mode production

Pendant le développement, Vue fournit beaucoup d'avertissements pour vous aider avec les erreurs usuelles et les pièges. Ces messages d'avertissement deviennent cependant inutiles en production et gonflent la taille de votre application en production. De plus, certains de ces avertissements ont un léger cout d'exécution qui peut être évité en mode production.

Sans outils de build

Si vous utilisez la version complète, c.-à-d. en incluant directement Vue via une balise script sans outil de build, assurez-vous d'utiliser la version minifiée (`vue.min.js`) pour la production. Ces deux versions peuvent être trouvées dans le [guide d'installation](#).

Avec outils de build

Quand vous utilisez un outil de build comme webpack ou Browserify, le mode production sera déterminé par la valeur de `process.env.NODE_ENV` utilisée dans le code source de Vue, et sera en mode développement par défaut. Ces deux outils fournissent des moyens de surcharger cette variable pour activer le mode production de Vue, et les avertissements seront retirés par le minificateur pendant le build. Tous les templates `vue-cli` sont préconfigurés pour vous, mais il peut être utile de savoir comment cela fonctionne :

webpack

Dans webpack 4+, vous pouvez utiliser l'option `mode` :

```
module.exports = {
  mode: 'production'
}
```

Mais dans webpack 3 et précédent, vous devez utiliser la fonction `DefinePlugin` :

```
var webpack = require('webpack')

module.exports = {
  // ...
  plugins: [
    // ...
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify('production')
    })
  ]
}
```

Browserify

- Lancer votre commande d'empaquetage avec la variable d'environnement `NODE_ENV` assignée à `"production"`. Cela indique à `vueify` d'éviter d'inclure le code utilisé pour le rechargement à chaud ou lié au développement.
- Appliquer une transformation `envify` globale à votre paquetage (« bundle »). Cela permet au minificateur de retirer tous les avertissements dans le code source de Vue qui sont entourés d'une instruction conditionnelle sur la variable `env`. Par exemple :

```
NODE_ENV=production browserify -g envify -e main.js | uglifyjs -c -m > build.js
```

- Ou, utiliser `envify` avec gulp :

```
// Utiliser le module personnalisé envify pour spécifier les variables d'environnement
var envify = require('envify/custom')

browserify(browserifyOptions)
  .transform(vueify)
  .transform(
    // Requis sur les fichiers dans `node_modules`
    { global: true },
    envify({ NODE_ENV: 'production' })
  )
  .bundle()
```

- Ou, utilisez `envify` avec Grunt et `grunt-browserify` :

```
// Utiliser le module personnalisé envify pour spécifier les variables d'environnement
var envify = require('envify/custom')

browserify: {
  dist: {
    options: {
      // Fonction à dévier de l'ordre par défaut de `grunt-browserify`
      configure: b => b
        .transform('vueify')
        .transform(
          // Demande un ordre pour exécuter les fichiers
          `node_modules`
            { global: true },
            envify({ NODE_ENV: 'production' })
        )
        .bundle()
    }
  }
}
```

```
    }  
}
```

Rollup

Utiliser le plugin [rollup-plugin-replace](#):

```
const replace = require('rollup-plugin-replace')  
  
rollup({  
  // ...  
  plugins: [  
    replace({  
      'process.env.NODE_ENV': JSON.stringify( 'production' )  
    })  
  ]  
}).then(...)
```

Templates précompilés

Quand vous utilisez des templates dans le DOM ou dans des chaînes de caractères en JavaScript, une compilation des templates vers les fonctions de rendu est exécutée à la volée. C'est habituellement assez rapide dans la plupart des cas, mais il vaut mieux l'éviter si la performance de vos applications est quelque chose de critique.

La manière la plus simple de précompiler les templates est d'utiliser les [composants monofichiers](#). Les outils de build associés effectuent la précompilation pour vous, afin que le code produit contienne les fonctions de rendu déjà compilées au lieu des templates en chaînes de caractères à l'état brut.

Si vous utilisez webpack, et préférez séparer le JavaScript des fichiers de template, vous pouvez utiliser [vue-template-loader](#), qui transforme également les fichiers de template en fonctions de rendu pendant la phase de build.

Extraire le CSS des composants

Lors de l'utilisation de composants monofichiers, le CSS à l'intérieur des composants est injecté dynamiquement sous la forme de balises `<style>` via JavaScript. Ceci a un léger cout d'exécution, et si vous utilisez du rendu côté serveur, cela causera le phénomène de FOUC (« Flash Of Unstyled Content »). Extraire le CSS depuis tous les composants dans le même fichier évite ces problèmes et amène à une meilleure minification et mise en cache du CSS.

Référez-vous aux documentations respectives des outils de build pour voir comment cela est fait :

- [webpack + vue-loader](#) (le template webpack [vue-cli](#) a cela préconfiguré)
- [Browserify + vueify](#)
- [Rollup + rollup-plugin-vue](#)

Suivre les erreurs d'exécution

Si une erreur d'exécution est levée pendant le rendu d'un composant, elle sera passée à la fonction de configuration globale `Vue.config.errorHandler` si elle a été affectée. C'est toujours une bonne idée de coupler ce hook avec un service de suivi d'erreur comme [Sentry](#), qui fournit [une intégration officielle](#) pour Vue.

Server Side Rendering.

Le guide complet du rendu côté serveur

Nous avons créé un guide dédié à la création d'applications Vue utilisant du rendu côté serveur. C'est un guide avancé pour ceux qui sont déjà habitués au développement côté client avec Vue, au développement serveur avec Node.js et à webpack. Vous pouvez y accéder à l'adresse ssr.vuejs.org.

Nuxt.js

Configurer proprement tous les aspects dont nous avons discuté pour un rendu côté serveur prêt pour la production peut être intimidant. Fort heureusement, il y a un excellent projet communautaire dont le but est de rendre tout cela plus simple : [Nuxt.js](#). Nuxt.js est un framework de haut niveau construit par-dessus l'écosystème de Vue et qui fournit une expérience de développement toute tracée pour écrire des applications Vue universelles. Encore mieux, vous pouvez vous en servir comme générateur de sites web statiques (avec des pages écrites comme des composants Vue monofichiers) ! Nous vous recommandons grandement de l'essayer.

Quasar Framework SSR + PWA

Le [Quasar Framework](#) va générer une application SSR (avec un transfert PWA) qui tire parti de son système de build de premier ordre, une configuration sensible et une extensibilité pour développeur afin de faciliter la conception et la construction de votre idée. Avec plus de cent composants spécifiques au "Material Design 2.0", vous pouvez choisir ceux à exécuter sur le serveur, disponibles dans le navigateur, et même gérer les balises `<meta>` de votre site. Quasar est un environnement de développement basé sur Node.js et webpack, qui rationalise et rationalise le développement rapide des applications SPA, PWA, SSR, Electron et Cordova, le tout à partir d'une seule base de code.

Développement d'applications mobiles.

<https://vue-native.io/>

Vue Native est un framework mobile pour construire une application mobile vraiment native en utilisant Vue.js. Son est conçu pour connecter React Native et Vue.js.

Vue Native est un wrapper autour des API React Native, qui vous permet d'utiliser Vue.js et de composer une riche interface utilisateur mobile.



Annexe(s).

Annexe(s).

Annexe .1 : API Vue

Configuration globale

`Vue.config` est un objet contenant les configurations globales de Vue. Vous pouvez modifier les propriétés listées ci-dessous avant de mettre en place votre application :

silent

- **Type :** `boolean`
- **Par défaut :** `false`
- **Utilisation :**

```
Vue.config.silent = true
```

Supprime tous les logs et warnings de Vue.js.

optionMergeStrategies

- **Type :** `{ [key: string]: Function }`
- **Par défaut :** `{}`
- **Utilisation :**

```
Vue.config.optionMergeStrategies._mon_option = function (parent,
enfant, vm) {
    return enfant + 1
}

const Profil = Vue.extend({
    _mon_option: 1
})

// Profil.options._mon_option = 2
```

Définit des stratégies personnalisées de fusion pour les options.

La stratégie de fusion reçoit en arguments la valeur de cette option définie dans le parent et les instances enfants en tant que premier et second argument, respectivement. L'instance de Vue est passée en troisième argument.

- **Voir aussi :** [Stratégie de fusion des options personnalisées](#)

devtools

- **Type :** `boolean`

- **Par défaut :** `true` (`false` dans les versions de production)

- **Utilisation :**

```
// assurez-vous d'assigner ça de manière synchrone immédiatement après
// avoir chargé Vue
Vue.config.devtools = true
```

Autorise ou non l'inspection des [vue-devtools](#). Cette option a comme valeur par défaut `true` dans les versions de développement et `false` dans les versions de production. Vous pouvez l'assigner à `true` pour activer l'inspection avec les versions de production.

errorHandler

- **Type :** `Function`

- **Par défaut :** `undefined`

- **Utilisation :**

```
Vue.config.errorHandler = function (err, vm, info) {
  // gérer le cas d'erreur `info` est une information spécifique
  // à Vue sur l'erreur, par exemple dans quel hook du cycle de vie
  // l'erreur a été trouvée. Disponible uniquement en 2.2.0+
}
```

Définit un gestionnaire pour les erreurs non interceptées pendant le rendu d'un composant et les appels aux observateurs. Ce gestionnaire sera appelé avec comme arguments l'erreur et l'instance de Vue associée.

En 2.2.0+, ce hook capture également les erreurs dans les hooks du cycle de vie des composants.

De plus, quand ce hook est `undefined`, les erreurs capturées seront loguées avec `console.error` plutôt qu'avoir un crash de l'application.

En 2.4.0+ ce hook capture également les erreurs lancées depuis un gestionnaire d'évènement Vue personnalisé.

En 2.6.0+, ce hook capture également les erreurs lancées depuis les écouteurs de DOM `v-on`. De plus, si les hooks couverts ou gestionnaires retournent une chaîne de Promesse (par exemple les fonctions `async`), l'erreur sera également remontée depuis cette Promesse.

[Sentry](#) et [Bugsnag](#) fournissent une intégration officielle utilisant cette option.

warnHandler

Nouveau dans la 2.4.0+

- **Type :** `Function`

- ***Par défaut :** `undefined`

- **Utilisation :**

```
Vue.config.warnHandler = function (msg, vm, trace) {
  // `trace` est la trace de hiérarchie de composant
}
```

Assigne un gestionnaire personnalisé pour les avertissements à l'exécution de Vue. Notez que cela n'est fonctionnel qu'en développement et est ignoré en production.

ignoredElements

- **Type :** `Array<string | RegExp>`
- **Par défaut :** `[]`
- **Utilisation :**

```
Vue.config.ignoredElements = [
  'mon-web-component',
  'un-autre-web-component',
  // Utilisez une `RegExp` pour ignorer tous les éléments qui
  commencent par « ion- »
  // 2.5+ seulement
  /^ion-/
]
```

Indique à Vue d'ignorer les éléments personnalisés définis en dehors de Vue (ex. : en utilisant les API Web Components). Autrement, un message d'avertissement `Unknown custom element` sera affiché, supposant que vous avez oublié d'inscrire un composant global ou fait une faute de frappe dans son nom.

keyCodes

- **Type :** `{ [key: string]: number | Array<number> }`
- **Par défaut :** `{}`
- **Utilisation :**

```
Vue.config.keyCodes = {
  v: 86,
  f1: 112,
  // La camelCase ne marche pas
  mediaPlayPause: 179,
  // à la place vous devez utiliser la kebab-case avec des guillemets
  // doubles
  "media-play-pause": 179,
  up: [38, 87]
}
```

```
<input type="text" @keyup.media-play-pause="method">
```

Définit des alias pour les touches avec `v-on`.

performance

Nouveau dans la 2.2.0+

- **Type :** `boolean`
- **Par défaut :** `false` (à partir de la 2.2.3+)
- **Utilisation :**

Assignez ceci à `true` pour activer le suivi des performances pour l'initialisation, la compilation, le rendu et la mise à jour des composants dans la timeline des outils développeur des navigateurs. Fonctionne uniquement en mode développement et dans les navigateurs supportant l'API `performance.mark`.

productionTip

Nouveau dans la 2.2.0+

- **Type :** `boolean`
- **Par défaut :** `true`
- **Utilisation :**

Assignez ceci à `false` pour ne plus avoir la notification de production au démarrage de Vue.

API globale

`Vue.extend(options)`

- **Arguments :**
- `{Object} options`
- **Utilisation :**

Crée une « sous-classe » du constructeur de base Vue. L'argument doit être un objet contenant les options du composant.

Le cas spécial à noter ici est l'option `data` - il doit s'agir d'une fonction quand utilisé avec `Vue.extend()`.

```
<div id="point-de-montage"></div>
```

```
// crée un constructeur réutilisable
var Profil = Vue.extend({
  template: '<p>{{prenom}} {{nom}} alias {{alias}}</p>',
  data: function () {
    return {
      prenom: 'Walter',
      nom: 'White',
      alias: 'Heisenberg'
    }
  }
})
// crée une instance de Profil et la monte sur un élément
new Profil().$mount('#point-de-montage')
```

Cela donnera comme résultat :

```
<p>Walter White aka Heisenberg</p>
```

- **Voir aussi :** [Composants](#)

`Vue.nextTick([callback, contexte])`

- **Arguments :**

- `{Function} [callback]`
- `{Object} [contexte]`

- **Utilisation :**

Reporte l'exécution de la fonction de rappel au prochain cycle de mise à jour du DOM. Utilisez-le immédiatement après avoir changé des données afin d'attendre la mise à jour du DOM.

```
// modification de données
vm.msg = 'Salut'
// le DOM n'a pas encore été mis à jour
Vue.nextTick(function () {
  // le DOM est à jour
})

// utilisation en tant que promesse (2.1.0+, voir la note ci-dessous)
Vue.nextTick()
  .then(function () {
    // le DOM est à jour
  })
```

Nouveauté de la 2.1.0+ : retourne une promesse si aucune fonction de rappel n'est fournie et si les promesses sont supportées par l'environnement d'exécution. Notez que Vue ne fournit pas de

polyfill aux promesses. Aussi, si vous ciblez des navigateurs qui ne supportent pas les promesses nativement (on parle de toi, IE), vous pouvez fournir un polyfill vous-même.

- **Voir aussi :** [File d'attente de mise à jour asynchrone](#)

`Vue.set(cible, nomDePropriete/index, valeur)`

- **Arguments :**

- `{Object | Array} cible`
- `{string | number} nomDePropriete/index`
- `{any} valeur`

- **Retourne:** la valeur assignée.

- **Utilisation :**

Assigne une propriété à un objet réactif, s'assurant que la nouvelle propriété soit également réactive de manière à déclencher une nouvelle mise à jour de la vue. Ceci doit être utilisé pour les nouvelles propriétés d'objets réactifs car Vue ne peut pas détecter normalement les ajouts de propriétés (par ex.

`this.myObject.newProperty = 'bonjour');`

L'objet ne peut pas être une instance de Vue, ou l'objet de données à la racine d'une instance de Vue.

- **Voir aussi :** [Réactivité en détail](#)

`Vue.delete(cible, nomDePropriete/index)`

- **Arguments :**

- `{Object | Array} cible`
- `{string | number} nomDePropriete/index`

Seulement dans la 2.2.0+ : fonctionne aussi avec Array + index.

- **Utilisation :**

Supprime une propriété d'un objet cible. Si l'objet est réactif, cette méthode s'assure que la suppression déclenche les mises à jour de la vue. Ceci est principalement utilisé pour passer outre la limitation de Vue qui est de ne pas pouvoir détecter automatiquement la suppression de propriétés, mais vous devriez rarement en avoir besoin.

L'objet cible ne peut pas être une instance de Vue, ou l'objet de données à la racine d'une instance de Vue.

- **Voir aussi :** [Réactivité en détail](#)

`Vue.directive(id, [définition])`

- **Arguments :**

- `{string} id`
- `{Function | Object} [définition]`

- **Utilisation :**

Inscrit ou récupère une directive globale.

```
// inscrit une directive
Vue.directive('ma-directive', {
  bind: function () {},
  inserted: function () {},
  update: function () {},
  componentUpdated: function () {},
  unbind: function () {}
})

// inscription (directive comme simple fonction)
Vue.directive('ma-directive', function () {
  // cette fonction sera appelée comme `bind` et `update` ci-dessus
})

// accesseur, retourne la définition de la directive si inscrite
var maDirective = Vue.directive('ma-directive')
```

- **Voir aussi :** [Directives personnalisées](#)

Vue.filter(id, [définition])

- **Arguments :**

- {string} id
- {Function} [définition]

- **Utilisation :**

Inscrit ou récupère un filtre global.

```
// inscrit un filtre
Vue.filter('mon-filtre', function (value) {
  // retourne la valeur modifiée
})

// accesseur, retourne le filtre si inscrit
var monFiltre = Vue.filter('mon-filtre')
```

- **Voir aussi :** [Filtres](#)

Vue.component(id, [définition])

- **Arguments :**

- {string} id
- {Function | Object} [définition]

- **Utilisation :**

Inscrit ou récupère un composant global. L'inscription assigne aussi automatiquement la propriété **name** du composant au paramètre **id** donné.

```
// inscrit un constructeur étendu
Vue.component('mon-composant', Vue.extend({ /* ... */ }))

// inscrit un composant avec un objet options (appelle automatiquement
// Vue.extend)
Vue.component('mon-composant', { /* ... */ })

// récupère un composant inscrit (retourne toujours le constructeur)
var MonComposant = Vue.component('mon-composant')
```

- **Voir aussi :** [Composants](#)

`Vue.use(plugin)`

- **Arguments :**

- `{Object | Function} plugin`

- **Utilisation :**

Cette méthode doit être appelée avant d'appeler `new Vue()`

Installe un plugin Vue.js. Si l'argument `plugin` est de type **Object**, il doit exposer une méthode **install**. S'il s'agit d'une fonction, elle sera utilisée comme méthode d'installation. Cette méthode d'installation sera appelée avec `Vue` en tant qu'argument.

Quand cette méthode est appelée avec le même plugin plusieurs fois, le plugin ne sera installé qu'une seule fois.

- **Voir aussi :** [Plugins](#)

`Vue.mixin(mixin)`

- **Arguments :**

- `{Object} mixin`

- **Utilisation :**

Applique une mixin globale, qui affecte toutes les instances de `Vue` créées par la suite. Cela peut être utilisé par les créateurs de plugins pour injecter un composant personnalisé dans les composants. **Non recommandé dans le code applicatif.**

- **Voir aussi :** [Mixin global](#)

`Vue.compile(template)`

- **Arguments :**

- `{string} template`

- **Utilisation :**

Compile une string template en une fonction de rendu. **Disponible uniquement sur la version complète.**

```
var res = Vue.compile('<div><span>{{ msg }}</span></div>')

new Vue({
  data: {
    msg: 'salut'
  },
  render: res.render,
  staticRenderFns: res.staticRenderFns
})
```

- **Voir aussi :** [Fonctions de rendu](#)

`Vue.observable(object)`

Nouveau en 2.6.0+

- **Arguments :**

- `{Object} object`

- **Utilisation :**

Rend un objet réactif. En interne, Vue l'utilise sur l'objet retourné par la fonction `data`.

L'objet retourné peut être directement utilisé à l'intérieur des [fonctions de rendu](#) ainsi que des [propriétés calculées](#). Il déclenchera les mises à jour appropriées lors de mutations. Il peut également être utilisé comme un store minimal intercomposant dans des scénarios basiques :

```
const state = Vue.observable({ count: 0 })

const Demo = {
  render(h) {
    return h('button', {
      on: { click: () => { state.count++ } }
    }, `le compteur est : ${state.count}`)
  }
}
```

Dans Vue 2.x, `Vue.observable` mute directement l'objet qui lui est transmis, de sorte qu'il soit équivalent à l'objet renvoyé, comme [démontré ici](../guide/instance.html#Donnees-et-methodes). Dans Vue 3.x, un proxy réactif sera retourné à la place, laissant l'objet original non réactif s'il est muté directement.

Cependant, pour de futures compatibilités, nous recommandons toujours de travailler avec l'objet retourné par `Vue.observable`, au lieu de l'objet originel.

- **Voir aussi :** Réactivité en détail

Vue.version

- **Détails :** donne la version de Vue installée sous forme de **String**. C'est particulièrement utile pour les plugins et les composants de la communauté, où vous pouvez être amenés à utiliser différentes stratégies pour différentes versions.

- **Utilisation :**

```
var version = Number(Vue.version.split('.').[0])

if (version === 2) {
  // Vue v2.x.x
} else if (version === 1) {
  // Vue v1.x.x
} else {
  // Versions non supportées de Vue
}
```

Options / Data

data

- **Type :** **Object | Function**
- **Restriction :** accepte uniquement une fonction lorsqu'utilisé dans une définition de composant.
- **Détails :**

L'objet de données pour l'instance de Vue. Vue va de manière récursive convertir ses propriétés en des accesseurs / mutateurs (**getter/setters**) afin de les rendre « réactives ». **L'objet doit être un simple objet de base**: les objets natifs tels que les API du navigateur et les propriétés issues du prototype sont ignorées. Une règle d'or est que la donnée doit juste être de la donnée. Il n'est pas recommandé d'observer des objets ayant leur propre comportement avec états.

Une fois observé, vous ne pouvez plus ajouter de propriétés réactives à l'objet de données racine. C'est pourquoi il est recommandé de déclarer dès le départ toutes les propriétés réactives à la racine de l'objet de données, avant de créer l'instance.

Après que l'instance ait été créée, l'objet de données initial peut être accédé via **vm.\$data**. L'instance de Vue servira également de proxy pour toutes les propriétés trouvées dans l'objet de données, donc **vm.a** sera l'équivalent de **vm.\$data.a**.

Les propriétés commençant par **_** ou **\$** ne seront **pas** proxyfiées par l'instance de Vue car elles pourraient entrer en conflit avec certaines propriétés internes et méthodes d'API de Vue. Vous devrez y accéder via **vm.\$data._propriete**.

Lors de la définition d'un **composant**, la propriété **data** doit être déclarée en tant que fonction renvoyant l'objet de données initial, car il y aura plusieurs instances créées utilisant la même définition. Si nous utilisons un objet classique pour **data**, le même objet sera **partagé par référence** à toutes les instances créées ! En fournissant une fonction **data**, chaque fois qu'une nouvelle instance est créée, nous l'appelons simplement afin de récupérer une copie fraîche des données initiales.

Si nécessaire, un clone profond de l'objet original peut être obtenu en passant **vm.\$data** à travers **JSON.parse(JSON.stringify(...))**.

- **Exemple :**

```
var data = { a: 1 }

// création directe d'instances
var vm = new Vue({
  data: data
})
vm.a // => 1
vm.$data === data // => true

// data doit être une fonction lorsqu'utilisée dans Vue.extend()
var Composant = Vue.extend({
  data: function () {
    return { a: 1 }
  }
})
```

Notez que si vous utilisez la fonctions fléchées avec la propriété **data**, **this** ne sera plus l'instance du composant, mais vous pouvez toujours la récupérer en tant que premier argument de la fonction :

```
data: vm => ({ a: vm.myProp })
```

- **Voir aussi :** [Réactivité en détail](#)

props

- **Type :** `Array<string> | Object`

- **Détails :**

Une liste ou un objet décrivant les attributs exposés par le composant afin de passer des données depuis le composant parent. Ce paramètre a une syntaxe simple basée sur un tableau (**Array**) et une syntaxe alternative basée sur un objet (**Object**) qui permet une configuration avancée telle qu'une vérification de typage, des contrôles de validation personnalisés et des valeurs par défaut.

Avec la syntaxe objet, vous pouvez passer les options suivantes :

- Le **type** peut être l'un des constructeurs natifs suivants : `String, Number, Boolean, Array, Object, Date, Function, Symbol` ainsi que n'importe quelle fonction de construction ou

tableau de ces types. Les types des props seront vérifiés. Un avertissement sera fourni si une prop n'est pas du bon type. [Plus d'information](#) sur les types des props.

- **default: any** Spécifie la valeur par défaut de la prop. Si la prop n'est pas passée, cette valeur sera utilisée à la place. Les objets ou tableaux par défaut doivent être retournés depuis une fonction de fabrique.
- **required: Boolean** Définie si la prop est requise. Dans un environnement autre que de production, un avertissement sera affiché dans la console si cette valeur est évaluée à vrai et que la prop n'est pas passée.
- **validator: Function** Une fonction de validation personnalisée qui prend la valeur de prop comme seul argument. Dans un environnement autre que de production, un avertissement sera affiché dans la console si cette fonction retourne une valeur évaluée à faux. (c.-à-d. que la validation a échoué). Vous pouvez en lire plus à propos de la validation des props [ici](#).

- **Exemple :**

```
// syntaxe simple
Vue.component('props-démo-simple', {
  props: ['taille', 'monMessage']
})

// syntaxe avancée avec validation
Vue.component('props-démo-avancée', {
  props: {
    // juste une vérification de type
    hauteur: Number,
    // vérification du type ainsi que d'autres validations
    âge: {
      type : Number,
      default: 0,
      required: true,
      validator: function (valeur) {
        return valeur >= 0
      }
    }
  }
})
```

- **Voir aussi :** [Props](#)

propsData

- **Type :** { [key: string]: any }
- **Restriction :** utilisé uniquement si l'instance est créée via [new](#).
- **Détails :**

Passe des valeurs d'attribut à l'instance durant sa création. Cette propriété a pour but principal de faciliter les tests unitaires.

- **Exemple :**

```

var Comp = Vue.extend({
  props: ['msg'],
  template: '<div>{{ msg }}</div>'
})

var vm = new Comp({
  propsData: {
    msg: 'salut'
  }
})

```

computed

- **Type :** { [key: string]: Function | { get: Function, set: Function } }
- **Détails :**

Les propriétés calculées qui seront ajoutées à l'instance de Vue. Tous les accesseurs (« getters ») et mutateurs (« setters ») ont leur contexte `this` automatiquement lié à l'instance de Vue.

Notez que si vous utilisez la fonctions fléchées avec la propriété `data`, `this` ne sera plus l'instance du composant, mais vous pouvez toujours la récupérer en tant que premier argument de la fonction :

```

computed: {
  aDouble: vm => vm.a * 2
}

```

Les propriétés calculées sont mises en cache, et réévaluées uniquement lorsque leurs dépendances réactives changent. Notez que si une certaine dépendance est en dehors de la portée de l'instance (et donc non réactive), la propriété calculée ne sera **pas** mise à jour.

- **Exemple :**

```

var vm = new Vue({
  data: { a: 1 },
  computed: {
    // accesseur uniquement
    aDouble: function () {
      return this.a * 2
    },
    // accesseur et mutateur à la fois
    aPlus: {
      get: function () {
        return this.a + 1
      },
      set: function (v) {
        this.a = v - 1
      }
    }
})

```

```

        }
    })
vm.aPlus // => 2
vm.aPlus = 3
vm.a // => 2
vm.aDouble // => 4

```

- **Voir aussi :** [Propriétés calculées](#)

methods

- **Type :** { [key: string]: Function }
- **Détails :**

Les méthodes qui seront ajoutées à l'instance de Vue. Vous pouvez accéder à ces méthodes directement depuis l'instance VM ou les utiliser à travers des expressions de directives. Toutes les méthodes ont leur contexte d'appel **this** automatiquement assigné à l'instance de Vue.

Notez que __vous ne devriez pas utiliser de fonctions fléchées pour définir une méthode__ (exemple: `plus: () => this.a++`). La raison est que les fonctions fléchées sont liées au contexte parent, donc `this` ne correspondra pas à l'instance de Vue et `this.a` vaudra `undefined`.

- **Exemple :**

```

var vm = new Vue({
  data: { a: 1 },
  methods: {
    plus: function () {
      this.a++
    }
  }
})
vm.plus()
vm.a // 2

```

- **Voir aussi :** [Gestion des évènements](#)

watch

- **Type :** { [key: string]: string | Function | Object | Array }
- **Détails :**

Un objet où les clés sont des expressions à surveiller et où la valeur associée est la fonction de rappel exécutée quand cette expression change. On parle alors d'observateur ou **watcher** pour décrire ce lien. La valeur peut également être une **String** correspondant au nom d'une méthode de l'instance, ou un objet avec des options avancées. L'instance de Vue appellera **\$watch()** pour chaque clé de l'objet à l'initialisation.

- **Exemple :**

```

var vm = new Vue({
  data: {
    a: 1,
    b: 2,
    c: 3,
    d: 4,
    e: {
      f: {
        g: 5
      }
    }
  },
  watch: {
    a: function (valeur, ancienneValeur) {
      console.log('nouveau: %s, ancien: %s', valeur, ancienneValeur)
    },
    // nom d'une méthode
    b: 'uneMéthode',
    // la fonction de rappel sera appelée quelque soit les changements
    // des propriétés de l'objet observé indépendamment de leur profondeur
    c: {
      handler: function (valeur, ancienneValeur) { /* ... */ },
      deep: true
    },
    // la fonction de rappel va être appelée immédiatement après le
    // début de l'observation
    d: {
      handler: 'uneMéthode',
      immediate: true
    },
    e: [
      'handle1',
      function handle2 (valeur, ancienneValeur) { /* ... */ },
      {
        handler: function handle3 (valeur, ancienneValeur) { /* ... */ }
      },
      /* ... */
    ],
    // observe la valeur de `vm.e.f` : `{g: 5}`
    'e.f': function (valeur, ancienneValeur) { /* ... */ }
  }
})
vm.a = 2 // => nouveau : 2, ancien : 1

```

Notez que __vous ne devriez pas utiliser de fonctions fléchées pour définir un observateur__ (exemple: `saisie: nouvelleValeur => this.actualiserSuggestions(nouvelleValeur)`). La raison est que les fonctions fléchées sont liées au contexte parent, donc `this` ne correspondra pas à l'instance de Vue et `this.actualiserSuggestions` vaudra `undefined`.

- **Voir aussi :** Méthodes et données d'instance - vm.\$watch

Options / DOM

el

- **Type :** `string | Element`
- **Restriction :** uniquement respecté quand l'instance est créée via `new`.
- **Détails :**

Fournit à l'instance de Vue un élément existant du DOM sur lequel se monter. Cela peut être une `String` représentant un sélecteur CSS ou une référence à un `HTMLElement`.

Une fois l'instance montée, l'élément correspondant sera accessible via `vm.$el`.

Si cette option est disponible à l'instanciation, l'instance sera immédiatement compilée; sinon, l'utilisateur devra explicitement appeler `vm.$mount()` pour démarrer manuellement la compilation.

L'élément fourni sert seulement de point de montage. Contrairement à Vue 1.x, l'élément monté sera remplacé par le DOM généré par Vue dans tous les cas. C'est pourquoi il n'est pas recommandé de monter l'instance racine sur `` ou ``.

Si ni la fonction `render` ni l'option `template` ne sont présentes, le code HTML de l'élément du DOM sur lequel le composant est monté sera extrait et défini comme template de ce composant. Dans ce cas, la version "Runtime + Compilateur" de Vue doit être utilisée.

- **Voir aussi :**

- [Diagramme du cycle de vie](#)
- [Runtime + Compiler vs. Runtime seul](#)

template

- **Type :** `string`
- **Détails :**

Un template sous forme de chaîne de caractères qui sera utilisé comme balisage HTML pour l'instance de Vue. Le template viendra **remplacer** l'élément monté. Tout code HTML existant à l'intérieur de l'élément monté sera ignoré, à moins que des emplacements de distribution de contenu (slots) soient présents dans le template.

Si la chaîne de caractères commence par `#`, elle sera évaluée comme `querySelector` et le `innerHTML` de l'élément sélectionné sera utilisé comme template. Cela permet d'utiliser l'astuce du `<script type="x-template">` pour inclure des templates.

D'un point de vue sécurité, vous devriez uniquement utiliser des templates Vue auxquels vous pouvez faire confiance. N'utilisez jamais du contenu généré côté utilisateur comme template.

Si la fonction `render` est présente comme option de l'instance de Vue, le template sera ignoré.

- **Voir aussi :**

- [Diagramme du cycle de vie](#)

- Distribution de contenu avec des slots

render

- **Type :** (`createElement: () => VNode`) => `VNode`
- **Détails :**

Une alternative aux templates en chaîne de caractères vous permettant d'exploiter toute la puissance programmatique de JavaScript. La fonction de rendu `render` reçoit une méthode `createElement` comme premier argument servant à créer des `VNodes`.

Si le composant est un composant fonctionnel, la fonction `render` recevra aussi un argument supplémentaire `context`, qui donne accès aux données contextuelles puisque les composants fonctionnels sont sans instance.

La fonction `render` a la priorité par rapport à la fonction de rendu compilée à partir de l'option `template`, ou par rapport au template HTML de l'élément d'ancre dans le DOM qui est spécifié par l'option `el`.

- **Voir aussi :** [Fonctions de rendu](#)

renderError

Nouveau dans la 2.2.0+

- **Type :** (`createElement: () => VNode, error: Error`) => `VNode`
- **Détails :**

Fonctionne uniquement en mode développement.

Fournit un rendu alternatif en sortie quand la fonction `render` par défaut rencontre une erreur. L'erreur sera passée à `renderError` comme second argument. C'est particulièrement appréciable quand utilisé conjointement avec du chargement à chaud (hot-reload).

- **Exemple :**

```
new Vue({
  render (h) {
    throw new Error('oups')
  },
  renderError (h, err) {
    return h('pre', { style: { color: 'red' } }, err.stack)
  }
}).$mount('#app')
```

- **Voir aussi :** [Fonctions de rendu](#)

Options / Cycle de vie des hooks

Tous les hooks du cycle de vie ont automatiquement leur contexte `this` rattaché à l'instance, afin que vous puissiez accéder aux données, propriétés calculées et méthodes. Cela signifie que __vous ne devriez pas utiliser

une fonction fléchée pour définir une méthode du cycle de vie__ (p. ex. `created: () => this.fetchTodos()`). La raison est que les fonctions fléchées utilisent le contexte parent, donc `this` ne sera pas l'instance de Vue comme vous pouvez vous y attendre et `this.fetchTodos` sera `undefined`.

beforeCreate

- **Type :** `Function`

- **Détails :**

Appelé de manière synchrone juste après que l'instance ait été initialisée, et avant l'observation des données et l'installation des évènements / observateurs.

- **Voir aussi :** [Diagramme du cycle de vie](#)

created

- **Type :** `Function`

- **Détails :**

Appelé de manière synchrone après que l'instance ait été créée. À ce stade, l'instance a fini de traiter les options, ce qui signifie que les éléments suivants ont été installés: observation des données, propriétés calculées, méthodes, fonctions de retour des observateurs et évènements. Cependant, la propriété `$el` n'est pas encore disponible.

- **Voir aussi :** [Diagramme du cycle de vie](#)

beforeMount

- **Type :** `Function`

- **Détails :**

Appelé juste avant que le montage commence: la fonction `render` est sur le point d'être appelée pour la première fois.

Ce hook n'est pas appelé durant le rendu côté serveur.

- **Voir aussi :** [Diagramme du cycle de vie](#)

mounted

- **Type :** `Function`

- **Détails :**

Appelé juste après que l'instance ait été montée, là où `el` est remplacé par le nouvellement créé `vm.$el`. Si l'instance à la racine est montée sur un élément du document, alors `vm.$el` sera aussi dans le document quand `mounted` est appelé.

Notez que `mounted` ne garantit pas que tous les composants aient été montés. Si vous souhaitez attendre jusqu'à ce que le rendu de la vue entière ait été fait, vous pouvez utiliser `vm.$nextTick` à l'intérieur de `mounted` :

```
mounted: function () {
  this.$nextTick(function () {
    // Ce code va être exécuté seulement
    // une fois le rendu de la vue entière terminé
  })
}
```

Ce hook n'est pas appelé durant le rendu côté serveur.

- **Voir aussi :** [Diagramme du cycle de vie](#)

beforeUpdate

- **Type :** [Function](#)
- **Détails :**

Appelé quand les données changent, avant le patch du DOM virtuel. C'est le bon endroit pour accéder au DOM existant avant la mise à jour, par ex. pour retirer manuellement des écouteurs d'évènement.

Ce hook n'est pas appelé durant le rendu côté serveur car seul le rendu initial est généré côté serveur.

- **Voir aussi :** [Diagramme du cycle de vie](#)

updated

- **Type :** [Function](#)
- **Détails :**

Appelé après qu'un changement d'une donnée ait causé un nouveau rendu et patch du DOM virtuel.

Le DOM du composant aura été mis à jour quand ce hook est appelé, donc vous pouvez effectuer des opérations dépendantes du DOM ici. Cependant, dans la plupart des cas, vous devriez éviter de changer l'état dans ce hook. Pour réagir à des changements d'état, il vaut généralement mieux utiliser une [propriété calculée](#) ou un [observateur](#) à la place.

Notez que [updated](#) ne garantit **pas** que tous les composants aient été montés. Si vous souhaitez attendre jusqu'à ce que le rendu de la vue entière ait été fait, vous pouvez utiliser [vm.\\$nextTick](#) à l'intérieur de [updated](#) :

```
updated: function () {
  this.$nextTick(function () {
    // Ce code va être exécuté seulement
    // une fois le rendu de la vue entière terminé
  })
}
```

Ce hook n'est pas appelé durant le rendu côté serveur.

- **Voir aussi :** [Diagramme du cycle de vie](#)

activated

- **Type :** [Function](#)

- **Détails :**

Appelé quand un composant gardé en vie ([keep-alive](#)) est activé.

Ce hook n'est pas appelé durant le rendu côté serveur.

- **Voir aussi :**

- [Composants intégrés par défaut - keep-alive](#)
- [Composants dynamiques - keep-alive](#)

deactivated

- **Type :** [Function](#)

- **Détails :**

Appelé quand un composant gardé en vie ([keep-alive](#)) est désactivé.

Ce hook n'est pas appelé durant le rendu côté serveur.

- **Voir aussi :**

- [Composants intégrés par défaut - keep-alive](#)
- [Composants dynamiques - keep-alive](#)

beforeDestroy

- **Type :** [Function](#)

- **Détails :**

Appelé juste avant qu'une instance de Vue ne soit détruite. À ce stade, l'instance est toujours pleinement fonctionnelle.

Ce hook n'est pas appelé durant le rendu côté serveur.

- **Voir aussi :** [Diagramme du cycle de vie](#)

destroyed

- **Type :** [Function](#)

- **Détails :**

Appelé après qu'une instance de Vue ait été détruite. Quand ce hook est appelé, toutes les directives de l'instance de Vue ont été détachées, tous les écouteurs d'évènements ont été supprimés et toutes les instances de Vue enfants ont également été détruites.

Ce hook n'est pas appelé durant le rendu côté serveur.

- **Voir aussi :** [Diagramme du cycle de vie](#)

errorCaptured

Nouveau dans la 2.5.0+

- **Type :** `(err: Error, vm: Component, info: string) => ?boolean`

- **Détails :**

Appelé quand une erreur provenant de n'importe quel composant enfant est capturée. Le hook reçoit trois arguments : l'erreur, l'instance du composant qui a déclenché l'erreur et une chaîne de caractères contenant des informations sur l'endroit où l'erreur a été capturée. Le hook peut retourner `false` pour arrêter la propagation de l'erreur.

Vous pouvez modifier l'état du composant dans ce hook. Il est cependant important d'avoir une condition dans votre template ou fonction de rendu qui court-circuite les autres composants quand une erreur est capturée. Autrement le composant va partir dans une boucle de rendu infinie.

Règles de propagation d'erreur

- Par défaut, toutes les erreurs sont toujours envoyées à l'objet global `config.errorHandler` s'il est défini. Donc ces erreurs peuvent toujours être reportées à un service d'analyse centralisé.
- Si des hooks `errorCaptured` multiples existent sur une chaîne de composant hérité ou sur une chaîne parente, toutes seront invoquées avec la même erreur.
- Si le hook `errorCaptured` renvoie lui-même une erreur, l'erreur originale capturée et l'erreur courante seront toutes deux envoyées au `config.errorHandler` global.
- Un hook `errorCaptured` peut retourner `false` pour empêcher la propagation de l'erreur vers le haut. Il est essentiel d'indiquer que « cette erreur a été traitée et doit être ignorée ». Cela empêchera n'importe quel hook `errorCaptured` supplémentaire ou le `config.errorHandler` d'être invoqué par cette erreur.

Options / Ressources

directives

- **Type :** `Object`

- **Détails :**

Un objet de mappage des directives à mettre à disposition de l'instance de Vue.

- **Voir aussi :** [Directives personnalisées](#)

filters

- **Type :** `Object`

- **Détails :**

Un objet de mappage des filtres à mettre à disposition de l'instance de Vue.

- **Voir aussi :** [Vue.filter](#)

components

- **Type :** [Object](#)

- **Détails :**

Un objet de mappage des composants à mettre à disposition de l'instance de Vue.

- **Voir aussi :** [Composants](#)

Options / Divers

parent

- **Type :** [Instance de Vue](#)

- **Détails :**

Spécifie l'instance parente pour l'instance qui va être créée. Établit une relation parent-enfant entre les deux. Le parent sera accessible via [this.\\$parent](#) pour l'enfant, et l'enfant sera ajouté à la liste [\\$children](#) du parent.

Utilisez `\\$parent` et `\\$children` avec parcimonie - ils servent surtout comme écoutille de secours. Préférez l'utilisation de propriétés et d'évènements pour la communication parent-enfant.

mixins

- **Type :** [Array<Object>](#)

- **Détails :**

L'option [mixins](#) accepte une liste d'objets mixin. Ces objets mixin peuvent contenir des options d'instance tout comme des objets d'instance normaux, et elles seront fusionnées avec les éventuelles options existantes en utilisant la même stratégie de fusion que dans [Vue.extend\(\)](#). Par exemple, si votre mixin contient un hook [created](#) et que le composant lui-même en a également un, les deux fonctions seront appelées.

Les hooks de *.mixin* sont appelés dans l'ordre dans lequel ils sont fournis, et appelés avant les propres hooks du composant.

- **Exemple :**

```
var mixin = {
  created: function () { console.log(1) }
}
var vm = new Vue({
  created: function () { console.log(2) },
  mixins: [mixin]
})
```

```
// => 1  
// => 2
```

- **Voir aussi :** [Mixins](#)

extends

- **Type :** `Object | Function`

- **Détails :**

Permet d'étendre déclarativement un autre composant (qui peut être un simple objet d'options ou un constructeur) sans avoir à utiliser `Vue.extend`. C'est destiné en premier lieu à rendre plus facile les extensions entre composants monofichiers.

Cette option est similaire à `mixins`.

- **Exemple :**

```
var CompA = { ... }  
  
// étend `CompA` sans avoir à appeler `Vue.extend` sur l'un ou l'autre  
var CompB = {  
    extends: CompA,  
    ...  
}
```

provide / inject

Nouveau dans la 2.2.0+

- **Type :**

- `provide : Object | () => Object`
- `inject : Array<string> | { [key: string]: string | Symbol | Object }`

- **Détails :**

`provide` et `inject` sont fournis principalement pour des cas d'utilisation avancés dans les bibliothèques de plugins / composants. Il n'est PAS recommandé de les utiliser dans du code applicatif générique.

Ces deux options sont utilisées ensemble pour permettre à un composant parent de servir d'injecteur de dépendances pour tous ses descendants, peu importe la profondeur de la hiérarchie de composants, tant qu'ils sont dans la même chaîne parente. Si vous êtes familiers avec React, c'est très similaire à la fonctionnalité de contexte dans React.

L'option `provide` doit être un objet ou une fonction renvoyant un objet. Cet objet contient les propriétés qui sont disponibles pour l'injection dans ses descendants. Vous pouvez utiliser des `Symbol` ES2015 comme clés dans cet objet, mais seulement dans les environnements supportant nativement `Symbol` et `Reflect.ownKeys`.

L'option **inject** doit être soit :

- un **Array** de **String**, ou
- un objet où les clés sont les noms des liaisons locales et où les valeurs sont :
 - les clés (**String** ou **Symbol**) à rechercher dans les injections disponibles, ou
 - un objet où :
 - la propriété **from** est la clé (**String** ou **Symbol**) à rechercher dans les injections disponibles, et
 - la propriété **default** est utilisé comme valeur de substitution.

Note : les liaisons **provide** et **inject** ne sont PAS réactives. C'est intentionnel. Cependant, si vous passez un objet observé, les propriétés sur cet objet resteront réactives.

- **Exemple :**

```
// parent component providing 'foo'  
var Provider = {  
  provide: {  
    foo: 'bar'  
  },  
  // ...  
}  
  
// le composant enfant injectant 'foo'  
var Enfant = {  
  inject: ['foo'],  
  created () {  
    console.log(this.foo) // => "bar"  
  }  
  // ...  
}
```

Avec les **Symbol** ES2015, la fonction **provide** et l'objet **inject** :

```
const s = Symbol()  
  
const Provider = {  
  provide () {  
    return {  
      [s]: 'foo'  
    }  
  }  
}  
  
const Enfant = {  
  inject: { s },  
  // ...  
}
```

Les deux prochains exemples fonctionnent seulement avec Vue 2.2.1+. En dessous de cette version, les valeurs injectées étaient résolues après l'initialisation des **props** et de **data**.

En utilisant une valeur injectée comme valeur par défaut pour une prop :

```
const Enfant = {
  inject: ['foo'],
  props: {
    bar: {
      default () {
        return this.foo
      }
    }
  }
}
```

En utilisant une valeur injectée comme entrée de données :

```
const Enfant = {
  inject: ['foo'],
  data () {
    return {
      bar: this.foo
    }
  }
}
```

Dans la 2.5.0+ les injections peuvent être optionnelles avec une valeur par défaut :

```
const Child = {
  inject: {
    foo: { default: 'foo' }
  }
}
```

S'il est nécessaire de pouvoir injecter via une propriété avec un nom différent, utilisez **from** pour indiquer la source de la propriété :

```
const Child = {
  inject: {
    foo: {
      from: 'bar',
      default: 'foo'
    }
  }
}
```

De façon similaire aux valeurs par défaut des props, vous devez utiliser une fabrique de fonctions pour les valeurs non primitives :

```
const Child = {
  inject: {
    foo: {
      from: 'bar',
      default: () => [1, 2, 3]
    }
  }
}
```

Options / Divers

name

- **Type :** `string`
- **Restriction :** respecté uniquement lorsqu'utilisé comme option du composant.
- **Détails :**

Permet au composant de s'invoquer lui-même récursivement dans son template. Notez que lorsqu'un composant est déclaré globalement avec `Vue.component()`, l'identifiant global est automatiquement assigné à sa propriété `name`.

Un autre bénéfice du fait de spécifier une option `name` est le débogage. Les composants nommés donnent des messages d'avertissement plus utiles. De plus, lorsque vous inspectez une application via les `vue-devtools`, les composants non nommés s'afficheront en tant que `<AnonymousComponent>`, ce qui n'est pas très instructif. En fournissant une option `name`, vous obtiendrez bien plus d'informations dans l'arbre de composants.

delimiters

- **Type :** `Array<string>`
- **Par défaut :** `["{{", "}}"]`
- **Restrictions :** Cette option n'est disponible que dans la version complète du build, avec la compilation dans le navigateur.
- **Détails :**

Change les délimiteurs d'interpolation de texte. **Cette option est uniquement disponible en version complète.**

- **Exemple :**

```
new Vue({
  delimiters: ['${{', '}}']
})
```

```
// Les délimiteurs ont été changés pour suivre le style des templates  
ES6
```

functional

- **Type :** Boolean

- **Détails :**

Rend le composant sans état (pas de propriété `data`) et sans instance (pas de contexte `this`). Il s'agit simplement d'une fonction `render` qui retourne des noeuds virtuels, ce qui réduit fortement les couts en performance au rendu pour ce type de composants.

- **Voir aussi :** Composants fonctionnels

model

Nouveau dans la 2.2.0

- **Type :** { prop?: string, event?: string }

- **Détails :**

Permet à un composant personnalisé de définir la prop et l'évènement utilisé quand il est utilisé avec `v-model`. Par défaut, `v-model` sur un composant utilise `value` comme prop et `input` comme évènement, mais certains types de champs de saisie comme les cases à cocher et les boutons radio peuvent vouloir utiliser la prop `value` à d'autres fins. Utiliser l'option `model` peut éviter le conflit dans ce genre de cas.

- **Exemple :**

```
Vue.component('ma-checkbox', {  
  model: {  
    prop: 'checked',  
    event: 'change'  
  },  
  props: {  
    // cela permet d'utiliser la prop `value` à d'autres fins  
    value: String,  
    // utilise `checked` comme prop qui prend la place de `value`  
    checked: {  
      type: Number,  
      default: 0  
    }  
  },  
  // ...  
})
```

```
<ma-checkbox v-model="foo" value="une certaine valeur"></ma-checkbox>
```

Le code ci-dessus est équivalent à :

```
<ma-checkbox  
  :checked="foo"  
  @change="val => { foo = val }"  
  value="une certaine valeur">  
</ma-checkbox>
```

inheritAttrs

Nouveau dans la 2.4.0+

- **Type :** boolean
- **Par défaut :** true
- **Détails :**

Par défaut, les attributs de portée parente qui ne sont pas reconnus en tant que props vont « échouer » et être appliqués à l'élément racine du composant enfant en tant qu'attribut HTML normal. Quand on crée un composant qui encapsule un élément cible ou un autre composant, cela peut ne pas être le comportement souhaité. En mettant `inheritAttrs` à false, ce comportement par défaut peut être désactivé. Les attributs sont disponibles via la propriété d'instance `$attrs` (aussi nouvelle en 2.4) et peuvent être explicitement liée a un élément non-racine en utilisant `v-bind`.

Note : cette option n'affecte pas les liaisons `class` et `style`.

comments

Nouveau dans la 2.4.0+

- **Type :** boolean
- **Par défaut :** false
- **Restrictions :** Cette option est uniquement disponible dans le build complet, avec la compilation dans le navigateur.
- **Détails :**

Quand mis à true, cela va conserver et faire le rendu HTML des commentaires trouvés dans les templates. Le comportement par défaut est de les enlever.

Propriétés d'instance

vm.\$data

- **Type :** Object
- **Détails :**

L'objet **data** est ce que l'instance de Vue observe. L'instance de Vue agit comme un proxy pour l'accès aux propriétés de cet objet **data**.

- **Voir aussi :** [Options / Data - data](#)

vm.\$props

Nouveau dans la 2.2.0+

- **Type :** [Object](#)

- **Détails :**

Un objet représentant les props actuelles qu'un composant a reçues. L'instance de Vue agit comme un proxy pour accéder aux propriétés sur son objet **props**.

vm.\$el

- **Type :** [Element](#)

- **Accessible uniquement en lecture**

- **Détails :**

L'élément racine du DOM que gère l'instance de Vue.

vm.\$options

- **Type :** [Object](#)

- **Accessible uniquement en lecture**

- **Détails :**

Les options d'instanciation utilisées pour cette instance de Vue. C'est utile lorsque vous voulez inclure des propriétés personnalisées dans les options:

```
new Vue({
  optionPersonnalisée: 'foo',
  created: function () {
    console.log(this.$options.optionPersonnalisée) // => 'foo'
  }
})
```

vm.\$parent

- **Type :** [Instance de Vue](#)

- **Accessible uniquement en lecture**

- **Détails :**

L'instance parente, si l'instance actuelle en a une.

vm.\$root

- **Type :** Instance de Vue
- **Accessible uniquement en lecture**
- **Détails :**

L'instance de Vue à la racine de l'arbre de composants actuel. Si l'instance actuelle n'a pas de parents, la valeur sera l'instance elle-même.

vm.\$children

- **Type :** Array<Instance de Vue>
- **Accessible uniquement en lecture**
- **Détails :**

Les composants enfants directs de l'instance actuelle. **Notez que l'ordre des enfants n'est pas garanti pour \$children, et que cette propriété n'est pas réactive.** Si vous vous retrouvez à essayer d'utiliser \$children pour de la liaison de données, optez plutôt pour un **Array** et **v-for** pour générer les composants enfants, en utilisant l'**Array** comme unique source fiable.

vm.\$slots

- **Type :** { [name: string]: ?Array<VNode> }
- **Accessible uniquement en lecture**
- **Détails :**

Utilisé pour accéder programmatiquement à du contenu distribué par slots. Chaque slot nommé a sa propriété correspondante (p. ex. le contenu de **v-slot:foo** sera trouvé dans **vm.\$slots.foo**). La propriété **default** contient tous les noeuds non inclus dans un slot nommé ou contenus de **v-slot:default**.

Note: **v-slot:foo** est supporté dans la v2.6+. Pour des versions plus anciennes, vous pouvez utiliser la syntaxe dépréciée.

Accéder à **vm.\$slots** est plus utile lorsque vous écrivez un composant avec une fonction de rendu.

- **Exemple :**

```
<blog-post>
  <template v-slot:header>
    <h1>À propos de moi</h1>
  </template>

  <p>Voici du contenu pour la page, qui sera inclus dans
  vm.$slots.default, car il n'est pas à l'intérieur d'un slot nommé.</p>

  <template v-slot:footer>
```

```

<p>Copyright 2016 Evan You</p>
</template>

<p>Si j'ai du contenu ici, il sera aussi inclus dans
vm.$slots.default.</p>
</blog-post>

```

```

Vue.component('blog-post', {
  render: function (createElement) {
    var header = this.$slots.header
    var body   = this.$slots.default
    var footer = this.$slots.footer
    return createElement('div', [
      createElement('header', header),
      createElement('main', body),
      createElement('footer', footer)
    ])
  }
})

```

- **Voir aussi :**

- Composant [`<slot>`](#)
- Distribution de contenu avec des slots
- Fonctions de rendu - Slots

vm.\$scopedSlots

Nouveau dans la 2.1.0+

- **Type :** { [name: string]: props => Array<VNode> | undefined }
- **Accessible uniquement en lecture**
- **Détails :**

Utilisé pour accéder programmatiquement aux [slots avec portée](#). Pour chaque slot, y compris celui par défaut [default](#), l'objet contient une fonction correspondante qui retourne des nœuds virtuels [VNode](#).

Accéder à [vm.\\$scopedSlots](#) est surtout utile lors de l'écriture d'un composant avec une [fonction de rendu](#).

Note : depuis la 2.6.0+, il y a deux changements notables pour cette propriété :

1. Les fonctions de slot avec portée garantissent de retourner un tableau de VNodes, sauf si la valeur de retour est invalide, auquel cas la fonction retournera [undefined](#).
2. Tous les [\\$slots](#) sont maintenant exposés via [\\$scopedSlots](#) en tant que fonctions. Si vous travaillez avec des fonctions de rendu, il est maintenant recommandé de toujours accéder aux slots via [\\$scopedSlots](#), si elles utilisent actuellement une portée ou non. Cela ne va pas seulement

permettre d'ajouter des scopes simplement lors de futures refactorisations, mais également de faciliter votre éventuelle migration vers Vue 3, où tous les slots seront des fonctions.

- **Voir aussi :**

- [Composant <slot>](#)
- [Slots avec portée](#)
- [Fonctions de rendu - Slots](#)

vm.\$refs

- **Type :** `Object`

- **Accessible uniquement en lecture**

- **Détails :**

Un objet contenant des éléments du DOM et des composants, ayant des attributs `ref` enregistrée.

- **Voir aussi :**

- [Les refs des composants enfants](#)
- [Attributs spéciaux - ref](#)

vm.\$isServer

- **Type :** `boolean`

- **Accessible uniquement en lecture**

- **Détails :**

Vaut `true` si l'instance actuelle de Vue s'exécute côté serveur.

- **Voir aussi :** [Rendu côté serveur](#)

vm.\$attrs

Nouveau dans la 2.4.0+

- **Type :** `{ [key: string]: string }`

- **Accessible uniquement en lecture**

- **Détails :**

Contient les attributs liés de portée parente (à l'exception de `class` et `style`) qui ne sont pas reconnus (et extrait) en tant que props. Quand un composant n'a aucune props de déclarée, il contient essentiellement toutes les liaisons de portée parente (à l'exception de `class` et `style`), et peut être passé à l'intérieur d'un composant enfant via `v-bind="$attrs"`. Ceci est utile pour la création de composants d'ordre supérieur.

vm.\$listeners

Nouveau dans la 2.4.0+

- **Type :** { [key: string]: Function | Array<Function> }

- **Accessible uniquement en lecture**

- **Détails :**

Contient le gestionnaire d'évènement `v-on` de portée parente (sans le modificateur `.native`). Il peut être passé à l'intérieur d'un composant enfant via `v-on="$listeners"`. Ceci est utile pour la création de composants d'ordre supérieur.

Méthodes et données d'instance

`vm.$watch(expOrFn, callback, [options])`

- **Arguments :**

- `{string | Function}` `expOrFn`
- `{Function | Object}` `callback`
- `{Object}` `[options]`
 - `{boolean}` `deep`
 - `{boolean}` `immediate`

- **Retourne :** `{Function}` `unwatch`

- **Utilisation :**

Observe les changements sur l'instance de Vue à partir d'une expression ou d'une fonction calculée. La fonction de rappel est appelée avec la nouvelle et l'ancienne valeur. L'expression accepte uniquement les chemins simples délimités par des points. Pour des expressions plus complexes, utilisez plutôt une fonction.

Note: lors de la modification (et non la réassignation) d'un `Object` ou d'un `Array`, l'ancienne valeur sera la même que la nouvelle valeur car ils réfèrent au même `Object/Array`. Vue ne conserve pas de copie de la valeur avant modification.

- **Exemple :**

```
// keypath
vm.$watch('a.b.c', function (nouvelleValeur, ancienneValeur) {
  // fait quelque-chose
})

// fonction
vm.$watch(
  function () {
    // chaque fois que l'expression `this.a + this.b` va produire un
    // résultat différent,
    // le gestionnaire d'écoute va être appelé. C'est comme si nous
    // surveillons
    // une propriété calculée sans définir de propriété calculée en
    // elle-même.
    return this.a + this.b
},
```

```

        function (nouvelleValeur, ancienneValeur) {
            // fait quelque-chose
        }
    )

```

`vm.$watch` retourne une fonction `unwatch` qui une fois exécutée stoppe le déclenchement de la fonction `callback`:

```

var unwatch = vm.$watch('a', cb)
// plus tard, démonte l'observateur
unwatch()

```

- **Option: deep**

Pour aussi détecter les changements des valeurs imbriquées dans les objets, vous devez passer `deep: true` dans l'argument des options. Notez que vous n'avez pas besoin de cela pour observer les modifications d'un `Array`.

```

vm.$watch('monObjet', callback, {
    deep: true
})
vm.monObjet.valeurImbriquée = 123
// la fonction de rappel est déclenchée

```

- **Option: immediate**

Passer `immediate: true` dans les options déclenchera immédiatement la fonction `callback` avec la valeur actuelle de l'expression :

```

vm.$watch('a', callback, {
    immediate: true
})
// la fonction `callback` est immédiatement déclenchée avec la valeur
actuelle de `a`

```

Notez qu'avec l'option `immediate` vous serez incapable d'utiliser `unwatch` sur la propriété fournie par le premier appel de la fonction de rappel.

```

// Ceci va générer une erreur
var unwatch = vm.$watch(
    'value',
    function () {
        doSomething()
        unwatch()
    },

```

```
    { immediate: true }  
)
```

Si vous voulez toujours appeler la fonction `unwatch` à l'intérieur de la fonction de rappel, vous devrez d'abord vérifier sa disponibilité :

```
var unwatch = vm.$watch(  
  'value',  
  function () {  
    doSomething()  
    if (unwatch) {  
      unwatch()  
    }  
  },  
  { immediate: true }  
)
```

`vm.$set(cible, nomDePropriete/index, valeur)`

- **Arguments :**

- `{Object | Array} cible`
- `{string | number} nomDePropriete/index`
- `{any} valeur`

- **Retourne :** la valeur assignée

- **Utilisation :**

C'est un **alias** à la fonction globale `Vue.set`.

- **Voir aussi :** [Vue.set](#)

`vm.$delete(cible, nomDePropriete/index)`

- **Arguments :**

- `{Object | Array} cible`
- `{string | number} nomDePropriete/index`

- **Utilisation :**

C'est un **alias** à la fonction globale `Vue.delete`.

- **Voir aussi :** [Vue.delete](#)

Méthodes et Évènements d'Instance

`vm.$on(évènement, callback)`

- **Arguments :**

- {string | Array<string>} évènement (Array supportée unique depuis la 2.2.0)
- {Function} callback

- **Utilisation :**

Écoute un évènement personnalisé sur l'instance `vm`. Les évènements peuvent être déclenchés avec `vm.$emit`. La fonction de rappel recevra tous les arguments additionnels passés dans ces méthodes de déclenchement d'évènement.

- **Exemple :**

```
vm.$on('test', function (msg) {
  console.log(msg)
})
vm.$emit('test', 'salut')
// => "salut"
```

`vm.$once(évènement, callback)`

- **Arguments :**

- {string} évènement
- {Function} callback

- **Utilisation :**

Écoute un évènement personnalisé, mais qu'une seule fois. L'écouteur sera supprimé une fois déclenché pour la première fois.

`vm.$off([event, callback])`

- **Arguments :**

- {string | Array<string>} évènement (les tableaux sont seulement supportés dans la 2.2.2+)
- {Function} [callback]

- **Utilisation :**

Supprime un ou des écouteurs d'évènements.

- Si aucun argument n'est fourni, supprime tous les écouteurs d'évènements;
- Si seul l'argument événement est fourni, supprime tous les écouteurs de cet évènement;
- Si l'évènement et la fonction de rappel sont fournis, supprime l'écouteur uniquement pour cet événement et cette fonction de rappel spécifique.

`vm.$emit(eventName, [...args])`

- **Arguments :**

- {string} nom d'évènement
- [...arguments]

Déclenche un évènement sur l'instance actuelle. Tous les arguments additionnels sont passés à la fonction de rappel de l'écouteur.

- **Exemples:**

Utiliser `$emit` avec un nom d'évènement :

```
Vue.component('welcome-button', {
  template: `
    <button v-on:click="$emit('welcome')">
      Cliquez-moi pour être salué
    </button>
  `
})
```

```
<div id="emit-example-simple">
  <welcome-button v-on:welcome="sayHi"></welcome-button>
</div>
```

```
new Vue({
  el: '#emit-example-simple',
  methods: {
    sayHi: function () {
      alert('Salut !')
    }
  }
})
```

```
<div id="emit-example-simple" class="demo">
  <welcome-button v-on:welcome="sayHi"></welcome-button>
</div>
<script>
  Vue.component('welcome-button', {
    template: `
      <button v-on:click="$emit('welcome')">
        Cliquez-moi pour être salué
      </button>
    `
  })
  new Vue({
    el: '#emit-example-simple',
    methods: {
      sayHi: function () {
        alert('Salut !')
      }
    }
  })
</script>
```

```

        }
    })
</script>

```

Utiliser `$emit` avec des arguments additionnels :

```

Vue.component('magic-eight-ball', {
  data: function () {
    return {
      possibleAdvice: ['Oui', 'Non', 'Peut-être']
    }
  },
  methods: {
    giveAdvice: function () {
      var randomAdviceIndex = Math.floor(Math.random() *
this.possibleAdvice.length)
      this.$emit('give-advice',
this.possibleAdvice[randomAdviceIndex])
    }
  },
  template: `
    <button v-on:click="giveAdvice">
      Cliquez-moi pour un indice
    </button>
  `
})

```

```

<div id="emit-example-argument">
  <magic-eight-ball v-on:give-advice="showAdvice"></magic-eight-ball>
</div>

```

```

new Vue({
  el: '#emit-example-argument',
  methods: {
    showAdvice: function (advice) {
      alert(advice)
    }
  }
})

```

```

<div id="emit-example-argument" class="demo">
  <magic-eight-ball v-on:give-advice="showAdvice"></magic-eight-ball>
</div>
<script>
  Vue.component('magic-eight-ball', {

```

```

    data: function () {
      return {
        possibleAdvice: ['Oui', 'Non', 'Peut-être']
      }
    },
    methods: {
      giveAdvice: function () {
        var randomAdviceIndex = Math.floor(Math.random() *
this.possibleAdvice.length)
        this.$emit('give-advice',
this.possibleAdvice[randomAdviceIndex])
      }
    },
    template: `
      <button v-on:click="giveAdvice">
        Cliquez-moi pour un indice
      </button>
    `
  })
new Vue({
  el: '#emit-example-argument',
  methods: {
    showAdvice: function (advice) {
      alert(advice)
    }
  }
})
</script>

```

Méthodes d'Instance / Cycle de Vie

`vm.$mount([élémentOuSelecteur])`

- **Arguments :**

- `{Element | string} [élémentOuSelecteur]`
- `{boolean} [hydratation]`

- **Retourne :** `vm` - l'instance elle-même

- **Utilisation :**

Si une Instance de Vue n'a pas reçu l'option `el` à l'instanciation, il sera dans un état « non monté », sans élément du DOM associé. `vm.$mount()` peut être utilisé pour démarrer manuellement le montage d'une Instance de Vue non montée.

Si l'argument `élémentOuSelecteur` n'est pas fourni, le rendu du template se fera dans un élément hors du document, et vous devrez utiliser les API natives du DOM pour l'insérer vous-même dans le document.

La méthode retourne l'instance elle-même afin que vous puissiez chainer d'autres méthodes d'instance ensuite.

- **Exemple :**

```
var MonComposant = Vue.extend({
  template: '<div>Salut!</div>'
})

// crée et monte sur #app (remplacera #app)
new MonComposant().$mount('#app')

// le code ci-dessus est l'équivalent de :
new MonComposant({ el: '#app' })

// ou bien, fait le rendu hors du document et l'insère par la suite
var composant = new MonComposant().$mount()
document.getElementById('app').appendChild(composant.$el)
```

- **Voir aussi :**

- [Diagramme du cycle de vie](#)
- [Rendu côté serveur](#)

vm.\$forceUpdate()

- **Utilisation :**

Force l'instance de Vue à refaire son rendu. Notez que cela n'affecte pas tous les composants enfants, seulement l'instance elle-même et les composants enfants avec un contenu de slot inséré.

vm.\$nextTick([callback])

- **Arguments :**

- [{Function} \[callback\]](#)

- **Utilisation :**

Reporte l'exécution de la fonction `callback` au prochain cycle de mise à jour du DOM. Utilisez ceci immédiatement après avoir changé des données pour attendre la mise à jour du DOM. C'est la même chose que la fonction globale `Vue.nextTick`, sauf que le contexte `this` dans la fonction `callback` est automatiquement lié à l'instance appelant cette méthode.

Nouveau dans la 2.1.0+ : retourne une promesse si aucune fonction de rappel n'est fournie et si les promesses ne sont pas supportées dans l'environnement d'exécution. Notez que Vue ne fournit pas de polyfill aux promesses. Aussi, si vous ciblez des navigateurs qui ne supportent pas les promesses nativement (on parle de toi, IE), vous pouvez fournir un polyfill vous-même.

- **Exemple :**

```
new Vue({
  // ...
  methods: {
```

```

// ...
exemple: function () {
    // modifie des données
    this.message = 'changé'
    // le DOM n'est pas encore mis à jour
    this.$nextTick(function () {
        // le DOM est maintenant à jour
        // `this` est lié à l'instance courante
        this.faireAutreChose()
    })
}
}
)

```

- **Voir aussi :**

- [Vue.nextTick](#)
- [File d'attente de mise à jour asynchrone](#)

`vm.$destroy()`

- **Utilisation :**

Détruit complètement une instance de Vue `vm`. Supprime ses connexions avec d'autres instances de Vue, détache toutes ses directives, désactive tous les écouteurs d'évènements.

Déclenche les hooks `beforeDestroy` et `destroyed`.

Dans les cas d'utilisation normaux, vous ne devriez pas avoir à appeler cette méthode vous-même. Contrôlez plutôt le cycle de vie de vos composants enfants de manière pilotée par les données, en utilisant `v-if` et `v-for`.

- **Voir aussi :** [Diagramme du cycle de vie](#)

Directives

v-text

- **Attend comme valeur :** `string`

- **Détails :**

Met à jour le contenu textuel (`textContent`) de l'élément. Si vous n'avez besoin de mettre à jour qu'une partie de ce contenu, vous devriez utiliser les interpolations `{{ Mustache }}`.

- **Exemple :**

```

<span v-text="msg"></span>
<!-- même chose que -->
<span>{{msg}}</span>

```

- **Voir aussi :** [Syntaxe de liaison de données - Interpolations](#)

v-html

- **Attend comme valeur :** [string](#)

- **Détails :**

Met à jour le contenu HTML ([innerHTML](#)) de l'élément. **Notez que les contenus sont insérés en pur HTML - ils ne seront pas compilés en tant que templates Vue.** Si vous vous retrouvez à vouloir composer des templates avec [v-html](#), essayez de repenser la solution en utilisant des composants à la place.

Faire le rendu dynamique de code HTML arbitraire sur votre site web peut être très dangereux car cela peut mener facilement à des [attaques XSS](https://fr.wikipedia.org/wiki/Cross-site_scripting). Utilisez `v-html` uniquement avec du contenu de confiance et **jamais** avec du contenu fourni par les utilisateurs.

Dans les [composants monofichiers]([../guide/single-file-components.html](#)), les styles `scoped` ne seront pas appliqués aux contenus à l'intérieur de `v-html`, car ce HTML n'est pas géré par le compilateur de template de Vue. Si vous voulez cibler le contenu `v-html` dans une CSS avec portée, vous devez utiliser à la place les [modules CSS](<https://vue-loader.vuejs.org/en/features/css-modules.html>) ou un élément `