Become a

NINJA

with

Vue.js

ninja squad

Deviens un ninja avec Vue (extrait gratuit)

Ninja Squad

Table des matières

Chapter 1. Extrait gratuit

Ce que tu vas lire ici est un extrait gratuit de notre ebook sur Vue : c'est le début du livre, qui explique son but et son contenu, donne un aperçu d'ECMAScript 2015+, TypeScript, et des Web Components, décrit la philosophie de Vue, puis te propose de construire ta première application.

Cet extrait ne demande aucune connaissance préliminaire.

Chapter 2. Introduction

Alors comme ça on veut devenir un ninja ?! Ça tombe bien, tu es entre de bonnes mains ! Pour y parvenir, nous avons un bon bout de chemin à parcourir ensemble, semé d'embûches et de connaissances à acquérir.

On vit une époque excitante pour le développement web. Il y a un nouveau Vue !

En tant que développeurs "front", nous avons à notre disposition de nombreux frameworks JavaScript. React et Angular sont toujours incroyablement populaires. Comme tu le sais peut-être, nous sommes assez fans d'Angular. Je suis un contributeur régulier au framework, proche de l'équipe principale. Dans notre petite entreprise, on l'a utilisé pour construire plusieurs projets, on a formé des centaines de développeurs (oui, des centaines, littéralement), et on a même écrit un livre sur le sujet.

Angular est incroyablement productif une fois maîtrisé. Mais cela ne nous empêche pas de voir quel outil formidable est Vue.

Mon aventure avec Vue a commencé il y a quelques années, comme une journée "amusons-nous avec Vue 2". Je voulais juste voir comment Vue fonctionnait et construire une petite application pour me faire ma propre idée. Après avoir enseigné Angular pendant des années, j'ai immédiatement réalisé à quel point Vue était plus facile à apprendre tout étant très puissant. Beaucoup de choses sont similaires entre les deux frameworks, mais Vue a su trouver un bon équilibre.

Donc voilà, j'aimais bien Vue 2.x. J'avais même commencé à écrire ce livre à l'époque. Un point m'ennuyait cependant : l'intégration TypeScript était… pas terrible, dirons-nous. Et s'il y a bien une chose que j'adore quand je travaille avec Angular, c'est son intégration presque parfaite avec TypeScript.

C'est pour cela que quand Vue 3.0 a été annoncé comme une ré-écriture complète en TypeScript en septembre 2018, j'étais ravi et ai commencé à travailler à nouveau sur ce que tu es en train de lire.

J'ai suivi le développement de Vue 3 de très près, relisant chaque commit (si, si, tu as bien lu), et contribuant même quelques petites corrections de bugs et minifonctionnalités. Au même moment, nous commencions quelques projets Vue pour nos clients, et, une chose en entraînant une autre, je me retrouvais à contribuer au framework, à la bibliothèque de tests, à la bibliothèque de formulaires, et partageant mon "temps libre open-source" entre Vue et Angular.

Vue a beaucoup de points très intéressants et une vision dont peu de frameworks peuvent se targuer. Cet ebook est un "effet de bord" de mes contributions open-source, et de mon envie de partager ce que j'aime à propos de Vue. Je trouve fascinant de comprendre comment les différents frameworks résolvent des problèmes similaires, et j'espère partager mon enthousiasme avec vous ❤.

L'ambition de cet ebook est d'évoluer avec Vue. Tu recevras des mises à jour (gratuites !) avec des bonnes pratiques et de nouvelles fonctionnalités quand elles émergeront (et avec moins de fautes de frappe, parce qu'il en reste probablement malgré nos nombreuses relectures…). J'adorerais avoir tes retours, si certains chapitres ne sont pas assez clairs, si tu as repéré une erreur, ou si tu as

une meilleure solution pour certains points.

Je suis cependant assez confiant sur nos exemples de code, car ils sont extraits d'un vrai projet, et sont couverts par des douzaines de tests unitaires et de bout-en-bout. C'était la seule façon d'écrire un livre sur un framework en gestation et de repérer les problèmes qui arrivaient inévitablement avec chaque release.

Tout le code est écrit en TypeScript, parce que l'on croit vraiment que c'est un outil fantastique. Tu peux bien sûr écrire tes applications Vue en JavaScript, mais tu verras que TypeScript n'est pas très intrusif quand tu écris des applications Vue et peut t'amener une énorme plus-value. Même si, finalement, tu n'es pas convaincu·e par TypeScript (ou Vue), je suis à peu près sûr que tu vas apprendre deux-trois trucs en chemin.

Si tu as acheté notre cours en ligne, le "pack pro" (merci !), tu pourras construire une petite application morceau par morceau, tout au long du livre. Cette application s'appelle **PonyRacer**, c'est une application web où tu peux parier sur des courses de poneys. Tu peux tester cette application ici ! Vas-y, je t'attends.

Cool, non ?

Mais en plus d'être amusante, c'est une application complète. Tu devras écrire des composants, des formulaires, des tests, tu devras utiliser le routeur, appeler une API HTTP (fournie), et même faire des WebSockets. Elle utilise Vite et intègre tous les morceaux dont tu auras besoin pour construire une vraie application.

Chaque exercice viendra avec son squelette, un ensemble d'instructions et quelques tests. Quand tous les tests passent, tu as terminé l'exercice !

Les 6 premiers exercices du Pack Pro sont gratuits et disponibles à l'adresse vue-exercises.ninja-squad.com. Les autres ne sont accessibles que pour les acheteurs de notre formation en ligne. À la fin de chaque chapitre, nous listerons les exercices du Pack Pro liés aux fonctionnalités expliquées dans le chapitre, en signalant les exercices gratuits avec le symbole suivant : 🐴 , et les autres avec le symbole suivant : 🦄.

Si tu n'as pas acheté le "pack pro" (tu devrais), ne t'inquiète pas : tu apprendras tout ce dont tu auras besoin. Mais tu ne construiras pas cette application incroyable avec de beaux poneys en pixel art. Quel dommage 😊 !

Tu te rendras vite compte qu'au-delà de Vue, nous avons essayé d'expliquer les concepts au cœur du framework. Les premiers chapitres ne parlent même pas de Vue : ce sont ceux que j'appelle les "chapitres conceptuels", ils te permettront de monter en puissance avec les nouveautés intéressantes de notre domaine.

Ensuite, nous construirons progressivement notre connaissance du framework (et des bibliothèques de l'écosystème), avec les composants, les templates, les directives, la nouvelle API de Composition, les formulaires, HTTP, le routeur, les tests…

Et, enfin, nous nous attaquerons à quelques sujets avancés.

Terminons cette trop longue introduction et découvrons les nouveautés introduites dans les

dernières versions d'ECMAScript, puis intéresserons-nous à TypeScript.

Cet ebook utilise Vue version 3.5.13 dans les exemples.

Chapter 3. Une rapide introduction à ECMAScript 2015+

Si tu lis ce livre, on peut imaginer que tu as déjà entendu parler de JavaScript. Ce que l'on appelle JavaScript (JS) est une des implémentations d'une spécification standardisée, appelée ECMAScript. La version de la spécification que tu connais le plus est probablement la version 5 : c'est celle utilisée depuis de nombreuses années.

En 2015, une nouvelle version de cette spécification a été validée : ECMAScript 2015, ES2015, ou ES6, puisque c'est la sixième version de cette spécification. Et nous avons depuis une nouvelle version chaque année (ES2016, ES2017, etc.), avec à chaque fois quelques nouvelles fonctionnalités. Je l'appellerai désormais systématiquement ES2015, parce que c'est son petit nom le plus populaire, ou ES2015+ pour parler de ES2015, ES2016, ES2017, etc. Elle ajoute une tonne de fonctionnalités à JavaScript, comme les classes, les constantes, les *arrow functions*, les générateurs… Il y a tellement de choses que l'on ne peut pas tout couvrir, sauf à y consacrer entièrement ce livre. Mais Vue a été conçu pour bénéficier de cette nouvelle version de JavaScript. Même si tu peux toujours utiliser ton bon vieux JavaScript, tu auras plein d'avantages à utiliser ES2015+. Ainsi, nous allons consacrer ce chapitre à découvrir ES2015+, et voir comment il peut nous être utile pour construire une application Vue.

On va laisser beaucoup d'aspects de côté, et on ne sera pas exhaustifs sur ce que l'on verra. Si tu connais déjà ES2015+, tu peux directement sauter ce chapitre. Sinon, tu vas apprendre des trucs plutôt incroyables qui te serviront à l'avenir même si tu n'utilises finalement pas Vue !

3.1. Transpileur

La sixième version de la spécification a atteint son état final en 2015. Il est donc supporté par les navigateurs modernes, mais il y a encore des navigateurs qui ne supportent pas toute la spécification, ou qui la supportent seulement partiellement. Et bien sûr, avec une spécification maintenant annuelle (ES2016, ES2017, etc.), certains navigateurs seront toujours en retard. Ainsi, on peut se demander à quoi bon présenter le sujet s'il est toujours en pleine évolution ? Et tu as raison, car rares sont les applications qui peuvent se permettre d'ignorer les navigateurs devenus obsolètes. Mais comme tous les développeurs qui ont essayé ES2015+ ont hâte de l'utiliser dans leurs applications, la communauté a trouvé une solution : un transpileur.

Un transpileur prend du code source ES2015+ en entrée et génère du code ES5, qui peut tourner dans n'importe quel navigateur. Il génère même les fichiers *sourcemap*, qui permettent de déboguer directement le code ES2015+ depuis le navigateur. En 2015, il y avait deux outils principaux pour transpiler de l'ES2015+ :

- Traceur, un projet Google, historiquement le premier, mais désormais non maintenu.
- Babeljs, un projet démarré par Sebastian McKenzie, un jeune développeur de 17 ans (oui, ça fait mal), et qui a reçu beaucoup de contributions extérieures.

Le code source de Vue était d'ailleurs transpilé avec Babel, avant de basculer en TypeScript pour la version 3.0. TypeScript est un langage open source développé par Microsoft. C'est un sur-ensemble typé de JavaScript qui compile vers du JavaScript standard, mais nous étudierons cela très bientôt.

Pour parler franchement, Babel est biiiiien plus populaire que Traceur aujourd'hui, on aurait donc tendance à te le conseiller. Le projet est maintenant le standard de-facto.

Si tu veux jouer avec ES2015+, ou le mettre en place dans un de tes projets, jette un œil à ces transpileurs et ajoute une étape à la construction de ton projet. Elle prendra tes fichiers sources ES2015+ et générera l'équivalent en ES5. Ça fonctionne très bien, mais, évidemment, certaines nouvelles fonctionnalités sont difficiles, voire impossibles à transformer, parce qu'elles n'existent tout simplement pas en ES5. Néanmoins, l'état d'avancement actuel de ces transpileurs est largement suffisant pour les utiliser sans problème, alors jetons un coup d'œil à ces nouveautés ES2015+.

3.2. `let`

Si tu pratiques le JS depuis un certain temps, tu dois savoir que la déclaration de variable avec `var` peut être délicate. Dans à peu près tous les autres langages, une variable existe à partir de la ligne contenant la déclaration de cette variable. Mais, en JS, il y a un concept nommé *hoisting* ("remontée") qui déclare la variable au tout début de la fonction, même si tu l'as écrite plus loin.

Ainsi, déclarer une variable `name` dans le bloc `if` :

```
function getPonyFullName(pony) {
  if (pony.isChampion) {
    var name = 'Champion ' + pony.name;
    return name;
  }
  return pony.name;
}
```

est équivalent à la déclarer tout en haut de la fonction :

```
function getPonyFullName(pony) {
  var name;
  if (pony.isChampion) {
    name = 'Champion ' + pony.name;
    return name;
  }
  // name is still accessible here
  return pony.name;
}
```

ES2015 introduit un nouveau mot-clé pour la déclaration de variable, `let`, qui se comporte enfin comme on pourrait s'y attendre :

```
function getPonyFullName(pony) {
  if (pony.isChampion) {
    let name = 'Champion ' + pony.name;
    return name;
```

```
  }
  // name is not accessible here
  return pony.name;
}
```

L'accès à la variable `name` est maintenant restreint à son bloc. `let` a été pensé pour remplacer définitivement `var` à long terme, donc tu peux abandonner ce bon vieux `var` au profit de `let`. La bonne nouvelle est que ça doit être indolore, et que si ça ne l'est pas, c'est que tu as mis le doigt sur un défaut de ton code !

3.3. Constantes

Tant que l'on est sur le sujet des nouveaux mot-clés et des variables, il y en a un autre qui peut être intéressant. ES2015 introduit aussi `const` pour déclarer des… constantes ! Si tu déclares une variable avec `const`, elle doit obligatoirement être initialisée, et tu ne pourras plus lui affecter de nouvelles valeurs ensuite.

```
const poniesInRace = 6;
```

```
poniesInRace = 7; // SyntaxError
```

Comme pour les variables déclarées avec `let`, les constantes ne sont pas hoisted ("remontées") et sont bien déclarées dans leur bloc.

Il y a un détail qui peut cependant surprendre le profane. Tu peux initialiser une constante avec un objet et modifier par la suite le contenu de l'objet.

```
const PONY = {};
PONY.color = 'blue'; // works
```

Mais tu ne peux pas assigner à la constante un nouvel objet :

```
const PONY = {};
```

```
PONY = {color: 'blue'}; // SyntaxError
```

Même chose avec les tableaux :

```
const PONIES = [];
PONIES.push({ color: 'blue' }); // works
```

```
PONIES = []; // SyntaxError
```

3.4. Raccourcis pour la création d'objets

Ce n'est pas un nouveau mot-clé, mais ça peut te faire tiquer en lisant du code ES2015. Il y a un nouveau raccourci pour créer des objets, quand la propriété de l'objet que tu veux créer a le même nom que la variable utilisée comme valeur pour l'attribut.

Exemple :

```
function createPony() {
  const name = 'Rainbow Dash';
  const color = 'blue';
  return { name: name, color: color };
}
```

peut être simplifié en :

```
function createPony() {
  const name = 'Rainbow Dash';
  const color = 'blue';
  return { name, color };
}
```

Tu peux aussi utiliser un autre raccourci, quand tu veux déclarer une méthode dans un objet :

```
function createPony() {
  return {
    run: () => {
      console.log('Run!');
    }
  };
}
```

qui peut être simplifié en :

```
function createPony() {
  return {
    run() {
      console.log('Run!');
    }
  };
}
```

3.5. Affectations déstructurées

Celui-là aussi peut te faire tiquer en lisant du code ES2015. Il y a maintenant un raccourci pour affecter des variables à partir d'objets ou de tableaux.

En ES5 :

```
var httpOptions = { timeout: 2000, isCache: true };
// later
var httpTimeout = httpOptions.timeout;
var httpCache = httpOptions.isCache;
```

Maintenant, en ES2015, tu peux écrire :

```
const httpOptions = { timeout: 2000, isCache: true };
// later
const { timeout: httpTimeout, isCache: httpCache } = httpOptions;
```

Et tu auras le même résultat. Cela peut être perturbant, parce que la clé est la propriété à lire dans l'objet et la valeur est la variable à affecter. Mais cela fonctionne plutôt bien ! Et même mieux : si la variable que tu veux affecter a le même nom que la propriété de l'objet à lire, tu peux écrire simplement :

```
const httpOptions = { timeout: 2000, isCache: true };
// later
const { timeout, isCache } = httpOptions;
// you now have a variable named 'timeout'
// and one named 'isCache' with correct values
```

Le truc cool est que ça marche aussi avec des objets imbriqués :

```
const httpOptions = { timeout: 2000, cache: { age: 2 } };
// later
const {
  cache: { age }
} = httpOptions;
// you now have a variable named 'age' with value 2
```

Et la même chose est possible avec des tableaux :

```
const timeouts = [1000, 2000, 3000];
// later
const [shortTimeout, mediumTimeout] = timeouts;
// you now have a variable named 'shortTimeout' with value 1000
```

```
// and a variable named 'mediumTimeout' with value 2000
```

Bien entendu, cela fonctionne avec des tableaux de tableaux, des tableaux dans des objets, etc.

Un cas d'usage intéressant de cette fonctionnalité est la possibilité de retourner de multiples valeurs. Imagine une fonction `randomPonyInRace` qui retourne un poney et sa position dans la course.

```
function randomPonyInRace() {
  const pony = { name: 'Rainbow Dash' };
  const position = 2;
  // ...
  return { pony, position };
}

const { position, pony } = randomPonyInRace();
```

Cette nouvelle fonctionnalité de déstructuration assigne la `position` retournée par la méthode à la variable `position` et le poney à la variable `pony`. Et, si tu n'as pas usage de la position, tu peux écrire :

```
function randomPonyInRace() {
  const pony = { name: 'Rainbow Dash' };
  const position = 2;
  // ...
  return { pony, position };
}

const { pony } = randomPonyInRace();
```

Et tu auras seulement une variable `pony`.

Il est aussi possible d'assigner une valeur à la variable déstructurée si elle est `undefined` dans l'objet :

```
function randomPonyInRace() {
  const pony = { name: 'Rainbow Dash' };
  const position = undefined;
  // ...
  return { pony, position };
}

const { position = 1 } = randomPonyInRace();
```

La variable `position` contient alors 1 dans ce cas.

3.6. Paramètres optionnels et valeurs par défaut

JS a la particularité de permettre aux développeurs d'appeler une fonction avec un nombre d'arguments variables :

- si tu passes plus d'arguments que déclarés par la fonction, les arguments supplémentaires sont tout simplement ignorés (pour être tout à fait exact, tu peux quand même les utiliser dans la fonction avec la variable spéciale `arguments`).
- si tu passes moins d'arguments que déclarés par la fonction, les paramètres manquants auront la valeur `undefined`.

Ce dernier cas est celui qui nous intéresse. Souvent, on passe moins d'arguments quand les paramètres sont optionnels, comme dans l'exemple suivant :

```
function getPonies(size, page) {
  size = size || 10;
  page = page || 1;
  // ...
  server.get(size, page);
}
```

Les paramètres optionnels ont la plupart du temps une valeur par défaut. L'opérateur OR (`||`) va retourner l'opérande de droite si celui de gauche est `undefined`, comme cela serait le cas si le paramètre n'avait pas été fourni par l'appelant (pour être précis, si l'opérande de gauche est *falsy*, c'est-à-dire `undefined`, `0`, `false`, `""`, etc.). Avec cette astuce, la fonction `getPonies` peut ainsi être invoquée :

```
getPonies(20, 2);

getPonies(); // same as getPonies(10, 1);

getPonies(15); // same as getPonies(15, 1);
```

Cela fonctionnait, mais ce n'était pas évident de savoir que les paramètres étaient optionnels, sauf à lire le corps de la fonction. ES2015 offre désormais une façon plus formelle de déclarer des paramètres optionnels, dès la déclaration de la fonction :

```
function getPonies(size = 10, page = 1) {
  // ...
  server.get(size, page);
}
```

Maintenant, il est limpide que la valeur par défaut de `size` sera 10 et celle de `page` sera `1` s'ils ne sont pas fournis.

Il y a cependant une subtile différence, car `0` ou `""` sont alors des valeurs valides, et

> ne seront pas remplacées par les valeurs par défaut, comme `size = size || 10` l'aurait fait. C'est donc plutôt équivalent à `size = size === undefined ? 10 : size;`.

La valeur par défaut peut aussi être un appel de fonction :

```
function getPonies(size = defaultSize(), page = 1) {
  // the defaultSize method will be called if size is not provided
  // ...
  server.get(size, page);
}
```

ou même d'autres variables, d'autres variables globales, ou d'autres paramètres de la même fonction :

```
function getPonies(size = defaultSize(), page = size - 1) {
  // if page is not provided, it will be set to the value
  // of the size parameter minus one.
  // ...
  server.get(size, page);
}
```

Ce mécanisme de valeur par défaut ne s'applique pas qu'aux paramètres de fonction, mais aussi aux valeurs de variables, par exemple dans le cas d'une affectation déstructurée :

```
const { timeout = 1000 } = httpOptions;
// you now have a variable named 'timeout',
// with the value of 'httpOptions.timeout' if it exists
// or 1000 if not
```

3.7. *Rest operator*

ES2015 introduit aussi une nouvelle syntaxe pour déclarer un nombre variable de paramètres dans une fonction. Comme on le disait précédemment, tu peux toujours passer des arguments supplémentaires à un appel de fonction, et y accéder avec la variable spéciale `arguments`. Tu peux faire quelque chose comme :

```
function addPonies(ponies) {
  for (var i = 0; i < arguments.length; i++) {
    poniesInRace.push(arguments[i]);
  }
}

addPonies('Rainbow Dash', 'Pinkie Pie');
```

Mais tu seras d'accord pour dire que ce n'est ni élégant, ni évident : le paramètre `ponies` n'est jamais

utilisé, et rien n'indique que l'on peut fournir plusieurs poneys.

ES2015 propose une syntaxe bien meilleure, grâce au *rest operator* `...` ("opérateur de reste").

```
function addPonies(...ponies) {
  for (const pony of ponies) {
    poniesInRace.push(pony);
  }
}
```

`ponies` est désormais un véritable tableau, sur lequel on peut itérer. La boucle `for ... of` utilisée pour l'itération est aussi une nouveauté d'ES2015. Elle permet d'être sûr de n'itérer que sur les valeurs de la collection, et non pas sur ses propriétés comme `for ... in`. Ne trouves-tu pas que notre code est maintenant bien plus beau et lisible ?

Le *rest operator* peut aussi fonctionner avec des affectations déstructurées :

```
const [winner, ...losers] = poniesInRace;
// assuming 'poniesInRace' is an array containing several ponies
// 'winner' will have the first pony,
// and 'losers' will be an array of the other ones
```

Le *rest operator* ne doit pas être confondu avec le *spread operator* ("opérateur d'étalement"), même si, on te l'accorde, ils se ressemblent dangereusement ! Le *spread operator* est son opposé : il prend un tableau et l'étale en arguments variables. Le seul cas d'utilisation qui me vient à l'esprit serait pour les fonctions comme `min` ou `max`, qui peuvent recevoir des arguments variables, et que tu voudrais appeler avec un tableau :

```
const ponyPrices = [12, 3, 4];
const minPrice = Math.min(...ponyPrices);
```

3.8. Classes

Une des fonctionnalités les plus emblématiques : ES2015 introduit les classes en JavaScript ! Tu pourras dorénavant facilement faire de l'héritage de classes en JavaScript. C'était déjà possible, avec l'héritage prototypal, mais ce n'était pas une tâche aisée, surtout pour les débutants...

Maintenant, c'est les doigts dans le nez, regarde :

```
class Pony {
  constructor(color) {
    this.color = color;
  }

  toString() {
    return `${this.color} pony`;
```

```
    // see that? It is another cool feature of ES2015, called template literals
    // we'll talk about these quickly!
  }
}

const bluePony = new Pony('blue');
console.log(bluePony.toString()); // blue pony
```

Les déclarations de classes, contrairement aux déclarations de fonctions, ne sont pas *hoisted* ("remontées"), donc tu dois déclarer une classe avant de l'utiliser. Tu as probablement remarqué la fonction spéciale `constructor`. C'est le constructeur, la fonction appelée à la création d'un nouvel objet avec le mot-clé `new`. Dans l'exemple, il requiert une couleur, et nous créons une nouvelle instance de la classe `Pony` avec la couleur "blue". Une classe peut aussi avoir des méthodes, appelables sur une instance, comme la méthode `toString()` dans l'exemple.

Une classe peut aussi avoir des attributs et des méthodes statiques :

```
class Pony {
  static defaultSpeed() {
    return 10;
  }
}
```

Ces méthodes statiques ne peuvent être appelées que sur la classe directement :

```
const speed = Pony.defaultSpeed();
```

Une classe peut avoir des accesseurs (*getters, setters*), si tu veux implémenter du code sur ces opérations :

```
class Pony {
  get color() {
    console.log('get color');
    return this._color;
  }

  set color(newColor) {
    console.log(`set color ${newColor}`);
    this._color = newColor;
  }
}
const pony = new Pony();
pony.color = 'red';
// 'set color red'
console.log(pony.color);
// 'get color'
```

```
// 'red'
```

Et, bien évidemment, si tu as des classes, l'héritage est possible en ES2015.

```
class Animal {
  speed() {
    return 10;
  }
}
class Pony extends Animal {}
const pony = new Pony();
console.log(pony.speed()); // 10, as Pony inherits the parent method
```

`Animal` est appelée la classe de base et `Pony` la classe dérivée. Comme tu peux le voir, la classe dérivée possède toutes les méthodes de la classe de base. Mais elle peut aussi les redéfinir :

```
class Animal {
  speed() {
    return 10;
  }
}
class Pony extends Animal {
  speed() {
    return super.speed() + 10;
  }
}
const pony = new Pony();
console.log(pony.speed()); // 20, as Pony overrides the parent method
```

Comme tu peux le voir, le mot-clé `super` permet d'invoquer la méthode de la classe de base, avec `super.speed()` par exemple.

Ce mot-clé `super` peut aussi être utilisé dans les constructeurs, pour invoquer le constructeur de la classe de base :

```
class Animal {
  constructor(speed) {
    this.speed = speed;
  }
}
class Pony extends Animal {
  constructor(speed, color) {
    super(speed);
    this.color = color;
  }
}
const pony = new Pony(20, 'blue');
```

```
console.log(pony.speed); // 20
```

3.9. *Promises*

Les *promises* ("promesses") ne sont pas si nouvelles, et tu les connais ou les utilises peut-être déjà, parce qu'elles existaient déjà via des bibliothèques tierces. Mais, comme nous les utiliserons beaucoup avec Vue, et même si tu n'utilises que du pur JS sans Vue, on pense que c'est important de s'y attarder un peu.

L'objectif des *promises* est de simplifier la programmation asynchrone. Notre code JS est plein d'asynchronisme, comme des requêtes AJAX, et, en général, on utilise des *callbacks* pour gérer le résultat et l'erreur. Mais le code devient vite confus, avec des *callbacks* dans des *callbacks*, qui le rendent illisible et peu maintenable. Les *promises* sont plus pratiques que les *callbacks*, parce qu'elles permettent d'écrire du code à plat, et le rendent ainsi plus simple à comprendre. Prenons un cas d'utilisation simple, où on doit récupérer un utilisateur, puis ses droits, puis mettre à jour un menu quand on a récupéré tout ça.

Avec des *callbacks* :

```
getUser(login, function (user) {
  getRights(user, function (rights) {
    updateMenu(rights);
  });
});
```

Avec des *promises* :

```
getUser(login)
  .then(function (user) {
    return getRights(user);
  })
  .then(function (rights) {
    updateMenu(rights);
  })
```

J'aime cette version, parce qu'elle s'exécute comme elle se lit : je veux récupérer un utilisateur, puis ses droits, puis mettre à jour le menu.

Une *promise* est un objet *thenable*, ce qui signifie simplement qu'il a une méthode `then`. Cette méthode prend deux arguments : un callback de succès et un callback d'erreur. Une *promise* a trois états :

- *pending* ("en cours") : quand la *promise* n'est pas réalisée, par exemple quand l'appel serveur n'est pas encore terminé.
- *fulfilled* ("réalisée") : quand la *promise* s'est réalisée avec succès, par exemple quand l'appel HTTP serveur a retourné un status 200-OK.

- *rejected* ("rejetée") : quand la *promise* a échoué, par exemple si l'appel HTTP serveur a retourné un status 404-NotFound.

Quand la promesse est réalisée (*fulfilled*), alors le callback de succès est invoqué, avec le résultat en argument. Si la promesse est rejetée (*rejected*), alors le callback d'erreur est invoqué, avec la valeur rejetée ou une erreur en argument.

Alors, comment crée-t-on une *promise* ? C'est simple, il y a une nouvelle classe `Promise`, dont le constructeur attend une fonction avec deux paramètres, `resolve` et `reject`.

```
const getUser = function (_login) {
  return new Promise(function (resolve, reject) {
    // async stuff, like fetching users from server, returning a response
    if (response.status === 200) {
      resolve(response.data);
    } else {
      reject('No user');
    }
  });
};
```

Une fois la *promise* créée, tu peux enregistrer des *callbacks*, via la méthode `then`. Cette méthode peut recevoir deux arguments, les deux *callbacks* que tu veux voir invoqués en cas de succès ou en cas d'échec. Dans l'exemple suivant, nous passons simplement un seul *callback* de succès, ignorant ainsi une erreur potentielle :

```
getUser(login)
  .then(function (user) {
    console.log(user);
  })
```

Quand la promesse sera réalisée, le callback de succès (qui se contente ici de tracer l'utilisateur en console) sera invoqué.

La partie la plus cool, c'est que le code peut s'écrire à plat. Si par exemple ton *callback* de succès retourne lui aussi une *promise*, tu peux écrire :

```
getUser(login)
  .then(function (user) {
    return getRights(user) // getRights is returning a promise
      .then(function (rights) {
        return updateMenu(rights);
      });
  })
```

ou plus élégamment :

```
getUser(login)
  .then(function (user) {
    return getRights(user); // getRights is returning a promise
  })
  .then(function (rights) {
    return updateMenu(rights);
  })
```

Un autre truc cool est la gestion d'erreur : tu peux définir une gestion d'erreur par *promise* ou globale à toute la chaîne.

Une gestion d'erreur par *promise* :

```
getUser(login)
  .then(
    function (user) {
      return getRights(user);
    },
    function (error) {
      console.log(error); // will be called if getUser fails
      return Promise.reject(error);
    }
  )
  .then(
    function (rights) {
      return updateMenu(rights);
    },
    function (error) {
      console.log(error); // will be called if getRights fails
      return Promise.reject(error);
    }
  )
```

Une gestion d'erreur globale pour toute la chaîne :

```
getUser(login)
  .then(function (user) {
    return getRights(user);
  })
  .then(function (rights) {
    return updateMenu(rights);
  })
  .catch(function (error) {
    console.log(error); // will be called if getUser or getRights fails
  })
```

Tu devrais sérieusement t'intéresser aux *promises*, parce que ça va devenir la nouvelle façon

d'écrire des APIs, et toutes les bibliothèques vont bientôt les utiliser. Même les bibliothèques standards : c'est le cas de la nouvelle Fetch API par exemple.

3.10. (*arrow functions*)

Un truc que j'adore dans ES2015 est la nouvelle syntaxe *arrow function* ("fonction flèche"), utilisant l'opérateur *fat arrow* ("grosse flèche") : `⇒`. C'est très utile pour les *callbacks* et les fonctions anonymes !

Prenons notre exemple précédent avec des *promises* :

```
getUser(login)
  .then(function (user) {
    return getRights(user); // getRights is returning a promise
  })
  .then(function (rights) {
    return updateMenu(rights);
  })
```

Il peut être réécrit avec des *arrow functions* comme ceci :

```
getUser(login)
  .then(user => getRights(user))
  .then(rights => updateMenu(rights))
```

N'est-ce pas super cool ?!

Note que le `return` est implicite s'il n'y a pas de bloc : pas besoin d'écrire `user ⇒ return getRights(user)`. Mais, si nous avions un bloc, nous aurions besoin d'un `return` explicite :

```
getUser(login)
  .then(user => {
    console.log(user);
    return getRights(user);
  })
  .then(rights => updateMenu(rights))
```

Et les *arrow functions* ont une particularité bien agréable que n'ont pas les fonctions normales : le `this` reste attaché lexicalement, ce qui signifie que ces *arrow functions* n'ont pas un nouveau `this` comme les fonctions normales. Prenons un exemple où on itère sur un tableau avec la fonction `map` pour y trouver le maximum.

En ES5 :

```
var maxFinder = {
  max: 0,
```

```
  find: function (numbers) {
    // let's iterate
    numbers.forEach(function (element) {
      // if the element is greater, set it as the max
      if (element > this.max) {
        this.max = element;
      }
    });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

Ça semble pas mal, non ? Mais en fait ça ne marche pas… Si tu as de bons yeux, tu as remarqué que le `forEach` dans la fonction `find` utilise `this`, mais ce `this` n'est lié à aucun objet. Donc `this.max` n'est en fait pas le `max` de l'objet `maxFinder`… On pourrait corriger ça facilement avec un alias :

```
var maxFinder = {
  max: 0,
  find: function (numbers) {
    var self = this;
    numbers.forEach(function (element) {
      if (element > self.max) {
        self.max = element;
      }
    });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

ou en *bindant* le `this` :

```
var maxFinder = {
  max: 0,
  find: function (numbers) {
    numbers.forEach(
      function (element) {
        if (element > this.max) {
          this.max = element;
        }
      }.bind(this)
    );
  }
```

```
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

ou en le passant en second paramètre de la fonction `forEach` (ce qui est justement sa raison d'être) :

```
var maxFinder = {
  max: 0,
  find: function (numbers) {
    numbers.forEach(function (element) {
      if (element > this.max) {
        this.max = element;
      }
    }, this);
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

Mais il y a maintenant une solution bien plus élégante avec les *arrow functions* :

```
const maxFinder = {
  max: 0,
  find: function (numbers) {
    numbers.forEach(element => {
      if (element > this.max) {
        this.max = element;
      }
    });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

Les *arrow functions* sont donc idéales pour les fonctions anonymes en *callback* !

3.11. Async/await

Nous discutions des promesses précédemment, et il est intéressant de connaître un autre mot-clé introduit pour les gérer de façon plus synchrone : `await`.

Cette fonctionnalité n'est pas introduite par ECMAScript 2015, mais par ECMAScript 2017, et pour utiliser `await`, ta fonction doit être marquée comme `async`. Quand tu utilises le mot-clé `await` devant une promesse, tu pauses l'exécution de la fonction `async`, attends la résolution de la promesse, puis reprends l'exécution de la fonction `async`. La valeur retournée est la valeur résolue par la promesse.

On peut donc écrire notre exemple précédent en utilisant `async/await` comme ceci :

```
async function getUserRightsAndUpdateMenu() {
  // getUser is a promise
  const user = await getUser(login);
  // getRights is a promise
  const rights = await getRights(user);
  updateMenu(rights);
}
await getUserRightsAndUpdateMenu();
```

Et notre code a l'air complètement synchrone ! Une autre fonctionnalité assez cool de `async/await` est la possibilité d'utiliser un simple `try/catch` pour gérer les erreurs :

```
async function getUserRightsAndUpdateMenu() {
  try {
    // getUser is a promise
    const user = await getUser(login);
    // getRights is a promise
    const rights = await getRights(user);
    updateMenu(rights);
  } catch (e) {
    // will be called if getUser, getRights or updateMenu fails
    console.log(e);
  }
}
await getUserRightsAndUpdateMenu();
```

Note que, même si le code ressemble à du code synchrone, il reste asynchrone. L'exécution de la fonction est mise en pause puis reprise, mais, comme avec les callbacks, cela ne bloque pas le fil d'exécution : les autres événements peuvent être gérés pendant que l'exécution est mise en pause.

3.12. *Set* et *Map*

On va faire court : on a maintenant de vraies collections en ES2015. Youpi \o/ !

On utilisait jusque-là de simples objets JavaScript pour jouer le rôle de *map* ("dictionnaire"), c'est-à-dire un objet JS standard, dont les clés étaient nécessairement des chaînes de caractères. Mais nous pouvons maintenant utiliser la nouvelle classe *Map* :

```
const cedric = { id: 1, name: 'Cedric' };
const users = new Map();
```

```
users.set(cedric.id, cedric); // adds a user
console.log(users.has(cedric.id)); // true
console.log(users.size); // 1
users.delete(cedric.id); // removes the user
```

On a aussi une classe `Set` ("ensemble") :

```
const cedric = { id: 1, name: 'Cedric' };
const users = new Set();
users.add(cedric); // adds a user
console.log(users.has(cedric)); // true
console.log(users.size); // 1
users.delete(cedric); // removes the user
```

Tu peux aussi itérer sur une collection, avec la nouvelle syntaxe `for ⋯ of` :

```
for (const user of users) {
  console.log(user.name);
}
```

Tu verras que cette syntaxe `for ⋯ of` est aussi supportée par Vue pour itérer sur une collection dans un template.

3.13. Template de string

Construire des strings a toujours été pénible en JavaScript, où nous devions généralement utiliser des concaténations :

```
const fullname = 'Miss ' + firstname + ' ' + lastname;
```

Les templates de string sont une nouvelle fonctionnalité mineure, mais bien pratique, où on doit utiliser des accents graves (*backticks* `) au lieu des habituelles apostrophes (*quote* ') ou apostrophes doubles (*double-quotes* "), fournissant un moteur de template basique avec support du multiligne :

```
const fullname = `Miss ${firstname} ${lastname}`;
```

Le support du multiligne est particulièrement adapté à l'écriture de morceaux d'HTML, comme nous le ferons dans nos composants Vue :

```
const template = `<div>
  <h1>Hello</h1>
</div>`;
```

Une dernière fonctionnalité est la possibilité de les "tagger". Tu peux définir une fonction, et l'appliquer sur une chaîne de caractères template. Ici `askQuestion` ajoute un point d'interrogation à la fin de la chaîne de caractère :

```
const askQuestion = strings => strings + '?';
const template = askQuestion`Hello there`;
```

Mais quelle est la différence avec une fonction classique alors ? Une fonction de tag reçoit en fait plusieurs paramètres :

- un tableau des morceaux statiques de la chaîne de caractères
- les valeurs résultant de l'évaluation des expressions

Par exemple, si l'on a la chaîne de caractère template contenant les expressions suivantes :

```
const person1 = 'Cedric';
const person2 = 'Agnes';
const template = `Hello ${person1}! Where is ${person2}?`;
```

alors la fonction de tag reçoit les différents morceaux statiques et dynamiques. Ici, nous avons une fonction de tag qui passe en majuscule les noms des protagonistes :

```
const uppercaseNames = (strings, ...values) => {
  // `strings` is an array with the static parts ['Hello ', '! Where is ', '?']
  // `values` is an array with the evaluated expressions ['Cedric', 'Agnes']
  const names = values.map(name => name.toUpperCase());
  // `names` now has ['CEDRIC', 'AGNES']
  // let's merge the `strings` and `names` arrays
  return strings.map((string, i) => `${string}${names[i] ? names[i] : ''}`).join('');
};
const result = uppercaseNames`Hello ${person1}! Where is ${person2}?`;

// returns 'Hello CEDRIC! Where is AGNES?'
```

Passons à l'un des grands changements introduits : les modules.

3.14. Modules

Il a toujours manqué en JavaScript une façon standard de ranger ses fonctions dans un espace de nommage, et de charger dynamiquement du code. Node.js a été un leader sur le sujet, avec un écosystème très riche de modules utilisant la convention CommonJS. Côté navigateur, il y a aussi l'API AMD (Asynchronous Module Definition), utilisée par RequireJS. Mais aucun n'était un vrai standard, ce qui nous conduit à des débats incessants sur la meilleure solution.

ES2015+ a pour objectif de créer une syntaxe avec le meilleur des deux mondes, sans se préoccuper de l'implémentation utilisée. Le comité Ecma TC39 (qui est responsable des évolutions d'ES2015+ et

auteur de la spécification du langage) voulait une syntaxe simple (c'est indéniablement l'atout de CommonJS), mais avec le support du chargement asynchrone (comme AMD), et avec quelques bonus comme la possibilité d'analyser statiquement le code par des outils et une gestion claire des dépendances cycliques. Cette nouvelle syntaxe se charge de déclarer ce que tu exportes depuis tes modules, et ce que tu importes dans d'autres modules.

Cette gestion des modules est fondamentale dans Vue, parce que tout y est défini dans des modules, qu'il faut importer dès que l'on veut les utiliser. Supposons que l'on veuille exposer une fonction pour parier sur un poney donné dans une course et une fonction pour lancer la course.

Dans `races.service.js`:

```
export function bet(race, pony) {
  // ...
}
export function start(race) {
  // ...
}
```

Comme tu le vois, c'est plutôt simple : le nouveau mot-clé `export` fait son travail et exporte les deux fonctions.

Maintenant, supposons qu'un composant de notre application veuille appeler ces deux fonctions.

Dans un autre fichier :

```
import { bet, start } from './races.service';
```

```
// later
bet(race, pony1);
start(race);
```

C'est ce que l'on appelle un *named export* ("export nommé"). Ici, on importe les deux fonctions, et on doit spécifier le nom du fichier contenant ces deux fonctions, ici 'races.service'. Évidemment, on peut importer une seule des deux fonctions, si besoin avec un alias :

```
import { start as startRace } from './races.service';
```

```
// later
startRace(race);
```

Et si tu veux importer toutes les fonctions du module, tu peux utiliser le caractère joker `*`.

Comme tu le ferais dans d'autres langages, il faut utiliser le caractère joker `*` avec modération, seulement si tu as besoin de toutes les fonctions ou la plupart. Et comme tout ceci sera

prochainement géré par ton IDE préféré qui prendra en charge la gestion automatique des imports, on n'aura plus à se soucier d'importer les seules bonnes fonctions.

Avec un caractère joker, tu dois utiliser un alias, et j'aime plutôt ça, parce que ça rend le reste du code plus lisible :

```
import * as racesService from './races.service';
```

```
// later
racesService.bet(race, pony1);
racesService.start(race);
```

Si ton module n'expose qu'une seule fonction, ou valeur, ou classe, tu n'as pas besoin d'utiliser un *named export*, et tu peux bénéficier de l'export par défaut, avec le mot-clé `default`. C'est pratique pour les classes notamment :

```
// pony.js
export default class Pony {}
// races.service.js
import Pony from './pony';
```

Note l'absence d'accolade pour importer un export par défaut. Tu peux l'importer avec l'alias que tu veux, mais pour être cohérent, c'est mieux de l'importer avec le nom du module (sauf évidemment si tu importes plusieurs modules portant le même nom, auquel cas, tu devras donner un alias pour les distinguer). Et, bien sûr, tu peux mélanger l'export par défaut et l'export nommé, mais un module ne pourra avoir qu'un seul export par défaut.

En Vue, tu utiliseras beaucoup de ces imports dans ton application. Chaque composant et service sera une classe, généralement isolée dans son propre fichier et exportée, et ensuite importée à la demande dans chaque autre composant.

3.15. Conclusion

Voilà qui conclue notre rapide introduction à ES2015+. On a zappé quelques parties, mais si tu as bien assimilé ce chapitre, tu n'auras aucun problème à coder ton application en ES2015+. Si tu veux approfondir, je te recommande chaudement Exploring JS par Axel Rauschmayer, ou Understanding ES6 par Nicholas C. Zakas. Ces deux ebooks peuvent être lus gratuitement en ligne, mais pense à soutenir ces auteurs qui ont fait un beau travail ! En l'occurrence, j'ai relu récemment Speaking JS, le précédent livre d'Axel, et j'ai encore appris quelques trucs, donc si tu veux rafraîchir tes connaissances en JS, je te le conseille vivement !

Chapter 4. Un peu plus loin qu'ES2015+

4.1. Types dynamiques, statiques et optionnels

Tu sais probablement que les applications Vue peuvent être écrites en ES5, ES2015+, ou TypeScript. Et tu te demandes peut-être qu'est-ce que TypeScript, et ce qu'il apporte de plus.

JavaScript est dynamiquement typé. Tu peux donc faire des trucs comme :

```
let pony = 'Rainbow Dash';
pony = 2;
```

Et ça fonctionne. Ça offre plein de possibilités : tu peux ainsi passer n'importe quel objet à une fonction, tant que cet objet a les propriétés requises par la fonction :

```
const pony = { name: 'Rainbow Dash', color: 'blue' };
const horse = { speed: 40, color: 'black' };
const printColor = animal => console.log(animal.color);
// works as long as the object has a `color` property
```

Cette nature dynamique est formidable, mais elle est aussi un handicap dans certains cas, comparée à d'autres langages plus fortement typés. Le cas le plus évident est quand tu dois appeler une fonction inconnue d'une autre API en JS : tu dois lire la documentation (ou pire le code de la fonction) pour deviner à quoi doivent ressembler les paramètres. Dans notre exemple précédent, la méthode `printColor` attend un paramètre avec une propriété `color`, mais encore faut-il le savoir. Et c'est encore plus difficile dans notre travail quotidien, où on multiplie les utilisations de bibliothèques et services développés par d'autres. Un des co-fondateurs de Ninja Squad se plaint souvent du manque de type en JS, et déclare qu'il n'est pas aussi productif, et qu'il ne produit pas du code aussi bon qu'il le ferait dans un environnement plus statiquement typé. Et il n'a pas entièrement tort, même s'il trolle aussi par plaisir ! Sans les informations de type, les IDEs n'ont aucun indice pour savoir si tu écris quelque chose de faux et les outils ne peuvent pas t'aider à trouver des bugs dans ton code. Bien sûr, nos applications sont testées et Vue a toujours facilité les tests, mais c'est pratiquement impossible d'avoir une parfaite couverture de tests.

Cela nous amène au sujet de la maintenabilité. Le code JS peut être difficile à maintenir, malgré les tests et la documentation. Refactoriser une grosse application JS n'est pas chose aisée, comparativement à ce qui peut être fait dans des langages statiquement typés. La maintenabilité est un sujet important, et les types aident les outils, ainsi que les développeurs, à éviter les erreurs lors de l'écriture et la modification de code.

Vue commença comme un pur framework JavaScript, mais les contributeurs voulaient nous aider à écrire du meilleur JS en ajoutant des informations de type à notre code. Ce n'est pas un nouveau concept pour JS, c'était même le sujet de la spécification ECMAScript 4, qui a été abandonnée.

C'est pourquoi Vue supporte TypeScript, le langage de Microsoft, depuis Vue 2.0. Le support de TypeScript dans Vue 2.0 n'était cependant pas parfait. Quand Vue 3.0 a été annoncé, l'une des

grandes nouveautés était le support amélioré de TypeScript : en fait, le framework lui-même est maintenant écrit en TypeScript.

4.2. Hello TypeScript

Je pense que c'était la meilleure chose à faire pour plusieurs raisons. TypeScript est très populaire, avec une communauté et un écosystème actifs. Nous l'utilisons beaucoup pour construire nos applications *front-end*, et, honnêtement, c'est un très bon langage.

TypeScript est un projet de Microsoft, mais ce n'est pas le Microsoft de l'ère Ballmer et Gates. C'est le Microsoft de Nadella, celui qui s'ouvre à la communauté, et donc, à l'open-source.

Mais la raison principale de parier sur TypeScript est le système de types qu'il offre. C'est un système optionnel qui vient t'aider sans t'entraver. De fait, après avoir codé quelque temps avec, il est difficile de revenir à du pur JavaScript. Tu peux toujours écrire des applications Vue juste en utilisant JavaScript, mais TypeScript améliore vraiment l'expérience.

Comme je le disais, j'aime beaucoup ce langage, et on jettera un coup d'œil à TypeScript dans le chapitre suivant. Tu pourras ainsi lire et comprendre n'importe quel code Vue, et tu pourras décider de l'utiliser, ou pas, ou juste un peu, dans tes applications.

Si tu te demandes "mais pourquoi avoir du code fortement typé dans une application Vue ?". D'après notre expérience, la raison principale est la facilité de refactoring. On a généralement des types qui représentent nos entités métiers : un `User`, un `Account`, une `Invoice`, etc. Mais il est assez rare d'avoir une modélisation parfaite au premier essai. Au cours du temps, les entités évoluent, grossissent, se divisent en d'autres entités. Les champs sont renommés, et dans une application JavaScript, tu n'as pas vraiment de garantie que tu n'as pas cassé la moitié de tes pages… C'est là où TypeScript brille : le compilateur va te guider dans les changements à faire et tu dormiras paisiblement la nuit.

C'est pourquoi nous allons passer un peu de temps à apprendre TypeScript (TS). Et peut-être qu'un jour le système de type sera approuvé par le comité de standardisation, et il sera normal d'avoir de vrais types en JS.

Il est temps désormais de se lancer dans TypeScript !

Chapter 5. Découvrir TypeScript

TypeScript, qui existe depuis 2012, est un sur-ensemble de JavaScript, ajoutant quelques trucs à ES5. Le plus important étant le système de type, lui donnant même son nom. Depuis la version 1.5, sortie en 2015, cette bibliothèque essaie d'être un sur-ensemble d'ES2015, incluant toutes les fonctionnalités vues précédemment et quelques nouveautés, comme les décorateurs. Écrire du TypeScript ressemble à écrire du JavaScript. Par convention les fichiers sources TypeScript ont l'extension `.ts`, et seront compilés en JavaScript standard, en général lors du build, avec le compilateur TypeScript. Le code généré reste très lisible.

```
npm install -g typescript
tsc test.ts
```

Mais commençons par le début.

5.1. Les types de TypeScript

La syntaxe pour ajouter des informations de type en TypeScript est basique :

```
let variable: type;
```

Les différents types sont simples à retenir :

```
const ponyNumber: number = 0;
const ponyName: string = 'Rainbow Dash';
```

Dans ces cas, les types sont facultatifs, car le compilateur TS peut les deviner depuis leur valeur (c'est ce que l'on appelle l'inférence de type).

Le type peut aussi être défini dans ton application, avec, par exemple, la classe suivante `Pony` :

```
const pony: Pony = new Pony();
```

TypeScript supporte aussi ce que certains langages appellent des types génériques, par exemple avec un `Array` :

```
const ponies: Array<Pony> = [new Pony()];
```

Cet `Array` ne peut contenir que des poneys, ce qu'indique la notation générique `<>`. On peut se demander quel est l'intérêt d'imposer cela. Ajouter de telles informations de type aidera le compilateur à détecter des erreurs :

```
ponies.push('hello'); // error TS2345
// Argument of type 'string' is not assignable to parameter of type 'Pony'.
```

Et comment faire si tu as besoin d'une variable pouvant recevoir plusieurs types ? TS a un type spécial pour cela, nommé `any`.

```
let changing: any = 2;
changing = true; // no problem
```

C'est pratique si tu ne connais pas le type d'une valeur, soit parce qu'elle vient d'un bout de code dynamique, ou en sortie d'une bibliothèque obscure.

Si ta variable ne doit recevoir que des valeurs de type `number` ou `boolean`, tu peux utiliser l'union de types :

```
let changing: number | boolean = 2;
changing = true; // no problem
```

5.2. Valeurs énumérées (*enum*)

TypeScript propose aussi des valeurs énumérées : `enum`. Par exemple, une course de poneys dans ton application peut être soit `ready`, `started` ou `done`.

```
enum RaceStatus {
  Ready,
  Started,
  Done
}
```

```
const race = new Race();
race.status = RaceStatus.Ready;
```

Un `enum` est en fait une valeur numérique, commençant à 0. Tu peux cependant définir la valeur que tu veux :

```
enum Medal {
  Gold = 1,
  Silver,
  Bronze
}
```

Depuis TypeScript 2.4, tu peux même donner une valeur sous forme de chaîne de caractères :

```
enum RacePosition {
  First = 'First',
  Second = 'Second',
  Other = 'Other'
}
```

Cependant, pour être tout à fait honnêtes, nous n'utilisons pas d'enum dans nos projets : on utilise des unions de types. Elles sont plus simples et couvrent à peu près les mêmes usages :

```
let color: 'blue' | 'red' | 'green';
// we can only give one of these values to `color`
color = 'blue';
```

TypeScript permet même de créer ses propres types, on pourrait donc faire comme suit :

```
type Color = 'blue' | 'red' | 'green';
const ponyColor: Color = 'blue';
```

5.3. Return types

Tu peux aussi spécifier le type de retour d'une fonction :

```
function startRace(race: Race): Race {
  race.status = RaceStatus.Started;
  return race;
}
```

Si la fonction ne retourne rien, tu peux le déclarer avec `void` :

```
function startRace(race: Race): void {
  race.status = RaceStatus.Started;
}
```

5.4. Interfaces

C'est déjà une bonne première étape. Mais, comme je le disais plus tôt, JavaScript est formidable par sa nature dynamique. Une fonction marchera si elle reçoit un objet possédant la bonne propriété :

```
function addPointsToScore(player, points) {
  player.score += points;
}
```

Cette fonction peut être appliquée à n'importe quel objet ayant une propriété `score`. Maintenant comment traduit-on cela en TypeScript ? Facile !, on définit une interface, un peu comme la "forme" de l'objet.

```
function addPointsToScore(player: { score: number }, points: number): void {
  player.score += points;
}
```

Cela signifie que le paramètre doit avoir une propriété nommée `score` de type `number`. Tu peux évidement aussi nommer ces interfaces :

```
interface HasScore {
  score: number;
}
```

```
function addPointsToScore(player: HasScore, points: number): void {
  player.score += points;
}
```

Tu verras que l'on utilise très souvent les interfaces dans le livre pour représenter nos entités.

On utilise également les interfaces pour représenter nos modèles métiers dans nos projets. Généralement, on ajoute un suffixe `Model` pour le montrer de façon claire. Il est alors très facile de créer une nouvelle entité :

```
interface PonyModel {
  name: string;
  speed: number;
}
const pony: PonyModel = { name: 'Light Shoe', speed: 56 };
```

5.5. Paramètre optionnel

Y'a un autre truc sympa en JavaScript : les paramètres optionnels. Si tu ne les passes pas à l'appel de la fonction, leur valeur sera `undefined`. Mais, en TypeScript, si tu déclares une fonction avec des paramètres typés, le compilateur te gueulera dessus si tu les oublies :

```
addPointsToScore(player); // error TS2346
// Supplied parameters do not match any signature of call target.
```

Pour montrer qu'un paramètre est optionnel dans une fonction (ou une propriété dans une interface), tu ajoutes `?` après le paramètre. Ici, le paramètre `points` est optionnel :

```
function addPointsToScore(player: HasScore, points?: number): void {
  points = points || 0;
  player.score += points;
}
```

5.6. Des fonctions en propriété

Tu peux aussi décrire un paramètre comme devant posséder une fonction spécifique plutôt qu'une propriété :

```
function startRunning(pony: Pony) {
  pony.run(10);
}
```

La définition de cette interface serait :

```
interface CanRun {
  run(meters: number): void;
}
```

```
function startRunning(pony: CanRun): void {
  pony.run(10);
}

const ponyOne = {
  run: (meters: number) => logger.log(`pony runs ${meters}m`)
};
startRunning(ponyOne);
```

5.7. Classes

Une classe peut implémenter une interface. Pour nous, un poney peut courir, donc on pourrait écrire :

```
class Pony implements CanRun {
  run(meters: number) {
    logger.log(`pony runs ${meters}m`);
  }
}
```

Le compilateur nous obligera à implémenter la méthode `run` dans la classe. Si nous l'implémentons mal, par exemple en attendant une `string` au lieu d'un `number`, le compilateur va crier :

```
class IllegalPony implements CanRun {
  run(meters: string) {
    console.log(`pony runs ${meters}m`);
  }
}
// error TS2420: Class 'IllegalPony' incorrectly implements interface 'CanRun'.
// Types of property 'run' are incompatible.
```

Tu peux aussi implémenter plusieurs interfaces si ça te fait plaisir :

```
class HungryPony implements CanRun, CanEat {
  run(meters: number) {
    logger.log(`pony runs ${meters}m`);
  }

  eat() {
    logger.log(`pony eats`);
  }
}
```

Et une interface peut en étendre une ou plusieurs autres :

```
interface Animal extends CanRun, CanEat {}

class Pony implements Animal {
  // ...
}
```

Une classe en TypeScript peut avoir des propriétés et des méthodes. Avoir des propriétés dans une classe n'est pas une fonctionnalité standard d'ES2015, c'est seulement possible en TypeScript.

```
class SpeedyPony {
  speed = 10;

  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}
```

Tout est public par défaut. Mais tu peux utiliser le mot-clé `private` pour cacher une propriété ou une méthode. Ajouter `public` ou `private` à un paramètre de constructeur est un raccourci pour créer et initialiser un membre privé ou public :

```
class NamedPony {
  constructor(
```

```
    public name: string,
    private speed: number
  ) {}

  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}
```

```
const pony = new NamedPony('Rainbow Dash', 10);
// defines a public property name with 'Rainbow Dash'
// and a private one speed with 10
```

Ce qui est l'équivalent du plus verbeux :

```
class NamedPonyWithoutShortcut {
  public name: string;
  private speed: number;

  constructor(name: string, speed: number) {
    this.name = name;
    this.speed = speed;
  }

  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}
```

Ces raccourcis sont très pratiques et nous allons beaucoup les utiliser en Vue !

5.8. Utiliser d'autres bibliothèques

Mais, si on travaille avec des bibliothèques externes écrites en JS, comment savoir les types des paramètres attendus par telle fonction de telle bibliothèque ? La communauté TypeScript est tellement cool que ses membres ont défini des définitions pour les types et les fonctions exposés par les bibliothèques JavaScript les plus populaires.

Les fichiers contenant ces définitions ont une extension spéciale : `.d.ts`. Ils contiennent une liste de toutes les fonctions publiques des bibliothèques. DefinitelyTyped est l'outil de référence pour récupérer ces fichiers. Par exemple, si tu veux utiliser la bibliothèque de test Jest dans ton application TypeScript, tu peux récupérer le fichier dédié depuis le repository directement avec NPM :

```
npm install --save-dev @types/jest
```

Maintenant, si tu te trompes dans l'appel d'une méthode Jest, le compilateur te le dira, et tu peux corriger !

Encore plus cool, depuis TypeScript 1.6, le compilateur est capable de trouver par lui-même ces définitions pour une dépendance si elles sont packagées avec la dépendance elle-même. De plus en plus de projets adoptent cette approche et Vue fait de même. Tu n'as donc même pas à t'occuper d'inclure ces interfaces dans ton projet Vue : le compilateur TS va tout comprendre comme un grand si tu utilises NPM pour gérer tes dépendances !

Ainsi mon conseil est d'essayer TypeScript. Tous mes exemples dans ce livre l'utiliseront à partir de maintenant, car Vue et tout l'outillage autour sont vraiment conçus pour en tirer parti.

Chapter 6. TypeScript avancé

Si tu commences juste ton apprentissage de TypeScript, tu peux sauter sans problème ce chapitre dans un premier temps et y revenir plus tard. Ce chapitre est là pour montrer des utilisations plus avancées de TypeScript, qui n'auront vraiment de sens que si le langage t'est déjà familier.

6.1. readonly

Tu peux utiliser le mot-clé `readonly` (lecture seule) pour marquer une propriété d'un objet ou d'une classe comme étant… en lecture seule. De cette façon, le compilateur refusera de compiler du code qui tente d'assigner une nouvelle valeur à cette propriété :

```
interface Config {
  readonly timeout: number;
}

const config: Config = { timeout: 2000 };

// `config.timeout` is now readonly and can't be reassigned
```

6.2. keyof

Le mot-clé `keyof` peut être utilisé pour un type représentant l'union de tous les noms des propriétés d'un autre type. Par exemple, si tu as une interface `PonyModel` :

```
interface PonyModel {
  name: string;
  color: string;
  speed: number;
}
```

Tu veux écrire une fonction qui renvoie la valeur d'une propriété. Voici une première implémentation naïve :

```
function getProperty(obj: any, key: string): any {
  return obj[key];
}

const pony: PonyModel = {
  name: 'Rainbow Dash',
  color: 'blue',
  speed: 45
};
const nameValue = getProperty(pony, 'name');
```

Il y a deux problèmes ici :

- tu peux donner n'importe quelle valeur au paramètre `key`, même une clé qui n'existe pas dans `PonyModel`.
- le type de retour étant `any`, nous perdons beaucoup en information de typage.

C'est ici que `keyof` peut être utile. `keyof` permet de lister toutes les clés d'un type :

```
type PonyModelKey = keyof PonyModel;
// this is the same as `'name'|'speed'|'color'`
let property: PonyModelKey = 'name'; // works
property = 'speed'; // works

// key = 'other' would not compile
```

On peut donc utiliser ce type pour rendre `getProperty` plus strictement typée, en déclarant que :

- le premier paramètre est de type `T`
- le second paramètre est de type `K`, qui est une clé de `T`

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
  return obj[key];
}

const pony: PonyModel = {
  name: 'Rainbow Dash',
  color: 'blue',
  speed: 45
};
// TypeScript infers that `nameValue` is of type `string`!
const nameValue = getProperty(pony, 'name');
```

On fait ici d'une pierre, deux coups :

- `key` peut maintenant seulement être une propriété existante de `PonyModel` ;
- le type de retour sera déduit par TypeScript (ce qui est sacrément cool !).

Maintenant voyons comment nous pouvons utiliser `keyof` pour aller encore plus loin.

6.3. Mapped type

Disons que tu veux construire un type qui a exactement les mêmes propriétés que `PonyModel`, mais tu souhaites que chaque propriété soit optionnelle. Tu peux bien sûr le définir manuellement :

```
interface PartialPonyModel {
  name?: string;
```

```
  color?: string;
  speed?: number;
}

const pony: PartialPonyModel = {
  name: 'Rainbow Dash'
};
```

Mais on peut faire quelque chose de plus générique avec un *mapped type*, un "type de transformation" :

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};

const pony: Partial<PonyModel> = {
  name: 'Rainbow Dash'
};
```

Le type `Partial` est une transformation qui applique le modificateur `?` à chaque propriété du type ! En fait, `Partial` est suffisamment fréquent pour qu'il soit inclus dans TypeScript depuis la version 2.1, et il est déclaré comme ceci dans la bibliothèque standard.

TypeScript offre également d'autres types de transformation.

6.3.1. Readonly

`Readonly` rend toutes les propriétés d'un objet `readonly` :

```
const pony: Readonly<PonyModel> = {
  name: 'Rainbow Dash',
  color: 'blue',
  speed: 45
};
// all properties are `readonly`
```

6.3.2. Pick

`Pick` t'aide à construire un type avec seulement quelques-unes des propriétés d'origine :

```
const pony: Pick<PonyModel, 'name' | 'color'> = {
  name: 'Rainbow Dash',
  color: 'blue'
};
// `pony` can't have a `speed` property
```

6.3.3. Record

`Record` t'aide à construire un type avec les mêmes propriétés et un autre type pour ces propriétés :

```
interface FormValue {
  value: string;
  valid: boolean;
}

const pony: Record<keyof PonyModel, FormValue> = {
  name: { value: 'Rainbow Dash', valid: true },
  color: { value: 'blue', valid: true },
  speed: { value: '45', valid: true }
};
```

Il y a encore d'autres types, mais ceux-ci sont les plus utiles.

6.4. Union de types et gardien de types

Les unions de types sont très pratiques. Disons que ton application a des utilisateurs connectés et des utilisateurs anonymes, et que, de temps en temps, tu dois faire une action différente selon le cas. Tu peux modéliser les entités comme ceci :

```
interface User {
  type: 'authenticated' | 'anonymous';
  name: string;
  // other fields
}

interface AuthenticatedUser extends User {
  type: 'authenticated';
  loggedSince: number;
}

interface AnonymousUser extends User {
  type: 'anonymous';
  visitingSince: number;
}

function onWebsiteSince(user: User): number {
  if (user.type === 'authenticated') {
    // this is a LoggedUser
    return (user as AuthenticatedUser).loggedSince;
  } else if (user.type === 'anonymous') {
    // this is an AnonymousUser
    return (user as AnonymousUser).visitingSince;
  }
  // TS doesn't know every possibility was covered
```

```
  // so we have to return something here
  return 0;
}
```

Je ne sais pas pour toi, mais je n'aime pas trop ces typages explicites `as …`. Peut-on faire mieux ?

La première possibilité est d'utiliser un *type guard*, un gardien de types, une fonction spéciale dont le seul but est d'aider le compilateur TypeScript.

```
function isAuthenticated(user: User): user is AuthenticatedUser {
  return user.type === 'authenticated';
}

function isAnonymous(user: User): user is AnonymousUser {
  return user.type === 'anonymous';
}

function onWebsiteSince(user: User): number {
  if (isAuthenticated(user)) {
    // this is inferred as a LoggedUser
    return user.loggedSince;
  } else if (isAnonymous(user)) {
    // this is inferred as an AnonymousUser
    return user.visitingSince;
  }
  // TS still doesn't know every possibility was covered
  // so we have to return something here
  return 0;
}
```

C'est mieux ! Mais on a toujours besoin de retourner une valeur par défaut, même si nous avons couvert tous les cas.

On peut légèrement améliorer la situation si l'on abandonne les gardiens de types et que l'on utilise une union de types à la place.

```
interface BaseUser {
  name: string;
  // other fields
}

interface AuthenticatedUser extends BaseUser {
  type: 'authenticated';
  loggedSince: number;
}

interface AnonymousUser extends BaseUser {
  type: 'anonymous';
  visitingSince: number;
```

```
}

type User = AuthenticatedUser | AnonymousUser;

function onWebsiteSince(user: User): number {
  if (user.type === 'authenticated') {
    // this is inferred as a LoggedUser
    return user.loggedSince;
  } else {
    // this is narrowed as an AnonymousUser
    // without even testing the type!
    return user.visitingSince;
  }
  // no need to return a default value
  // as TS knows that we covered every possibility!
}
```

C'est encore mieux, car TypeScript comprend automatiquement le type utilisé dans la branche `else`.

Parfois, tu sais que ce modèle va grandir dans le futur, et que d'autres cas devront être gérés. Par exemple, si tu ajoutes un `AdminUser`. Dans ce cas, on peut utiliser un `switch`. Un `switch` sur une union de types ne compilera pas si l'un des cas n'est pas géré. Donc introduire notre `AdminUser`, ou un autre type plus tard, ajouterait automatiquement des erreurs de compilation dans tous les endroits de notre code où nous devons les gérer !

```
interface AdminUser extends BaseUser {
  type: 'admin';
  adminSince: number;
}

type User = AuthenticatedUser | AnonymousUser | AdminUser;

function onWebsiteSince(user: User): number {
  switch (user.type) {
    case 'authenticated':
      return user.loggedSince;
    case 'anonymous':
      return user.visitingSince;
    case 'admin':
      // without this case, we could not even compile the code
      // as TS would complain that all possible paths are not returning a value
      return user.adminSince;
  }
}
```

J'espère que ces astuces vous aideront dans vos projets. Penchons-nous maintenant sur les Web Components.

Chapter 7. Le monde merveilleux des Web Components

Avant d'aller plus loin, j'aimerais faire une petite pause pour parler des Web Components. Vous n'avez pas besoin de connaître les Web Components pour écrire du code Vue. Mais je pense que c'est une bonne chose d'en avoir un aperçu, car en Vue certaines décisions ont été prises pour faciliter leur intégration, ou pour rendre les composants que l'on construit similaires à des Web Components. Tu es libre de sauter ce chapitre si tu ne t'intéresses pas du tout au sujet, mais je pense que tu apprendras deux-trois choses qui pourraient t'être utiles pour la suite.

7.1. Le nouveau Monde

Les composants sont un vieux rêve de développeur. Un truc que tu prendrais sur étagère et lâcherais dans ton application, et qui marcherait directement et apporterait la fonctionnalité à tes utilisateurs sans rien faire.

Mes amis, cette heure est venue.

Oui, bon, peut-être. En tout cas, on a le début d'un truc.

Ce n'est pas complètement neuf. On avait déjà la notion de composants dans le développement web depuis quelque temps, mais ils demandaient en général de lourdes dépendances comme jQuery, Dojo, Prototype, AngularJS, etc. Pas vraiment le genre de bibliothèques que tu veux absolument ajouter à ton application.

Les Web Components essaient de résoudre ce problème : avoir des composants réutilisables et encapsulés.

Ils reposent sur un ensemble de standards émergents, que les navigateurs ne supportent pas encore parfaitement. Mais, quand même, c'est un sujet intéressant, même si on ne pourra pas en bénéficier pleinement avant quelques années, ou même jamais si le concept ne décolle pas.

Ce standard émergent est défini dans trois spécifications :

- Custom elements ("éléments personnalisés")
- Shadow DOM ("DOM de l'ombre")
- Template

Note que les exemples présentés ont plus de chances de fonctionner dans un Chrome ou un Firefox récent.

7.2. Custom elements

Les éléments customs sont un nouveau standard qui permet au développeur de créer ses propres éléments du DOM, faisant de `<ns-pony></ns-pony>` un élément HTML parfaitement valide. La spécification définit comment déclarer de tels éléments, comment tu peux les faire étendre des éléments existants, comment tu peux définir ton API, etc.

Déclarer un élément custom se fait avec un simple `customElements.define` :

```
class PonyComponent extends HTMLElement {

  constructor() {
    super();
    console.log("I'm a pony!");
  }

}

customElements.define('ns-pony', PonyComponent);
```

Et ensuite l'utiliser avec :

```
<ns-pony></ns-pony>
```

Note que le nom doit contenir un tiret, pour indiquer au navigateur que c'est un élément custom.

Évidemment ton élément custom peut avoir des propriétés et des méthodes, et il aura aussi des callbacks liés au cycle de vie, pour exécuter du code quand le composant est inséré ou supprimé, ou quand l'un de ses attributs est modifié. Il peut aussi avoir son propre template. Par exemple, peut-être que ce `ns-pony` affiche une image du poney, ou seulement son nom :

```
class PonyComponent extends HTMLElement {

  constructor() {
    super();
    console.log("I'm a pony!");
  }


  /**
   * This is called when the component is inserted
   */
  connectedCallback() {
    this.innerHTML = '<h1>General Soda</h1>';
  }

}
```

Si tu jettes un coup d'œil au DOM, tu verras `<ns-pony><h1>General Soda</h1></ns-pony>`. Mais cela veut dire que le CSS ou la logique JavaScript de ton application peut avoir des effets indésirables sur ton composant. Donc, en général, le template est caché et encapsulé dans un truc appelé le Shadow DOM ("DOM de l'ombre"), et tu ne verras dans le DOM que `<ns-pony></ns-pony>`, bien que le navigateur affiche le nom du poney.

7.3. Shadow DOM

Avec un nom qui claque comme celui-là, on s'attend à un truc très puissant. Et il l'est. Le Shadow DOM est une façon d'encapsuler le DOM de ton composant. Cette encapsulation signifie que la feuille de style et la logique JavaScript de ton application ne vont pas s'appliquer sur le composant et le ruiner accidentellement. Cela en fait l'outil idéal pour dissimuler le fonctionnement interne de ton composant, et s'assurer que rien n'en fuit à l'extérieur.

Si on retourne à notre exemple précédent :

```
class PonyComponent extends HTMLElement {

  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    const title = document.createElement('h1');
    title.textContent = 'General Soda';
    shadow.appendChild(title);
  }

}
```

Si tu essaies maintenant de l'observer, tu devrais voir :

```
<ns-pony>
  #shadow-root (open)
    <h1>General Soda</h1>
</ns-pony>
```

Désormais, même si tu ajoutes du style aux éléments `h1`, rien ne changera : le Shadow DOM agit comme une barrière.

Jusqu'à présent, nous avions utilisé une chaîne de caractères pour notre template. Mais ce n'est habituellement pas la manière de procéder. La bonne pratique est de plutôt utiliser l'élément `<template>`.

7.4. Template

Un template spécifié dans un élément `<template>` n'est pas affiché par le navigateur. Son but est d'être à terme cloné dans un autre élément. Ce que tu déclareras à l'intérieur sera inerte : les scripts ne s'exécuteront pas, les images ne se chargeront pas, etc. Son contenu peut être requêté par le reste de la page avec la méthode classique `getElementById()`, et il peut être placé sans risque n'importe où dans la page.

Pour utiliser un template, il doit être cloné :

```
<template id="pony-template">
```

```
  <style>
    h1 { color: orange; }
  </style>
  <h1>General Soda</h1>
</template>
```

```
class PonyComponent extends HTMLElement {

  constructor() {
    super();
    const template = document.querySelector('#pony-template');
    const clonedTemplate = document.importNode(template.content, true);
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.appendChild(clonedTemplate);
  }

}
```

7.5. Les bibliothèques basées sur les Web Components

Toutes ces spécifications constituent les Web Components. Je suis loin d'en être expert, et ils présentent toute sorte de pièges.

Comme les Web Components ne sont pas complètement supportés par tous les navigateurs, il y a un *polyfill* à inclure dans ton application pour être sûr que ça fonctionne. Ce *polyfill* est appelé web-component.js, et il est bon de noter qu'il est le fruit d'un effort commun entre Google, Mozilla et Microsoft, entre autres.

Au-dessus de ce *polyfill*, quelques bibliothèques ont vu le jour. Elles proposent toutes de faciliter le travail avec les Web Components, et viennent souvent avec un lot de composants tout prêts.

Parmi les initiatives notables, on peut citer :

- Polymer, première tentative de la part de Google ;
- LitElement, projet plus récent de l'équipe Polymer ;
- X-tag de Mozilla et Microsoft ;
- Stencil.

Je ne vais pas rentrer dans les détails, mais tu peux facilement utiliser un composant existant. Supposons que tu veuilles embarquer une carte Google dans ton application :

```
<!-- Polyfill Web Components support for older browsers -->
<script src="webcomponents.js"></script>

<!-- Import element -->
<script src="google-map.js"></script>
```

```
<!-- Use element -->
<body>
  <google-map latitude="45.780" longitude="4.842"></google-map>
</body>
```

Il y a une tonne de composants disponibles. Tu peux en avoir un aperçu sur https://www.webcomponents.org/.

Tu peux faire plein de trucs cools avec LitElement et les autres frameworks similaires, comme du binding bidirectionnel, donner des valeurs par défaut aux attributs, émettre des événements custom, réagir aux modifications d'attribut, répéter des éléments si tu fournis une collection à un composant, etc.

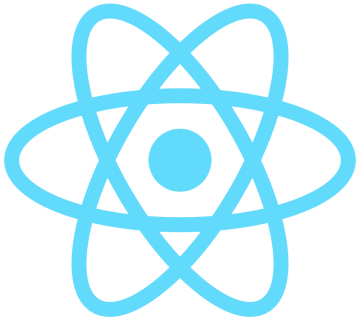
C'est un chapitre trop court pour te montrer sérieusement tout ce que l'on peut faire avec les Web Components, mais tu verras que certains de leurs concepts vont émerger dans la lecture à venir. Et tu verras sans aucun doute que Vue a été conçu pour rendre facile l'utilisation des Web Components aux côtés de nos composants Vue. Il est même possible d'exporter nos composants Vue sous forme de Web Components.

Chapter 8. La philosophie de Vue

Pour construire une application Vue, il te faut saisir quelques trucs sur la philosophie du framework.

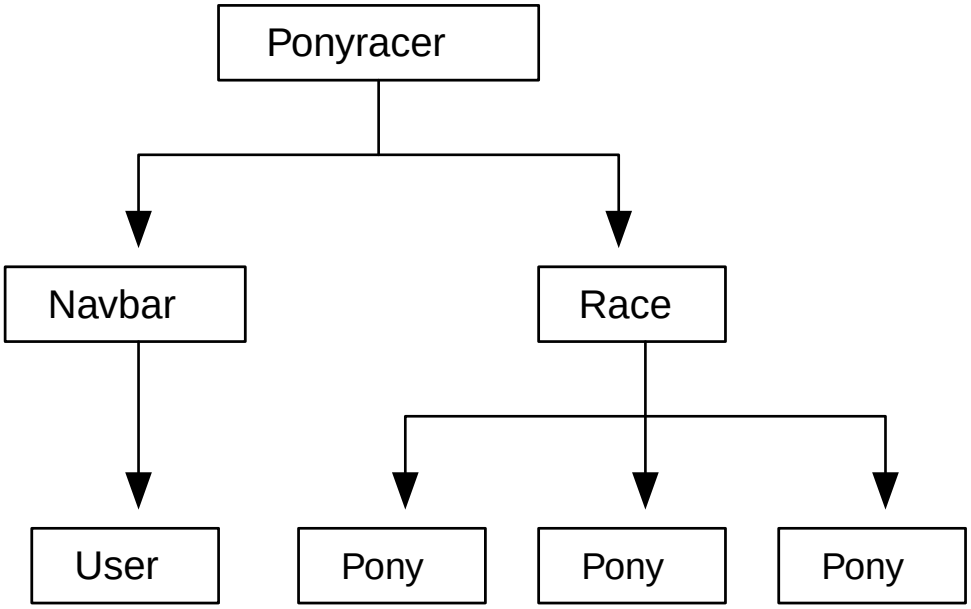
Avant tout, Vue est un framework orienté composant. Tu vas écrire de petits composants et, assemblés, ils vont constituer une application complète. Un composant est un groupe d'éléments HTML, dans un template, dédiés à une tâche particulière. Pour cela, tu auras probablement besoin d'un peu de logique métier derrière ce template, pour peupler les données et réagir aux événements par exemple.

Cette orientation composant est largement partagée par de nombreux frameworks front-end : c'est le cas depuis le début de React, le framework tendance de Facebook ; Ember et AngularJS ont leur propre façon de faire quelque chose de similaire ; et les petits nouveaux Svelte ou Angular parient aussi sur la construction de petits composants.



Vue n'est donc pas le seul sur le sujet, mais il est parmi les premiers à considérer sérieusement l'intégration des Web Components (ceux du standard officiel). Mais écartons ce sujet, trop avancé pour le moment.

Tes composants seront organisés de manière hiérarchique, comme le DOM : un composant racine aura des composants enfants, qui auront chacun des composants enfants, etc. Si tu veux afficher une course de poneys (qui ne voudrait pas ?), tu auras probablement une application (`Ponyracer`), affichant un menu (`Navbar`) avec l'utilisateur connecté (`User`) et une vue enfant (`Race`), affichant, évidemment, les poneys (`Pony`) en course :



Comme tu vas écrire des composants tous les jours (de la semaine au moins), regardons de plus près à quoi ça ressemble. L'équipe Vue voulait aussi bénéficier d'une autre pépite du développement web moderne : ES2015+. Mais pour avoir la meilleure expérience possible, il est aussi possible d'écrire nos applications en TypeScript. J'espère que tu es au courant, parce que je viens de consacrer deux chapitres à ces sujets !

Vue a la particularité d'offrir la possibilité d'écrire tout son composant dans le même fichier, contenant alors à la fois la partie affichage en HTML et la partie comportement en TypeScript. On peut également y joindre la partie style en CSS, ou SCSS, etc. De tels composants sont appelés des *Single File Components* (SFC pour leur petit nom). L'extension du fichier est alors `.vue`. C'est un peu original comparé à ce que font les autres frameworks, mais on s'y fait rapidement.

Par exemple, en simplifiant, le composant `Race` pourrait ressembler à ça :

```
<template>
  <div>
    <h1>{{ race.name }}</h1>
    <ul v-for="pony in race.ponies">
      <li>{{ pony.name }}</li>
    </ul>
  </div>
</template>

<script lang="ts">
import { defineComponent } from 'vue';

export default defineComponent({
  name: 'Race',
```

```
  setup() {
    return {
      race: {
        name: 'Buenos Aires',
        ponies: [{ name: 'Rainbow Dash' }, { name: 'Pinkie Pie' }]
      }
    };
  }
});
</script>
```

Si tu connais déjà un autre langage de templating, le template t'est peut-être familier, avec les expressions entre accolades `{{ }}`, qui seront évaluées et remplacées par les valeurs correspondantes. Je ne veux pas aller trop loin pour le moment, juste te donner un aperçu du code. On se penchera évidemment en détail sur les composants et templates dans les prochains chapitres.

Un composant est une partie complètement isolée de ton application. Ton application *est* un composant comme les autres.

Tu pourras aussi prendre des composants sur étagère fournis par la communauté, et les ajouter simplement dans ton application pour bénéficier de leurs fonctionnalités. Souvent, ces composants ou fonctionnalités supplémentaires sont rassemblés dans un plugin Vue.

De tels plugins fournissent des composants d'IHM, ou la gestion du glisser-déposer, ou des validations spécifiques pour tes formulaires, et tout ce que tu peux imaginer d'autre. On verra ensemble quels plugins peuvent être utiles.

Dans les chapitres suivants, on explorera quoi mettre en place, comment construire un petit composant, ton premier module et la syntaxe des templates.

On se penchera également sur les tests d'une application Vue. J'adore faire des tests et voir la barre de progression devenir entièrement verte dans mon IDE. Ça me donne l'impression de faire du bon boulot. Il y aura ainsi un chapitre entier consacré à tout tester : tes composants, tes services, ton interface…

Vue a ce côté un peu magique, où les modifications sont automatiquement détectées par le framework et appliquées au modèle et à la vue. Chaque framework a sa propre technique : on étudiera évidemment tout ça pour Vue, et on essaiera d'éclairer les concepts de proxies, de DOM virtuel et autres termes de magie noire derrière tout ça (pas d'inquiétude, ce n'est pas si compliqué).

Vue est aussi un écosystème complet, avec plein d'outils pour faciliter les tâches classiques du développement web. Construire des formulaires, appeler un serveur HTTP, configurer du routage d'URL, construire des animations, tout ce que tu veux : c'est possible !

Voilà, ça fait pas mal de trucs à apprendre ! Alors commençons par le commencement : initialiser une application et construire notre premier composant.

Chapter 9. Commencer de zéro

Maintenant que nous en savons un peu plus sur ECMAScript, TypeScript et la philosophie de Vue, mettons les mains dans le cambouis et démarrons une nouvelle application.

9.1. Un framework évolutif

Vue s'est toujours présenté comme un framework évolutif qui, au contraire d'alternatives comme Angular ou React, peut être adopté progressivement. On peut parfaitement prendre une page HTML statique, ou une application basée sur jQuery, et y ajouter un peu de Vue.

Donc, pour commencer, j'aimerais montrer comme c'est simple de mettre en place Vue dans une page HTML.

Créons une page HTML vide `index.html` :

index.html

```
<html lang="en">
  <meta charset="UTF-8" />
  <head>
    <title>Vue - the progressive framework</title>
  </head>
  <body>
  </body>
</html>
```

Ajoutons-y un peu de HTML que Vue devra gérer :

index.html

```
<html lang="en">
  <meta charset="UTF-8" />
  <head>
    <title>Vue - the progressive framework</title>
  </head>
  <body>
    <div id="app">
      <h1>Hello {{ user }}</h1>
    </div>
  </body>
</html>
```

Les accolades autour de `user` font partie de la syntaxe de Vue. Elles indiquent que `user` doit être remplacé par sa valeur. Nous expliquerons tout cela en détails dans le prochain chapitre, pas d'inquiétude.

Si tu ouvres cette page dans ton navigateur, tu verras qu'elle affiche `Hello {{ user }}`. C'est

normal : nous n'avons pas encore utilisé Vue.

Faisons-le. Vue est publié sur NPM et il existe des sites (appelés *CDNs*, pour *Content Delivery Network*) qui hébergent les packages NPM et permettent ainsi de les inclure dans nos pages HTML. Unpkg est l'un d'entre eux, et on peut donc l'utiliser pour ajouter Vue à notre page. Bien sûr, tu pourrais aussi choisir de télécharger le fichier et de l'héberger toi-même.

index.html

```
<html lang="en">
  <meta charset="UTF-8" />
  <head>
    <title>Vue - the progressive framework</title>
    <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
  </head>
  <body>
    <div id="app">
      <h1>Hello {{ user }}</h1>
    </div>
  </body>
</html>
```

Cet exemple utilise la dernière version de Vue. Tu peux spécifier n'importe quelle version explicitement en ajoutant `@version` dans l'URL, après `https://unpkg.com/vue`. Le livre que tu es en train de lire utilise `vue@3.5.13`

Si tu recharges la page, tu verras que Vue affiche un avertissement dans la console, nous informant que l'on utilise une version dédiée au développement. Tu peux utiliser `vue.global.prod.js` pour utiliser la version de production et faire disparaître l'avertissement. La version de production désactive toutes les validations sur le code, est minifiée, et est donc plus rapide et plus légère.

Il nous faut à présent créer notre application, avec la fonction `createApp`. Mais cette fonction a besoin d'un composant racine.

Pour créer ce composant, il nous suffit de créer un objet qui le définit. Cet objet peut avoir de nombreuses propriétés, mais pour l'instant, nous allons seulement y ajouter une fonction `setup`. Pas de panique, nous reviendrons en détails sur la définition d'un composant et sur cette fonction. Mais son nom est assez explicite : elle prépare le composant et Vue appellera cette fonction lorsque le composant sera initialisé.

index.html

```
<html lang="en">
  <meta charset="UTF-8" />
  <head>
    <title>Vue - the progressive framework</title>
    <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
  </head>
  <body>
    <div id="app">
```

```
      <h1>Hello {{ user }}</h1>
    </div>
    <script>
      const RootComponent = {
        setup() {
          return { user: 'Cédric' };
        }
      };
    </script>
  </body>
</html>
```

La fonction `setup` ne fait que retourner un objet avec une propriété `user` et une valeur pour cette propriété. Si tu recharges la page, toujours pas de changement : il nous reste toujours à appeler `createApp` avec notre composant racine.

Nous utilisons du JavaScript moderne dans cet exemple, et il te faut donc utiliser un navigateur suffisamment récent, qui supporte cette syntaxe.

index.html

```
<html lang="en">
  <meta charset="UTF-8" />
  <head>
    <title>Vue - the progressive framework</title>
    <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
  </head>
  <body>
    <div id="app">
      <h1>Hello {{ user }}</h1>
    </div>
    <script>
      const RootComponent = {
        setup() {
          return { user: 'Cédric' };
        }
      };
      const app = Vue.createApp(RootComponent);
      app.mount('#app');
    </script>
  </body>
</html>
```

`createApp` crée une application qui doit être "montée", c'est-à-dire attachée à un élément du DOM. Nous utilisons ici la `div` avec l'identifiant `app`. Si tu recharges la page, tu devrais voir `Hello Cédric` s'afficher. Bravo, tu viens de créer ta première application Vue.

Peut-être pourrions-nous ajouter un autre composant ?

Créons un composant qui affiche le nombre de messages non lus. Il nous faut donc un nouvel objet `UnreadMessagesComponent`, avec une fonction `setup` similaire :

index.html

```
<html lang="en">
  <meta charset="UTF-8" />
  <head>
    <title>Vue - the progressive framework</title>
    <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
  </head>
  <body>
    <div id="app">
      <h1>Hello {{ user }}</h1>
    </div>
    <script>
      const UnreadMessagesComponent = {
        setup() {
          return { unreadMessagesCount: 4 };
        }
      };
      const RootComponent = {
        setup() {
          return { user: 'Cédric' };
        }
      };
      const app = Vue.createApp(RootComponent);
      app.mount('#app');
    </script>
  </body>
</html>
```

Cette fois, au contraire du composant racine dont la vue est directement définie par la `div #app`, nous voudrions définir un template pour le composant `UnreadMessagesComponent`. Il suffit pour cela de définir un élément `script` avec le type `text/x-template`. Ce type garantit que le navigateur ignorera simplement le contenu du script. On peut ensuite référencer le template par son identifiant dans la définition du composant.

index.html

```
<html lang="en">
  <meta charset="UTF-8" />
  <head>
    <title>Vue - the progressive framework</title>
    <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
    <script type="text/x-template" id="unread-messages-template">
      <div>You have {{ unreadMessagesCount }} messages</div>
    </script>
  </head>
  <body>
```

```
    <div id="app">
      <h1>Hello {{ user }}</h1>
    </div>
    <script>
      const UnreadMessagesComponent = {
        template: '#unread-messages-template',
        setup() {
          return { unreadMessagesCount: 4 };
        }
      };
      const RootComponent = {
        setup() {
          return { user: 'Cédric' };
        }
      };
      const app = Vue.createApp(RootComponent);
      app.mount('#app');
    </script>
  </body>
</html>
```

On veut pouvoir insérer ce nouveau composant à l'intérieur du composant racine. Pour pouvoir faire ça, nous devons autoriser le composant racine à utiliser le composant *unread messages*, et lui assigner un nom en *PascalCase*.

index.html

```
<html lang="en">
  <meta charset="UTF-8" />
  <head>
    <title>Vue - the progressive framework</title>
    <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
    <script type="text/x-template" id="unread-messages-template">
      <div>You have {{ unreadMessagesCount }} messages</div>
    </script>
  </head>
  <body>
    <div id="app">
      <h1>Hello {{ user }}</h1>
    </div>
    <script>
      const UnreadMessagesComponent = {
        template: '#unread-messages-template',
        setup() {
          return { unreadMessagesCount: 4 };
        }
      };
      const RootComponent = {
        components: {
          UnreadMessages: UnreadMessagesComponent
```

```
        },
        setup() {
          return { user: 'Cédric' };
        }
      };
      const app = Vue.createApp(RootComponent);
      app.mount('#app');
    </script>
  </body>
</html>
```

On peut ensuite utiliser `<unread-messages></unread-messages>` (qui est la version *dash-case* de `UnreadMessages`) pour insérer le composant où on le veut.

index.html

```
<html lang="en">
  <meta charset="UTF-8" />
  <head>
    <title>Vue - the progressive framework</title>
    <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
    <script type="text/x-template" id="unread-messages-template">
      <div>You have {{ unreadMessagesCount }} messages</div>
    </script>
  </head>
  <body>
    <div id="app">
      <h1>Hello {{ user }}</h1>
      <unread-messages></unread-messages>
    </div>
    <script>
      const UnreadMessagesComponent = {
        template: '#unread-messages-template',
        setup() {
          return { unreadMessagesCount: 4 };
        }
      };
      const RootComponent = {
        components: {
          UnreadMessages: UnreadMessagesComponent
        },
        setup() {
          return { user: 'Cédric' };
        }
      };
      const app = Vue.createApp(RootComponent);
      app.mount('#app');
    </script>
  </body>
</html>
```

En comparaison des autres frameworks, une application Vue est extrêmement simple à mettre en œuvre : juste du pur JavaScript et HTML. Pas d'outillage nécessaire. Les composants sont de simples objets. Même un développeur qui ne connaît pas Vue peut sans doute comprendre ce qui se passe. Et c'est l'une des forces du framework : c'est facile de démarrer, facile à comprendre et les fonctionnalités peuvent être apprises progressivement.

On *pourrait* se contenter de cette approche minimale, mais soyons réalistes : ça ne tiendra pas très longtemps. Trop de composants vont devoir être définis dans le même fichier. On voudrait aussi pouvoir utiliser TypeScript au lieu de JavaScript, ajouter des tests, de l'analyse de code, etc.

Nous *pourrions* installer et configurer toute une série d'outils nous-mêmes. Mais profitons plutôt du travail de la communauté et utilisons Vue CLI (qui a été le standard pendant de nombreuses années) ou l'outil maintenant recommandé, Vite.

9.2. Vue CLI

La CLI est désormais en mode "maintenance", c'est-à-dire qu'elle ne reçoit plus de nouveautés. L'outil recommandé est maintenant Vite, que nous présentons dans la section suivante. Comme beaucoup de projets existants utilisent la CLI, nous pensons que cela vaut encore le coup de la présenter, et cela aide à comprendre les différences avec Vite.

La Vue CLI (*Command Line Interface*) est née pour simplifier le développement d'applications Vue. Elle permet de créer le squelette de l'application, et ensuite de la construire. Et elle offre un vaste écosystème de plugins. Chaque plugin ajoute une fonctionnalité spécifique, comme le support pour les tests unitaires, ou le linting, ou le support de TypeScript. La CLI a même une interface graphique !

L'une des caractéristiques de Vue CLI est de permettre d'écrire chaque composant dans un fichier unique avec l'extension `.vue`. Dans ce fichier, toutes les parties d'un composant sont définies : sa définition en JavaScript/TypeScript, son template HTML, et même ses styles CSS. Un tel fichier est appelé *Single File Component*, ou SFC.

Mais l'intérêt principal de la CLI est d'éviter d'avoir à apprendre et à configurer tous les outils qu'elle utilise (Node.js, NPM, Webpack, TypeScript, etc.), tout en restant flexible et configurable.

Mais la CLI est désormais en mode maintenance, et Vite est l'alternative recommandée. Explorons donc pourquoi.

9.3. Bundlers : Webpack, Rollup, esbuild

Quand on écrit des applications modernes en JavaScript/TypeScript, on a souvent besoin d'un outil qui *bundle* (rassemble) tous les assets (le code, les styles, les images, les polices de caractères).

Pendant longtemps, Webpack a été le favori indiscutable. Webpack vient avec une fonctionnalité simple, mais très pratique : il comprend tous les types de module JavaScript qui existent (les modules ECMAScript modernes, mais aussi les modules AMD et CommonJS, des formats qui existaient avant le standard). Cette compréhension rend facile d'utilisation n'importe quelle

bibliothèque que tu trouves sur Internet (le plus souvent sur NPM) : il y a juste besoin de l'installer, de l'importer dans un de tes fichiers, et Webpack s'occupe du reste. Même si tu utilises des bibliothèques avec des formats très différents, Webpack va joyeusement les convertir et packager tout ton code et le code de ces bibliothèques ensemble dans un seul gros fichier JS : le fameux *bundle*.

C'est une tâche très importante, parce que même si le standard a défini les modules ECMAScript en 2015, la plupart des navigateurs ne les supportent que depuis peu !

L'autre tâche de Webpack est aussi de t'aider pendant le développement, en fournissant un serveur de développement et en surveillant ton projet (il peut même faire du HMR, *Hot Module Reloading*, c'est-à-dire du rechargement de module à chaud). Quand quelque chose change, Webpack lit le point d'entrée de l'application (`main.ts` par exemple), puis il lit les imports et charge ces fichiers, puis il lit les imports de ces fichiers importés et les charge, et ainsi de suite récursivement. Tu vois l'idée ! Quand tout est chargé, il remet tout cela dans un grand fichier, contenant à la fois ton code et les bibliothèques importées depuis `node_modules`, en changeant le format des modules si besoin. Le navigateur recharge alors tout ce fichier pour afficher les changements . Toute cette boucle peut prendre un certain temps quand on travaille sur des gros projets avec des centaines, voire des milliers de fichiers, même si Webpack vient avec un système de caches et d'optimisations pour être le plus rapide possible.

La CLI Vue (comme beaucoup d'autres outils) utilise Webpack pour la majeure partie de son travail, aussi bien quand on construit l'application avec `npm run build`, que quand on lance le serveur de développement avec `npm run serve`.

Ce qui est chouette, c'est que l'écosystème Webpack est extraordinairement riche en *plugins* et *loaders* : tu peux donc faire pratiquement ce que tu veux, même les trucs les plus improbables. D'un autre côté, une configuration Webpack peut rapidement devenir assez difficile à comprendre avec toutes ces options.

Si je parle de Webpack et ce que font les bundlers, c'est parce que de sérieuses alternatives ont émergé ces derniers temps, et il peut être assez difficile de saisir ce qu'elles font et quelles sont leurs différences. Pour être honnête, je ne suis pas sûr de comprendre tous les détails moi-même, et j'ai pourtant pas mal contribué aux CLIs Vue et Angular, toutes deux utilisant massivement Webpack ! Mais je tente quand même une explication.

Une alternative sérieuse est Rollup. Rollup entend faire les choses de façon plus simple que Webpack, en en faisant moins par défaut, mais souvent plus vite que Webpack. Son auteur est Rich Harris, qui est aussi l'auteur du framework Svelte. Rich a écrit un article assez populaire appelé "Webpack et Rollup : les mêmes, mais différents". Sa conclusion est "Utilise Webpack pour les applications, et Rollup pour les bibliothèques". En fait, Rollup peut faire quasiment tout ce que fait Webpack pour les builds de production, mais il ne vient pas avec un serveur de développement qui pourrait surveiller tes fichiers pendant que tu travailles.

Une autre super alternative est esbuild. À la différence de Webpack et Rollup, esbuild n'est pas lui-même écrit en JavaScript. Il est écrit en Go et compilé en code natif. Il a aussi été conçu avec le parallélisme en tête. Cela le rend bien plus rapide que Webpack et Rollup. Genre 10 à 100 fois plus rapide .

Pourquoi ne pas utiliser esbuild plutôt que Webpack alors ? C'est exactement ce qu'Evan You, l'auteur de Vue, a pensé quand il développait Vue 3. Il a eu une autre brillante idée. En 2018, Firefox a officiellement supporté les Modules ECMAScript natifs (souvent appelés "ESM natifs"). En 2019, ce fut au tour de Node.js, et des autres navigateurs principaux. De nos jours, ton navigateur personnel peut probablement comprendre les ESM natifs sans problème. Evan a imaginé un outil qui servirait les fichiers au navigateur au format ESM, laissant le gros du travail à esbuild quand il faut transformer les fichiers sources en fichier ESM si besoin (par exemple pour les fichiers TypeScript ou Vue, ou pour des modules dans un format plus ancien).

Vite (encore un mot français) était né.

9.4. Vite

L'idée derrière Vite est que, comme les navigateurs modernes supportent les modules ES, on peut maintenant les utiliser directement, au moins pendant le développement, plutôt que de générer un bundle.

Donc lorsque tu charges une page dans un navigateur quand tu développes avec Vite, tu ne charges pas un seul gros fichier JS contenant toute l'application : tu charges juste les quelques ESM nécessaires pour cette page, chacun dans leur propre fichier (et chacun dans leur propre requête HTTP). Si un ESM a des imports, alors le navigateur demande à Vite ces fichiers également.

Vite est donc principalement un serveur de développement, chargé de répondre aux requêtes du navigateur, en lui envoyant les ESM demandés. Comme on a peut-être écrit notre code en TypeScript, ou utilisé un SFC avec une extension `.vue` (voir plus loin), Vite doit parfois transformer ces fichiers sur notre disque en un ESM que le navigateur peut comprendre. C'est ici qu'esbuild intervient. Vite est bâti au-dessus d'esbuild et quand un fichier demandé a besoin d'être transformé, Vite demande à celui-ci de s'en occuper et envoie ensuite le résultat au navigateur. Si tu changes quelque chose dans un fichier alors Vite n'envoie que le module qui a changé au navigateur, au lieu d'avoir à reconstruire toute l'application comme le font les outils basés sur Webpack !

Vite utilise aussi esbuild pour optimiser certaines choses. Par exemple si tu utilises une bibliothèque avec des tonnes de fichiers, Vite "pré-bundle" cette bibliothèque en un seul fichier grâce à esbuild et l'envoie au navigateur en une seule requête plutôt que quelques dizaines/centaines. Cette tâche est faite une seule fois au démarrage du serveur, tu n'as donc pas à payer le coût à chaque fois que tu rafraîchis la page.

Le truc marrant est que Vite n'est pas vraiment lié à Vue : il peut être utilisé avec Svelte, React ou autre. En fait, certains frameworks recommandent même son utilisation ! Svelte, de Rich Harris, a été l'un des premiers à sauter le pas et recommande maintenant officiellement Vite.

esbuild est très fort pour la partie JS, mais il n'est pas (encore) capable de découper l'application en plusieurs morceaux, ou de gérer entièrement les CSS (alors que Webpack et Rollup le font par défaut). Il n'est donc pas adapté pour packager l'application pour la prod. C'est là que Rollup entre en jeu : Vite utilise esbuild pendant le développement, mais utilise Rollup pour le build de prod. Peut-être que dans le futur, Vite utilisera esbuild pour tout.

Vite est cependant plus qu'une simple enveloppe autour d'esbuild. Comme on l'a vu, esbuild

transforme les fichiers très vite. Mais Vite ne lui demande cependant pas de faire ce travail à chaque fois qu'une page est rechargée : il utilise le cache du navigateur pour effectuer le moins de travail possible. Ainsi si tu affiches une page que tu as déjà chargé, elle sera affichée instantanément. Vite vient aussi avec plein d'autres fonctionnalités et un riche ensemble de plugins.

Une note importante : esbuild transpile TypeScript en JavaScript, mais il ne le compile pas : esbuild ignore complètement les types ! Ça le rend hyper rapide, mais cela veut donc dire que tu n'auras pas de vérification des types de la part de Vite pendant le développement. Pour vérifier que ton application compile, tu devras exécuter Volar (`vue-tsc`), généralement quand tu construis l'application.

Alors, tu as envie d'essayer ? Parce que moi oui !

Vite propose des exemples de projets pour React, Svelte et Vue, mais l'équipe Vue a également lancé un petit projet basé sur Vite appelé `create-vue`. Et ce projet est maintenant la façon recommandée de démarrer un nouveau projet Vue 3.

9.5. create-vue

create-vue est donc bâti au-dessus de Vite et fournit des squelettes de projets Vue 3.

Pour démarrer, tu lances simplement :

```
npm create vue@3
```

La commande `npm create quelquechose` télécharge et exécute en fait le package `create-quelquechose`. Donc ici `npm create vue` exécute le package `create-vue`.

Tu peux alors choisir :

- un nom de projet
- si tu veux TypeScript ou non
- si tu veux JSX ou non
- si tu veux Vue router ou non
- si tu veux Pinia (gestion de l'état) ou non
- si tu veux Vitest pour les tests unitaires ou non
- si tu veux Cypress pour les tests e2e ou non
- si tu veux ESLint/Prettier pour le lint et le formatage ou non

et ton projet est prêt !

Nous allons bien sûr explorer ces différentes technologies tout au long du livre.

Tu veux t'y mettre ?

Pour créer ta première application avec Vite, suis les instructions de l'exercice Getting Started 🐴 ! Il fait partie de notre Pack Pro, mais est accessible à tous nos lecteurs.

Terminé ?

Si tu as bien suivi les instructions (et obtenu un score de 100 % j'espère !), tu as à présent une application en place, qui fonctionne. Examinons quelques-uns de ses fichiers. Le point d'entrée de l'application est le fichier `main.ts` :

main.ts

```
import { createApp } from 'vue';
import App from './App.vue';

createApp(App).mount('#app');
```

Il monte un composant `App` défini dans `App.vue`, qui ressemble à cela lorsqu'il est créé :

App.vue

```
<script setup lang="ts"></script>

<template>
  <h1>You did it!</h1>
</template>

<style scoped></style>
```

`App.vue` est un *Single File Component*. Un quoi ?

9.6. Single File Components

Un *Single File Component* (SFC) est un composant dont toutes les parties sont définies dans un seul fichier.

Il définit :

- le template à l'intérieur de l'élément `template` ;
- la définition du composant dans l'élément `script`. Note la présence de l'attribut `lang="ts"` indiquant que l'on utilise TypeScript ;
- les styles CSS du composant dans l'élément `style`.

L'outillage compile le code TypeScript en JavaScript pour nous. Il compile aussi le template en JavaScript (nous expliquerons dans un prochain chapitre en quoi ça consiste). Le compilateur Vue, au contraire du navigateur, accepte et comprend les éléments en *PascalCase*, donc on peut utiliser `<hello-world>` ou `<HelloWorld>` pour le composant fils. Nous utiliserons la version *PascalCase* à partir de maintenant.

Comme nous l'avons vu dans l'exercice, Vite peut lancer les tests unitaires, les tests end-to-end, le linter, etc. Chaque exercice vient avec ses tests unitaires et end-to-end déjà écrits, ce qui te permettra de vérifier à chaque étape que ton code est correct. Et, comme Vite est extrêmement rapide, l'expérience de développement est super agréable .

Maintenant que nous avons appris à nous servir de l'outillage, et que nous sommes prêts à écrire plus de composants, passons à la syntaxe des templates.

Chapter 10. Fin de l’extrait gratuit

Voilà, j’espère que ce que tu as lu t’aura comblé. Si tu rêves désormais de lire la suite (tu devrais !), va l’acheter sur le site du livre ! :)

Annexe A: Historique des versions

Voici ci-dessous les changements que nous avons apportés à cet ebook depuis sa première version. C'est en anglais, mais ça devrait t'aider à voir les nouveautés depuis ta dernière lecture !

N'oublie pas qu'acheter cet ebook te donne droit à toutes ses mises à jour ultérieures, gratuitement. Rends-toi sur la page https://books.ninja-squad.com/claim pour obtenir la toute dernière version.

Current versions:

- Vue: `3.5.13`

A.1. Changes since last release - 2025-04-13

From zero to something

- Use the new `--bare` option of create-vue v3.14. (2025-02-07)

Script setup

- Use props destructuration in more examples (2025-02-19)

State Management

- Remove Vuex section (2025-04-13)

Advanced component patterns

- useTemplateRef is automatically infered. (2024-10-01)

A.2. v3.5.1 - 2024-09-05

The templating syntax

- Add the shorter `v-bind` syntax introduced in Vue v3.4. (2024-01-11)

How to build components

- Add a section about how to pause/resume watchers as introduced in Vue v3.5. (2024-09-05)
- Add a section about `useTemplateRef` as introduced in Vue v3.5. (2024-09-05)

Script setup

- Use props destructuration as it is now stable in Vue v3.5 (2024-09-05)

Slots

- Add a section about `v-slot` destructuration. (2024-07-26)

A.3. v3.4.0 - 2023-12-29

How to build components

- Add an example of a `validator` using other props, as introduced in Vue v3.4. (2023-12-29)

Forms

- Update the `defineModel` section with the new features from Vue v3.4. (2023-12-29)

A.4. v3.3.0 - 2023-05-12

The many ways to define components

- The "sugar ref" syntax has been removed as it is now deprecated as of Vue v3.3. (2023-05-11)

Script setup

- Use the shorter `defineEmits` syntax introduced in Vue v3.3. (2023-05-11)
- Add a section about the `defineOptions` macro introduced in Vue v3.3. (2023-05-11)

Forms

- Add a section about the `defineModel` macro introduced in Vue v3.3. (2023-05-12)
- Add a section about how to build custom form components. (2023-05-12)

Slots

- Add a section about the `defineSlots` macro introduced in Vue v3.3. (2023-05-11)

A.5. v3.2.45 - 2023-01-05

Global

- Reorder the chapters so that the composition API chapter is before the script setup one (as in the Pro Pack exercises). (2022-09-01)

Style your components

- Add a section about `v-bind` in CSS (2022-07-07)

Router

- Add a section about the route meta field and its usage with guards (2022-12-01)

Advanced component patterns

- New chapter about advanced component patterns! First sections are about template and component refs. (2022-09-02)

Custom directives

- New chapter about custom directives! (2023-01-05)

A.6. v3.2.37 - 2022-07-06

State Management

- Add some details about Pinia (SSR, plugins, HMR, etc.), and add a section about "Why use a store" (2022-03-11)

Internationalization

- New chapter about vue-i18n! (2022-07-06)

A.7. v3.2.30 - 2022-02-10

From zero to something

- The getting started section now uses Vite and create-vue! (2022-02-10)

Style your components

- Explain the differences between Vite and the CLI for styles handling (2022-02-10)

Testing your app

- Section about Vitest and the differences with Jest (2022-02-10)

Lazy-loading

- Add a section about lazy-loading with Vite (2022-02-10)

Performances

- Mention Rollup and Vite (2022-02-10)

A.8. v3.2.26 - 2021-12-17

The templating syntax

- Section about Templates and TypeScript support in Vue 3.2 (2021-10-01)

Script setup

- Section about `defineProps` destructuration and default value feature, introduced in Vue 3.2.20 (2021-12-01)

Composition API

- Section about the awesome VueUse library (2021-10-01)

State Management

- As Pinia is the new official recommandation for state management library in Vue 3 (instead of Vuex), the chapter now goes deeper into the details of how to use Pinia, and how to test it. (2021-12-17)

Router

- Section on how to use `vue-router-mock` for tests (2021-10-01)

A.9. v3.2.19 - 2021-09-30

The many ways to define components

- Update to sugar ref RFC take 2 (2021-08-30)

Script setup

- New chapter about the `script setup` syntax! All examples of the ebook and exercises have been migrated to this new recommended syntax, introduced in Vue 3.2. (2021-09-29)

Suspense

- Section about `script setup` and `await` (2021-09-29)

A.10. v3.2.0 - 2021-08-10

Global

- Add links to our quizzes! (2021-07-29)

The many ways to define components

- New chapter about the various ways to define a component in Vue 3 (2021-08-10)

Performances

- New chapter! Includes a section about the new `v-memo` directive introduced in Vue 3.2. (2021-08-10)

A.11. v3.1.0 - 2021-06-07

The templating syntax

- Mention the projects that can be used to have template type-checking at compile time. The ebook now uses Volar to check the examples. (2021-05-05)

Forms

- VeeValidate v4.3.0 introduced a new `url` validator. (2021-05-05)

A.12. v3.0.11 - 2021-04-02

A.13. v3.0.6 - 2021-02-26

Style your components

- New chapter about styles! (2021-01-07)

Provide/inject

- New chapter about provide/inject! (2021-02-03)

State Management

- New chapter about the Store pattern, Flux libraries, Vuex, and Pinia! (2021-02-25)

Animations and transition effects

- New chapter about animations and transitions! (2021-01-20)

A.14. v3.0.4 - 2020-12-10

How to build components

- Adds a section on how to choose between `ref` and `reactive`. (2020-11-06)

Forms

- Adds a section about custom validators with VeeValidate (2020-12-10)
- Adds a section on VeeValidate configuration (how to validate on input) (2020-12-09)
- VeeValidate now offers only some of the previous meta-flags. (2020-10-13)
- It is now possible to rename a field with VeeValidate to have nicer error messages. (2020-10-07)

Suspense

- Adds a section on the differences between using `Suspense` or `onMounted` (2020-11-20)

Router

- Adds a section about using the router with Suspense (2020-12-04)

A.15. v3.0.0 - 2020-09-18

Forms

- Update VeeValidate to v4, which supports Vue 3 (2020-08-07)

Slots

- The chapter now comes earlier in the book, before the Suspense chapter. (2020-08-07)

A.16. v3.0.0-rc.4 - 2020-07-24

Directives

- Clarify that `v-for` can be used with `in` or `of` (2020-07-16)

Router

- Guards can now return a value instead of having to call `next()`. (2020-07-24)

Slots

- New chapter about Slots! (2020-07-24)

A.17. v3.0.0-beta.19 - 2020-07-08

The wonderful world of Web Components

- Use `customElements.define` instead of the deprecated `document.registerElement`. (2020-06-17)

How to build components

- Explain the `emits` option and how it can be used to validate the emitted event. (2020-06-12)

Under the hood

- New chapter! Learn how Vue works under the hood (parsing, VDOM, etc.) (2020-07-08)
- Add a section about building the reactivity functions from scratch (2020-07-06)
- Add a section about reactivity with getter/setter and proxies (2020-07-06)

A.18. v3.0.0-beta.10 - 2020-05-11

Global

- First public release of the ebook! (2020-05-11)