

Become a
NINJA
with



Vue.js

ninja  *squad*

Deviens un ninja avec Vue

Ninja Squad

Table des matières

| | |
|--|----|
| 1. Introduction | 1 |
| 2. Une rapide introduction à ECMAScript 2015+ | 4 |
| 2.1. Transpileur | 4 |
| 2.2. <code>let</code> | 5 |
| 2.3. Constantes | 6 |
| 2.4. Raccourcis pour la création d'objets | 7 |
| 2.5. Affectations déstructurées | 8 |
| 2.6. Paramètres optionnels et valeurs par défaut | 9 |
| 2.7. <i>Rest operator</i> | 11 |
| 2.8. Classes | 12 |
| 2.9. <i>Promises</i> | 14 |
| 2.10. <i>(arrow functions)</i> | 17 |
| 2.11. <i>Async/await</i> | 20 |
| 2.12. <i>Set</i> et <i>Map</i> | 21 |
| 2.13. Template de string | 22 |
| 2.14. Modules | 23 |
| 2.15. Conclusion | 25 |
| 3. Un peu plus loin qu'ES2015+ | 26 |
| 3.1. Types dynamiques, statiques et optionnels | 26 |
| 3.2. Hello TypeScript | 27 |
| 4. Découvrir TypeScript | 28 |
| 4.1. Les types de TypeScript | 28 |
| 4.2. Valeurs énumérées (<i>enum</i>) | 29 |
| 4.3. Return types | 30 |
| 4.4. Interfaces | 30 |
| 4.5. Paramètre optionnel | 31 |
| 4.6. Des fonctions en propriété | 32 |
| 4.7. Classes | 32 |
| 4.8. Utiliser d'autres bibliothèques | 34 |
| 5. TypeScript avancé | 36 |
| 5.1. <code>readonly</code> | 36 |
| 5.2. <code>keyof</code> | 36 |
| 5.3. Mapped type | 37 |
| 5.4. Union de types et gardien de types | 39 |
| 6. Le monde merveilleux des Web Components | 42 |
| 6.1. Le nouveau Monde | 42 |
| 6.2. Custom elements | 42 |
| 6.3. Shadow DOM | 44 |

| | |
|---|-----|
| 6.4. Template | 44 |
| 6.5. Les bibliothèques basées sur les Web Components | 45 |
| 7. La philosophie de Vue | 47 |
| 8. Commencer de zéro | 51 |
| 8.1. Un framework évolutif | 51 |
| 8.2. Vue CLI | 57 |
| 8.3. Bundlers : Webpack, Rollup, esbuild | 57 |
| 8.4. Vite | 59 |
| 8.5. create-vue | 60 |
| 8.6. Single File Components | 62 |
| 9. Syntaxe des templates | 63 |
| 9.1. Interpolation | 64 |
| 9.2. Utiliser d'autres composants dans un template | 69 |
| 9.3. Lien de propriété avec <code>v-bind</code> | 71 |
| 9.4. Événements avec <code>v-on</code> | 74 |
| 9.5. Templates et TypeScript | 78 |
| 9.6. Résumé | 79 |
| 10. Directives | 80 |
| 10.1. Des conditions dans les templates avec <code>v-if</code> | 80 |
| 10.2. Masquer du contenu avec <code>v-show</code> | 81 |
| 10.3. Rendu unique avec <code>v-once</code> | 81 |
| 10.4. Répéter des éléments avec <code>v-for</code> | 82 |
| 10.5. Du contenu HTML avec <code>v-html</code> | 84 |
| 10.6. Du contenu non interprété <code>v-pre</code> | 85 |
| 10.7. Autres directives | 85 |
| 11. Créer des composants | 86 |
| 11.1. Une propriété réactive avec <code>ref</code> | 86 |
| 11.2. Une propriété réactive avec <code>reactive</code> | 87 |
| 11.3. <code>ref</code> ou <code>reactive</code> | 91 |
| 11.4. Dériver une valeur avec <code>computed</code> | 92 |
| 11.5. Produire un effet de bord avec <code>watchEffect</code> et <code>watch</code> | 93 |
| 11.6. Passer des <code>props</code> aux composants | 95 |
| 11.7. Des événements spécifiques avec <code>emit</code> | 101 |
| 11.8. Fonctions du cycle de vie | 104 |
| 12. API Composition | 107 |
| 12.1. Une conception propre grâce à l'API Composition | 107 |
| 12.2. Extraire de la logique commune | 110 |
| 12.3. L'API Composition hors des composants | 111 |
| 12.4. Un exemple de la communauté : VueUse | 114 |
| 13. Style des composants | 116 |
| 13.1. Styles <code>scoped</code> | 116 |

| | |
|---|-----|
| 13.2. Styles <code>module</code> | 117 |
| 13.3. <code>v-bind</code> en CSS | 117 |
| 13.4. <code>v-deep</code> , <code>v-global</code> et <code>v-slotted</code> | 118 |
| 13.5. PostCSS | 118 |
| 13.6. Pre-processeurs CSS | 119 |
| 14. Les nombreuses façons de définir des composants | 120 |
| 14.1. API Options | 120 |
| 14.2. API de Composition | 121 |
| 14.3. Script <code>setup</code> | 121 |
| 14.4. API Classe | 122 |
| 15. <code>script setup</code> | 124 |
| 15.1. Migrer un composant | 124 |
| 15.2. Retour implicite | 126 |
| 15.3. <code>defineProps</code> | 126 |
| 15.4. <code>defineEmits</code> | 128 |
| 15.5. <code>defineOptions</code> | 129 |
| 15.6. Fermé par défaut et <code>defineExpose</code> | 129 |
| 16. Tester ton application | 130 |
| 16.1. Tester c'est douter | 130 |
| 16.2. Tests unitaires | 130 |
| 16.3. Vitest | 131 |
| 16.4. <code>@vue/test-utils</code> | 134 |
| 16.5. Snapshot testing | 138 |
| 16.6. Tests de bout en bout | 140 |
| 16.7. Cypress | 141 |
| 17. Envoyer et recevoir des données avec HTTP | 143 |
| 17.1. Obtenir des données | 143 |
| 17.2. Options avancées | 145 |
| 17.3. Intercepteurs | 145 |
| 17.4. Tests | 146 |
| 18. Slots | 147 |
| 18.1. Projection de contenu avec <code>slot</code> | 147 |
| 18.2. Slots nommés | 148 |
| 18.3. Contenu par défaut | 150 |
| 18.4. <code>Slot props</code> | 150 |
| 18.5. Slots typés avec <code>defineSlots</code> | 152 |
| 19. Suspense | 154 |
| 19.1. Afficher une alternative | 155 |
| 19.2. Gérer les erreurs avec <code>onErrorCaptured</code> | 156 |
| 19.3. Événements <code>resolve</code> et <code>recede</code> | 157 |
| 19.4. Suspense contre <code>onMounted</code> | 157 |

| | |
|---|-----|
| 19.5. <code>script setup</code> et <code>await</code> | 158 |
| 20. Le routeur | 159 |
| 20.1. En route | 159 |
| 20.2. Navigation | 161 |
| 20.3. Paramètres | 162 |
| 20.4. Routeur et <code>Suspense</code> | 163 |
| 20.5. Passer les paramètres comme <code>props</code> | 164 |
| 20.6. Redirections | 165 |
| 20.7. Sélection de la route | 165 |
| 20.8. Routes imbriquées | 165 |
| 20.9. Gardes de navigation | 167 |
| 20.10. Meta information | 169 |
| 20.11. Tests avec <code>vue-router-mock</code> | 169 |
| 21. Chargement à la demande | 171 |
| 21.1. Composants asynchrones | 171 |
| 21.2. Composants asynchrones et <code>Suspense</code> | 172 |
| 21.3. Chargement à la demande par le routeur | 173 |
| 21.4. Grouper plusieurs composants dans le même <code>bundle</code> | 174 |
| 22. Formulaires | 175 |
| 22.1. La directive <code>v-model</code> | 175 |
| 22.2. De meilleurs formulaires avec <code>VeeValidate</code> | 179 |
| 22.3. Composants de formulaire personnalisés | 187 |
| 22.4. Macro <code>defineModel</code> | 189 |
| 23. Provide/inject | 191 |
| 23.1. Une manière d'éviter les <code>props</code> passe-plat | 191 |
| 23.2. Tester des composants qui utilisent <code>inject</code> | 193 |
| 23.3. <code>Providers</code> hiérarchiques | 193 |
| 23.4. Plugins | 194 |
| 24. Gestion d'état | 195 |
| 24.1. Le pattern <code>store</code> | 195 |
| 24.2. Les librairies <code>flux-like</code> | 197 |
| 24.3. <code>Vuex</code> | 197 |
| 24.4. <code>Pinia</code> | 200 |
| 24.5. Tester <code>Pinia</code> | 202 |
| 24.6. Pourquoi utiliser un store ? | 203 |
| 25. Animations et effets de transition | 205 |
| 25.1. Animations en pur CSS | 205 |
| 25.2. Transitions d'entrée/sortie | 206 |
| 25.3. Transitions dans les listes | 209 |
| 25.4. Et plus encore ! | 210 |
| 26. Patterns pour composants avancés | 211 |

| | |
|--|-----|
| 26.1. Références de template avec <code>ref</code> | 211 |
| 26.2. Références de composant | 212 |
| 27. Directives personnalisées | 213 |
| 27.1. Cycle de vie d'une directive | 213 |
| 27.2. Valeur des directives | 214 |
| 27.3. Argument des directives | 215 |
| 27.4. Modificateurs des directives | 216 |
| 28. Internationalisation | 217 |
| 28.1. Mise en place de vue-i18n | 217 |
| 28.2. Traduction de textes | 218 |
| 28.3. Messages paramétrisés | 218 |
| 28.4. Pluralisation | 219 |
| 28.5. Changer la locale | 219 |
| 28.6. Formatage | 220 |
| 28.7. Autres fonctionnalités (lazy-loading, support de Vite et plus) | 220 |
| 29. Sous le capot | 222 |
| 29.1. Les changements de rendu | 222 |
| 29.2. La compilation des templates | 223 |
| 29.3. Le DOM virtuel | 225 |
| 29.4. JSX | 234 |
| 29.5. Réactivité | 234 |
| 30. Performances | 245 |
| 30.1. Premier chargement | 245 |
| 30.2. Taille des ressources | 245 |
| 30.3. Faire un beau paquet : le <i>bundle</i> | 245 |
| 30.4. <i>Tree-shaking</i> | 246 |
| 30.5. Minification et élimination de code mort | 246 |
| 30.6. Autres types de ressources | 246 |
| 30.7. Compression | 247 |
| 30.8. Chargement fainéant : le <i>lazy-loading</i> | 247 |
| 30.9. Rendu côté serveur | 247 |
| 30.10. Cache pour rechargement | 248 |
| 30.11. Performances à l'exécution | 249 |
| 30.12. <code>key</code> dans <code>v-for</code> | 249 |
| 30.13. <code>v-memo</code> | 250 |
| 30.14. Conclusion | 251 |
| 31. Ce n'est qu'un au revoir | 253 |
| Annexe A: Historique des versions | 256 |
| A.1. v3.3.0 - 2023-05-12 | 256 |
| A.2. v3.2.45 - 2023-01-05 | 256 |
| A.3. v3.2.37 - 2022-07-06 | 257 |

| | |
|---|-----|
| A.4. v3.2.30 - 2022-02-10 | 257 |
| A.5. v3.2.26 - 2021-12-17 | 257 |
| A.6. v3.2.19 - 2021-09-30 | 258 |
| A.7. v3.2.0 - 2021-08-10 | 258 |
| A.8. v3.1.0 - 2021-06-07 | 258 |
| A.9. v3.0.11 - 2021-04-02 | 259 |
| A.10. v3.0.6 - 2021-02-26 | 259 |
| A.11. v3.0.4 - 2020-12-10 | 259 |
| A.12. v3.0.0 - 2020-09-18 | 259 |
| A.13. v3.0.0-rc.4 - 2020-07-24 | 260 |
| A.14. v3.0.0-beta.19 - 2020-07-08 | 260 |
| A.15. v3.0.0-beta.10 - 2020-05-11 | 260 |

Chapter 1. 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 commençons quelques projets Vue pour nos clients, et, une chose 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 3.3.4 dans les exemples.

Chapter 2. Une rapide introduction à ECMAScript 2015+

Si tu lis ce livre, on peut imaginer que tu as déjà entendu parler de JavaScript. Ce qu'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 qu'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 qu'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 !

2.1. 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+.

2.2. 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 !

2.3. Constantes

Tant qu'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
```

2.4. 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!');
    }
  };
}
```

2.5. 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`.

2.6. 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 maintenant `0` ou `" "` sont 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
```

2.7. 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);
```

2.8. Classes

Une des fonctionnalités les plus emblématiques : ES2015 introduit les classes en JavaScript ! Tu pourras désormais 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

```

2.9. 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.

2.10. (*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* !

2.11. Async/await

Nous discutons 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 renvoyée 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.

2.12. 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.

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

2.14. 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 qu'on veut les utiliser. Supposons qu'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 qu'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.

2.15. 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 3. Un peu plus loin qu'ES2015+

3.1. 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.

3.2. 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 parler 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 4. 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.

4.1. 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 qu'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
```

4.2. 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';
```

4.3. 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;
}
```

4.4. 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 évidemment 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 };
```

4.5. 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;
}
```

4.6. 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);
```

4.7. 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 !

4.8. 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 5. 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.

5.1. 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
```

5.2. 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 d'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.

5.3. 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 exactement comme ceci dans la bibliothèque standard.

TypeScript offre également d'autres types de transformation.

5.3.1. 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`
```

5.3.2. 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
```

5.3.3. 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.

5.4. 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, parfois, 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 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 6. 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.

6.1. 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.

6.2. 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.

6.3. 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 façon de procéder. La bonne pratique est de plutôt utiliser l'élément `<template>`.

6.4. 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);
  }

}
```

6.5. 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.

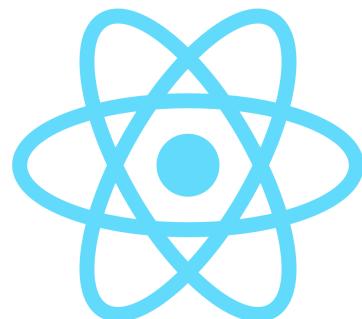
Chapter 7. La philosophie de Vue

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.

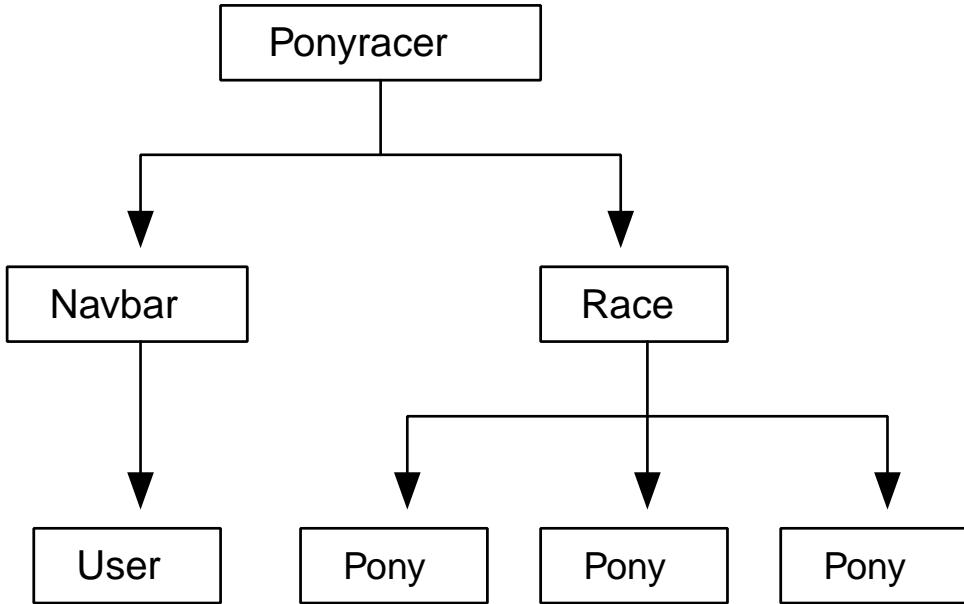
Cette orientation composant est largement partagée par de nombreux frameworks front-end : c'est le cas depuis le début de [React](#), le framework tendance de Facebook ; [Ember](#) et [AngularJS](#) ont leur propre façon de faire quelque chose de similaire ; et les petits nouveaux [Svelte](#) ou [Angular](#) 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 façon 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 8. 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.

8.1. 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@3.3.4`

Si tu rechargeas la page, tu verras que Vue affiche un avertissement dans la console, nous informant qu'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 rechargeas 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 rechargeas 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
      }
    };
    const app = Vue.createApp(RootComponent);
    app.mount('#app');
  </script>
</body>
</html>

```

```

    },
    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'outil 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.

8.2. Vue CLI



La CLI est maintenant 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 maintenant en mode maintenance, et Vite est l'alternative recommandée. Explorons donc pourquoi.

8.3. 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 indiscutible. 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 rechargeement 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 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.

8.4. 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.

8.5. **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 maintenant 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 './assets/main.css';

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">
import HelloWorld from './components/HelloWorld.vue';
import TheWelcome from './components/TheWelcome.vue';
</script>

<template>
  <header>
    

    <div class="wrapper">
      <HelloWorld msg="You did it!" />
    </div>
  </header>

  <main>
    <TheWelcome />
  </main>
</template>

<style scoped>
header {
  line-height: 1.5;
}
</style>
```

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

8.6. 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 qu'on utilise TypeScript ;
- les styles CSS du composant dans l'élément `style`.

L'outil 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'outil, et que nous sommes prêts à écrire plus de composants, passons à la syntaxe des templates.

Chapter 9. Syntaxe des templates

Nous avons déjà appris qu'un composant avait besoin d'un template. Pour simplifier, le template nous permet de générer du code HTML contenant des parties dynamiques, qui dépendent de nos données. C'est à la syntaxe des templates que nous allons nous intéresser dans ce chapitre.

Prenons un exemple simple, qui se contente d'afficher un titre statique :

App.vue

```
<template>
  <h1>Ponyracer</h1>
</template>

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

  export default defineComponent({
    name: 'App'
  });
</script>
```

Comme tu peux le voir, l'élément `<template>` de notre *Single File Component* contient un simple `<h1>`.

Maintenant, nous voulons afficher une donnée sur cette première page. Par exemple le nombre d'utilisateurs enregistrés dans notre application.

Plus loin, nous apprendrons comment obtenir des données depuis un serveur, mais pour le moment, faisons simple et utilisons un nombre en dur dans notre composant.

Pour exposer une donnée au template depuis le composant, il faut qu'elle soit une propriété de l'objet retourné par la fonction `setup` :

App.vue

```
import { defineComponent } from 'vue';

export default defineComponent({
  name: 'App',

  setup() {
    return { numberOfWorkers: 146 };
  }
});
```

Comment modifier notre template pour afficher cette valeur ? La réponse, c'est l'*interpolation*.

9.1. Interpolation

Un bien grand mot pour un concept simple.

Exemple rapide :

App.vue

```
<template>
  <div>
    <h1>Ponyracer</h1>
    <h2>{{ number0fUsers }} users</h2>
  </div>
</template>

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

  export default defineComponent({
    name: 'App',

    setup() {
      return { number0fUsers: 146 };
    }
  });
</script>
```

Note que la `div` racine n'est pas forcément nécessaire, car les templates Vue peuvent avoir plusieurs éléments racines (ce n'était pas le cas avant Vue 3.0).

Nous avons donc créé un composant qui sera activé chaque fois que Vue rencontrera un élément `App`. Ce composant a une propriété `number0fUsers`. Et son template contient un élément `<h2>`, qui utilise les fameuses doubles accolades (aussi appelées "moustaches") pour indiquer qu'une expression doit être évaluée. C'est ce que Vue appelle l'*interpolation*.

Nous devrions donc voir dans le document :

```
<div>
  <h1>PonyRacer</h1>
  <h2>146 users</h2>
</div>
```

`{{ number0fUsers }}` est remplacé par la valeur de l'expression contenue entre les moustaches. Lorsque Vue détecte un élément `App` dans la page, il crée une instance du composant et cette instance constitue le contexte d'évaluation des expressions contenues dans son template. Ici, le composant définit une propriété `number0fUsers` dont la valeur est 146, donc la valeur '146' est affichée à l'écran.

Tel que le code est écrit pour l'instant, si nous modifions la valeur de `number0fUsers` plus tard, cette

modification ne sera pas reflétée dans la vue.

App.vue

```
<template>
<div>
  <h1>Ponyracer</h1>
  <h2>{{ numberOfUsers }} users</h2>
</div>
</template>

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

export default defineComponent({
  name: 'App',

  setup() {
    const result = { numberOfUsers: 146 };
    // update the counter after 3 seconds
    // but the template does not reflect the new value
    setTimeout(() => (result.numberOfUsers = 147), 3000);
    return result;
  }
});
</script>
```

Pour que Vue détecte les changements de valeur de la propriété, il faut rendre cette propriété réactive en utilisant `ref`. Nous reviendrons plus tard en détails sur la syntaxe de `ref`, dans le chapitre dédié aux composants. Pour l'instant, tout ce qui t'importe est de savoir qu'une `ref` est un objet avec une propriété `value`.

App.vue

```
<template>
<div>
  <h1>Ponyracer</h1>
  <h2>{{ numberOfUsers }} users</h2>
</div>
</template>

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

export default defineComponent({
  name: 'App',

  setup() {
    const numberOfUsers = ref(146);
    // update the counter after 3 seconds
  }
});
</script>
```

```

    setTimeout(() => (numberOfUsers.value = 147), 3000);
    return { numberOfUsers };
}
});
</script>

```

Ainsi, pour mettre à jour une propriété réactive, il faut modifier sa propriété `value`. Mais dans le template, on peut utiliser la `ref` comme si c'était une propriété "normale", sans avoir à ajouter `.value` : Vue en extrait la valeur automatiquement.

```

<div>
  <h1>PonyRacer</h1>
  <h2>147 users</h2>
</div>

```

Ça y est, chaque fois que `numberOfUsers` change dans le composant, la vue est mise à jour automatiquement ! C'est l'une des grandes fonctionnalités de Vue. Comme je le disais, mettons de côté l'aspect réactivité pour le moment et concentrons-nous sur les templates.

Un aspect important à retenir : si on essaie d'afficher une propriété qui n'existe pas, au lieu d'afficher `undefined` comme on pourrait s'y attendre, Vue affichera simplement une chaîne vide. Il en va de même si la propriété existe, mais a la valeur `null`.

Imaginons à présent que, au lieu d'une simple valeur, notre composant doive afficher un objet `user` plus complexe, contenant les informations de l'utilisateur courant.

App.vue

```

<template>
  <div>
    <h1>Ponyracer</h1>
    <h2>Welcome {{ user.name }}</h2>
  </div>
</template>

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

export default defineComponent({
  name: 'App',

  setup() {
    const user = ref({
      name: 'Cédric'
    });
    return { user };
  }
});

```

```
</script>
```

Comme tu peux le voir, on peut intercaler des expressions plus complexes, par exemple la propriété d'un objet défini dans `setup`.

```
<div>
  <h1>PonyRacer</h1>
  <h2>Welcome Cédric</h2>
</div>
```

Fait intéressant : si la valeur de l'expression interpolée est un objet, tu verras la structure JSON de cet objet, ce qui est pratique pendant le développement :

App.vue

```
<template>
  <div>
    <h1>Ponyracer</h1>
    <h2>{{ user }}</h2>
    <!-- displays { "id": 1, "name": "Cédric" } -->
  </div>
</template>

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

export default defineComponent({
  name: 'App',

  setup() {
    const user = ref({
      id: 1,
      name: 'Cédric'
    });
    return { user };
  }
});
</script>
```

Que se passe-t-il si on se trompe dans le template, et qu'on tente d'accéder à une propriété d'un objet qui n'existe pas dans le composant ?

App.vue

```
<template>
  <div>
    <h1>Ponyracer</h1>
    <!-- Note the typo: `users` instead of `user` -->
    <h2>Welcome {{ users.name }}</h2>
```

```
</div>
</template>
```

Lorsque l'application démarre, une erreur sera générée, avec l'explication indiquant que la propriété n'existe pas :

```
'[Vue warn]: Property "users" was accessed during render but is not defined on
instance'
```

Vous n'observerez cette erreur qu'à l'exécution de l'application, dans le navigateur. Malheureusement, aucun avertissement ne sera levé à la compilation si un template contient des erreurs, à la différence d'autres frameworks comme Angular. On peut espérer cependant que vous ayez des tests unitaires qui permettent de détecter ce genre de problème avant de partir en production. Nous verrons comment tester unitairement les composants et leur template dans un prochain chapitre.



Il y a cependant quelques initiatives de la communauté pour avoir cette vérification des templates à la compilation :

- [Volar](#) (le plus avancé actuellement, que nous utilisons dans nos projets, les exemples de cet ebook et les exercices du Pack Pro)
- [VTI](#) de l'équipe qui fait également Vetur
- [VueDX](#) Tous ces projets sont encore expérimentaux, mais valent d'être essayé.

Revenons à notre exemple. On affiche maintenant un message de bienvenue. Allons un peu plus loin et affichons les courses de poneys à venir. La première intuition serait d'ajouter plus de code HTML et TypeScript à notre composant [App](#). Mais à y réfléchir, ne serait-il pas préférable de garder ce composant aussi simple et testable que possible. Et d'ailleurs, la liste des courses pourrait bien être nécessaire ailleurs que sur la page principale.

Cette réflexion nous conduit à écrire un second composant. Allons-y pas à pas et commençons très simple :

Races.vue

```
<template>
  <div>
    <h2>Races</h2>
  </div>
</template>

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

  export default defineComponent({
    name: 'Races'
```

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

Rien de compliqué : un simple composant avec un nom `Races` et un template affichant un titre dans un `h2`.

Il nous faut inclure ce nouveau composant dans le template du composant `App`. Comment faire ?

9.2. Utiliser d'autres composants dans un template

Nous avons donc un composant `App`, dans lequel on veut afficher un deuxième composant `Races`.

`App.vue`

```
<template>  
  <div>  
    <h1>Ponyracer</h1>  
    <Races />  
  </div>  
</template>  
  
<script lang="ts">  
  import { defineComponent } from 'vue';  
  
  export default defineComponent({  
    name: 'App'  
  });  
</script>
```

Comme tu peux le voir, nous avons ajouté le composant `Races` dans le template, en utilisant un élément HTML dont le nom est celui que nous avons choisi pour le composant `Races`.

Nous sommes sur la bonne voie, maaaiii... ça ne fonctionne pas : le navigateur n'affiche pas le composant `Races`.

Si tu ouvres la console du navigateur, tu peux y lire :

```
'[Vue warn]: Failed to resolve component: Races'
```

Pourquoi une telle erreur ? La raison est simple : Vue ne connaît pas encore ce composant `Races`.

9.2.1. Enregistrement local

La correction n'est pas compliquée. Dans notre composant `App`, nous pouvons ajouter une propriété `components`, pour y lister les sous-composants utilisés dans le template. La syntaxe est la suivante : `components` est un objet auquel tu peux ajouter une clé qui est le nom de l'élément HTML utilisé pour insérer un composant et la valeur correspondante est le composant qui viendra remplacer l'élément HTML dans le template.

Dans l'exemple suivant, nous y ajoutons la clé `Races` (puisque c'est le nom de l'élément que nous avons ajouté au template d'`App`), avec la valeur `Races`, qui est le composant que nous voulons utiliser. Bien sûr, nous devons importer le composant `Races` depuis le fichier où il est déclaré.

App.vue

```
<template>
  <div>
    <h1>Ponyracer</h1>
    < Races />
  </div>
</template>

<script lang="ts">
  import Races from '@/chapters/templates/Races.vue';
  import { defineComponent } from 'vue';

  export default defineComponent({
    name: 'App',
    components: {
      Races: Races
    }
  });
</script>
```

Nous pourrions aussi utiliser `<races></races>` dans le template, mais la plupart des développeurs utilisent le nom original, en *PascalCase*.

La clé pourrait être complètement différente cependant. Tu *pourrais* choisir d'utiliser par exemple `{ RacesList: Races }`, et donc utiliser `< RacesList ></ RacesList >` dans le template. Et tu pourrais ensuite utiliser `<races-list></races-list>`, la version *kebab-case*, dans le template. Cela fonctionnerait parfaitement, mais ce serait assez surprenant pour tes collègues. En règle générale, nous utilisons le nom du composant.

Note que la syntaxe de ES6 nous permet d'écrire `{ key }` au lieu de `{ key: key }` si tu te souviens des premiers chapitres. On peut donc simplifier légèrement la déclaration du composant en utilisant `{ Races }` au lieu de `{ Races: Races }`. Nous utiliserons cette syntaxe concise à partir de maintenant.

9.2.2. Enregistrement global

Si tu utilises un composant dans de nombreux autres composants ça devient un peu pénible de devoir l'enregistrer à *chaque fois*.

Vue permet d'enregistrer un composant globalement. Dans ton fichier `main`, tu peux utiliser la fonction `component` pour enregistrer globalement un ou plusieurs composants. Ces composants seront alors disponibles dans tous les templates de l'application.

```
createApp(App)
  .component('CustomButton', CustomButton)
  .mount('#app');
```

9.3. Lien de propriété avec v-bind

L'interpolation n'est qu'une des techniques pour introduire du dynamisme dans les templates.

Dans Vue, chaque attribut des éléments HTML peut prendre une valeur dynamique, non pas en utilisant les moustaches, mais en préfixant le nom de l'attribut avec **v-bind**:

v-bind est ce qu'on appelle une *directive* dans Vue. Il y en a d'autres que nous décrirons en détails. On peut les détecter facilement dans les templates parce qu'elles commencent par **v-**.

Ainsi, si tu veux affecter une valeur dynamique à la source d'une image, par exemple une propriété **dynamicUrl** de ton composant, tu peux écrire :

```

```

Cela fonctionne également avec des attributs booléens, comme **disabled** ou **selected**, qui sont un peu différents des autres. En effet, en HTML, ce qui importe pour ces attributs n'est pas leur valeur, mais leur présence ou leur absence sur l'élément. Avec **v-bind** cependant, on peut affecter une valeur à ces attributs. Si la valeur est "truthy" dans le sens JavaScript du terme (c'est-à-dire n'est pas **false**, **null**, **undefined**, une chaîne vide ou 0), alors l'attribut sera considéré comme présent :

```
<button v-bind:disabled="isDisabled">Log in</button>
```

Dans cet exemple, **isDisabled** est une propriété de notre composant, et peut être **true** ou **false**. Si elle est **true**, alors le bouton sera désactivé, comme lorsque l'attribut **disabled** est présent et l'utilisateur ne pourra donc pas cliquer sur le bouton. Si elle est **false**, le bouton sera activé et cliquable, comme lorsque l'attribut **disabled** est absent.

9.3.1. Syntaxe raccourcie

Tu peux aussi écrire une version plus concise de **v-bind:attribute** en utilisant **:attribute**. C'est cette syntaxe raccourcie que la plupart des développeurs Vue préfèrent :

```

<button :disabled="isDisabled">Log in</button>
```

9.3.2. Propriétés et attributs

J'ai parlé d'*attributs* depuis le début de cette section, mais son titre est "lien de propriété", et pas

"lien d'attribut". Pourquoi cela ?

Le titre fait référence aux propriétés du DOM, mais peut-être faut-il clarifier la différence entre les deux. Ce que nous écrivons, ce sont des balises HTML contenant des attributs, on est d'accord ? Prenons ce simple exemple de code HTML :

```
<input type="text" value="hello">
```

La balise `input` ci-dessus a deux *attributs* : l'attribut `type` et l'attribut `value`. Lorsque le navigateur analyse cette balise, il crée un nœud dans le document DOM (un `HTMLInputElement` si on veut être précis), qui a des *propriétés* correspondantes `type` et `value`. Chaque attribut HTML standard a une propriété correspondante dans le nœud du DOM. Mais le nœud du DOM a aussi d'autres propriétés, qui n'ont pas nécessairement d'attribut correspondant. Par exemple : `childElementCount`, `innerHTML` ou `textContent`.

L'interpolation que nous avons utilisée pour afficher le nom de l'utilisateur :

```
<div>{{ user.name }}</div>
```

est en fait à peu près équivalente au code suivant :

```
<div :textContent="user.name"></div>
```

La directive `v-bind` permet de modifier la propriété DOM `textContent`, et on lui assigne la valeur de l'expression `user.name`, qui est évaluée dans le contexte du composant courant, exactement comme on le faisait pour l'interpolation.

La mécanique utilisée par la directive `v-bind:something="value"` est très simple :

- d'abord, elle tente de modifier la propriété `something` si elle existe ;
- autrement, elle affecte la valeur à l'attribut `something`, ou l'enlève si `value` est null.

Note que cette mécanique est sensible à la casse, donc il faut utiliser le nom correct de la propriété : `textcontent` ou `TEXTCONTENT` ne fonctionneront pas ; il faut utiliser `textContent`.

9.3.3. `v-bind` avec un objet

Il est aussi possible d'utiliser `v-bind` sans préciser de nom de propriété, mais en lui donnant un objet directement. Dans ce cas, la directive lie toutes les clés de l'objet :

```
<button v-bind="buttonProperties">Log in</button>
```

avec une propriété `buttonProperties`, qui peut avoir autant de paires de clé/valeur que tu le désires :

```
setup() {
```

```

return {
  buttonProperties: ref({
    disabled: true,
    type: 'button' as const
  }),
};
}

```

9.3.4. Classes et styles

Il est assez commun dans les applications web d'avoir à calculer quelles classes ou styles CSS doivent être appliqués à un élément. On peut faire ça manuellement en construisant une chaîne de caractères `computedClasses` contenant toutes les classes à appliquer, et ensuite utiliser `:class="computedClasses"` dans le template. Il en va de même pour les styles avec `:style="computedStyles"`.

Mais `v-bind` permet de faire les choses plus élégamment pour ces deux cas d'utilisation fréquents. On peut utiliser `:class` et `:style` avec un objet comme valeur, pour dynamiquement ajouter ou enlever des classes et des styles CSS à l'élément.

Par exemple,

```
<button class="btn" :class="{ 'btn-primary': isPrimary, 'btn-sm': isSmall }">Log in</button>
```

avec les propriétés `isPrimary` et `isSmall`:

```

setup() {
  return {
    isPrimary: ref(true),
    isSmall: ref(false),
  };
}

```

ajoutera la classe CSS `btn-primary` au bouton et enlèvera la classe `btn-sm`.

Note qu'une alternative est d'exposer directement depuis le composant un tableau de classes CSS, ou un objet comme celui que nous avons défini dans le template :

```
<button class="btn" :class="buttonClasses">Log in</button>
```

```

setup() {
  return {
    buttonClasses: ref({
      'btn-primary': true,
      'btn-small': false
    })
  };
}

```

```
    }),
  };
}
```

Le même principe peut être utilisé pour les styles CSS, avec `:style` :

```
<div :style="textStyle">Some text</div>
```

```
setup() {
  return {
    textStyle: ref({
      color: 'red',
      fontWeight: 'bold' as const
    })
  };
}
```

9.4. Événements avec `v-on`

Si tu développes des applications web, tu sais qu'afficher des données n'est qu'une petite partie du travail : il faut aussi gérer les interactions avec l'utilisateur. Pour permettre ces interactions, le navigateur émet des événements, qu'on peut écouter : `click`, `keyup`, `mousemove`, etc. Vue est encore là pour nous aider à les gérer !

Supposons que nous voulions qu'un bouton enregistre une action lorsque l'utilisateur clique dessus.

Réagir à un événement peut être fait en utilisant la directive `v-on`, comme suit :

```
<button v-on:click="save()">Save</button>
```

Un clic sur le bouton de l'exemple ci-dessus appellera la fonction `save()` du composant.

Ajoutons-la :

```
export default defineComponent({
  name: 'Races',
  setup() {
    function save() {
      // Do something when user saves
    }
    return {
      save
    };
  }
})
```

```
});
```

Comme pour les données, les composants Vue exposent des fonctions au template en les retournant en tant que propriétés dans la fonction `setup`.

Comme c'était le cas pour `v-bind`, la plupart des développeurs Vue préfèrent utiliser la syntaxe raccourcie de `v-on:event :@event`.

```
<button @click="save()">Save</button>
```

C'est la syntaxe que nous utiliserons dans le reste du livre.

La valeur passée à la directive `v-on` peut être un appel de fonction, mais elle peut être n'importe quelle instruction exécutable, par exemple :

```
<button @click="firstname = 'Cédric'">Save</button>
```

Note que `firstname` doit être une propriété du composant.

```
const firstname = ref('JB');
return {
  firstname,
};
```

Cependant, je ne recommande pas d'utiliser de telles instructions dans le template. Appeler une fonction respecte l'encapsulation du composant, rend le code plus maintenable et testable, et rend le code du template plus simple.

L'événement peut être émis par l'élément lui-même, ou par l'un de ses descendants. Vue réagira à l'événement même s'il se propage depuis un descendant. Considérons le template suivant :

```
<div @click="save()">
  <button>Save</button>
</div>
```

Même si c'est sur le bouton que l'utilisateur clique, la fonction `save()` sera appelée, parce que l'événement se propage à son `div` parent.

On peut bien sûr passer des arguments à la fonction. Et l'un de ces arguments peut être l'événement lui-même ! Il suffit pour cela de passer `$event` à la fonction :

```
<button @click="saveName('Cédric', $event)">Save</button>
```

On peut ainsi gérer l'événement comme on l'entend depuis la fonction :

```
function saveName(name: string, event: Event) {  
    // Do something with the name  
    event.preventDefault();  
    event.stopPropagation();  
}
```

Si on ne stoppe pas sa propagation, l'événement continuera à se propager vers les éléments parents et déclenchera donc potentiellement d'autres gestionnaires d'événement installés plus haut dans l'arbre.

Tu peux utiliser l'événement pour empêcher le comportement par défaut ou stopper sa propagation comme montré ci-dessus. Ou tu peux utiliser les *modifiers* `.stop` et `.prevent`.

9.4.1. Les *modifiers* `.prevent` et `.stop`

Au lieu de gérer l'événement dans la fonction, tu peux ajouter un *modifier* à la directive `v-on`.

`.prevent` empêche le comportement par défaut du navigateur. Par exemple, pour réagir à l'événement `submit` d'un formulaire, mais ne pas recharger la page à la soumission :

```
<form @submit.prevent="save()">  
    <!-- a form -->  
    <button type="submit">Save</button>  
</form>
```

`.stop` quant à lui arrête la propagation de l'événement vers le haut de l'arbre :

```
<button @click.stop="save()">Save</button>
```

Tu peux chaîner plusieurs *modifiers* si nécessaire :

```
<button @click.prevent.stop="save()">Save</button>
```

Il en existe quelques autres pour `v-on`, intéressants notamment pour la gestion des événements clavier et souris.

9.4.2. Événements clavier

Les événements clavier sont vraiment faciles à gérer grâce aux *modifiers* qu'on peut ajouter après `keyup`, `keydown` ou `keypress` :

- `.esc` déclenche le gestionnaire si la touche est `esc` ;
- `.delete` déclenche le gestionnaire si la touche est `delete` ;
- `.space` déclenche le gestionnaire si la touche est `space` ;

- `.up` déclenche le gestionnaire si la touche est `up` ;
- `.down` déclenche le gestionnaire si la touche est `down` ;
- `.left` déclenche le gestionnaire si la touche est `left` ;
- `.right` déclenche le gestionnaire si la touche est `right` ;
- `.[n'importe quel caractère]`, par exemple `.b` déclenche le gestionnaire si la touche est `b`.

```
<textarea @keydown.space="onSpacePress()"></textarea>
```

Chaque fois qu'on enfonce la barre d'espace, la fonction `onSpacePress()` sera appelée. Et on peut aussi les combiner avec d'autres *modifiers*, appelés *system modifiers* : `.ctrl`, `.shift`, `.alt`, `.meta`, comme `@keydown.ctrl.space`.

```
<textarea @keydown.ctrl.space="showHint()"></textarea>
```

9.4.3. `.exact`

Si tu veux qu'un gestionnaire soit appelé pour la combinaison de touches données, et uniquement cette combinaison, sans aucun *system modifier* additionnel, tu peux ajouter le *modifier* `.exact`.

```
<!-- 'save' is called on ctrl+click, ctrl+alt+click, etc. -->
<button @click.ctrl="save()">Save</button>
<!-- 'save' is called only on ctrl+click -->
<button @click.ctrl.exact="save()">Save</button>
```

9.4.4. Événements souris

Le même principe peut être utilisé pour les événements souris avec les *modifiers* `.left`, `.middle` et `.right`.

Par exemple :

```
<button @mousedown.right="save()">Save</button>
```

9.4.5. `.once`

Pour qu'un gestionnaire soit appelé une seule fois, tu peux utiliser le *modifier* `.once` :

```
<button @click.once="save()">Save</button>
```

9.4.6. `.passive` modifier

Le *modifier* `.passive` est un peu différent : il permet de dire au navigateur que le comportement par

défaut devrait être exécuté sans attendre que le gestionnaire d'événement ait terminé.

Cela peut être pratique pour gérer des événements qui sont émis à haute fréquence, comme un `scroll` ou un `mousemove`, mais sans affecter la fluidité du scroll du navigateur :

```
<div @mousemove.passive="onMouseMove()">Some content</div>
```

9.4.7. `.self`

Ce *modifier* est utilisé pour ne réagir que si l'événement est émis par l'élément lui-même, et pas par un de ses descendants :

```
<!-- calls 'save' only if the click is on the 'div' -->
<!-- and not if it is on the 'button' -->
<div @click.self="save()">
  <button>Save</button>
</div>
```

9.4.8. `.capture`

Pour être complet, il y a un dernier *modifier*, qu'on n'utilise que très rarement : `.capture`. Il permet de capturer un événement émis par un descendant et de le gérer en premier, avant que les descendants ne puissent le gérer eux-mêmes :

```
<!-- calls 'save' first if the click is on the 'button' -->
<!-- before calling 'saveName' from the 'button' -->
<div @click.capture="save()">
  <button @click="saveName('Cédric', $event)">Save</button>
</div>
```

9.5. Templates et TypeScript

Comme on le disait plus haut, il n'y a pas d'erreur de compilation si tu as une erreur dans une expression de ton template. Tu auras seulement une erreur au runtime. `Volar` peut cependant vous aider et va être capable d'analyser vos templates pour remonter les erreurs potentielles ! Pour ce faire, il s'appuie sur `TypeScript`, et n'est pas parfait, mais c'est de loin le mieux que vous pouvez espérer (nous, on l'utilise partout).

Note que Vue 3.2+ permet d'écrire du `TypeScript` dans les templates directement. Ce qui est parfois pratique pour indiquer à `Volar` que telle variable n'est pas nulle, ou de tel type :

```
<template>
  <div>
    <h2>Welcome {{ (user!.name as string).toLowerCase() }}</h2>
  </div>
```

```
</template>
```

9.6. Résumé

Les templates de Vue offrent une syntaxe riche et déclarative pour exprimer comment gérer les parties dynamiques de la vue des composants. Vue permet de faire de l'affichage, du binding de propriétés, d'attributs et d'événements, de façon claire, avec peu d'éléments syntaxiques différents :

- `{{}}` pour l'interpolation ;
- `v-bind:property` ou `:property` pour le binding de propriétés et d'attributs ;
- `v-on:event` ou `@event` pour le binding d'événements.

Cela nécessite un peu de pratique pour être complètement à l'aise avec cette syntaxe, mais je suis sûr que ça viendra vite, et qu'une fois maîtrisée, les templates te sembleront lisibles et faciles à écrire.

Si tu veux mettre en pratique ce chapitre, exerce-toi avec l'exercice *Templates* de notre formation en ligne !



Essaie notre [quiz](#) et l'exercice [Templates](#) . Ils sont gratuits et font partie de notre formation en ligne (Pro Pack), où tu apprendras à construire une application complète, pas à pas. Cet exercice consiste à construire un petit composant, un menu responsive, et à jouer avec son template.

Parlons à présent des directives !

Chapter 10. Directives

Les directives sont des outils essentiels à l'écriture d'une application avec Vue. Nous en avons déjà utilisé deux : `v-bind` et `v-on`. Il y en a d'autres. Certaines sont vraiment très utiles. D'autres sont moins fréquemment utilisées, mais néanmoins bonnes à connaître.

Commençons par celle qu'on utilise quotidiennement : `v-if`.

10.1. Des conditions dans les templates avec `v-if`

Cette directive porte un nom évocateur. `v-if` permet, comme son nom le laisse supposer, de n'afficher une section du template que si une condition est vraie :

```
<div v-if="user.role === 'ADMIN'>
  <h2>Admin {{ user.name }}</h2>
</div>
```

La `div` elle-même n'est affichée que si l'utilisateur est un administrateur.

Tu peux combiner `v-if` avec la directive `v-else`, pour afficher une alternative si la condition est fausse.

```
<div v-if="user.role === 'ADMIN'>
  <h2>Admin {{ user.name }}</h2>
</div>
<div v-else>
  <h2>Hello</h2>
</div>
```

Et tu peux aller plus loin en utilisant `v-else-if` :

```
<div v-if="user.role === 'ADMIN'>
  <h2>Admin {{ user.name }}</h2>
</div>
<div v-else-if="user.role === 'ACCOUNTANT'>
  <h2>Accountant</h2>
</div>
<div v-else-if="user.role === 'DEVELOPER'>
  <h2>Developer</h2>
</div>
<div v-else>
  <h2>Hello</h2>
</div>
```

10.2. Masquer du contenu avec v-show

Il existe une autre directive similaire à `v-if` : `v-show`. `v-show` applique le style CSS `display: none;` à un élément si la condition qui lui est fournie est fausse. `v-if` permet aussi de masquer du contenu en fonction d'une condition, mais elle fait plus que ça : lorsque la condition est fausse, `v-if` enlève le contenu du DOM, et elle le recrée et le réinsère dans le DOM lorsque la condition devient vraie.

```
<div v-show="user.role === 'ADMIN'">
  <h2>Admin {{ user.name }}</h2>
</div>
```

Très souvent, tu peux utiliser l'une ou l'autre sans différence visible de comportement.

Ce n'est pas toujours le cas cependant. Par exemple, masquer un élément de formulaire invalide avec `v-show` ne rendra pas le formulaire valide. De même, si des expressions utilisées dans la section invisible lèvent des erreurs lorsqu'elles sont évaluées et que la condition est fausse, utiliser `v-if` est impératif.

Il peut aussi y avoir des différences de performance :

- `v-if` détruit réellement les éléments du DOM et les *listeners* associés, et les recrée au besoin. Elle a donc un coût un peu plus grand si la condition change. Mais quand la condition est fausse, Vue n'a plus à surveiller et mettre à jour toute la section enlevée du DOM. Elle permet donc une meilleure performance quand une large part du DOM est enlevée, et que la valeur de la condition change rarement.
- `v-show` ne fait que rendre invisible une partie du DOM, mais Vue continue à surveiller et mettre à jour cette partie invisible. En revanche, le coût d'un changement de valeur de la condition est minimal, puisqu'elle ne fait que changer la valeur d'un style CSS.

Ne t'embarrasse pas trop avec ces considérations de performance cependant, et utilise ce qui est correct, et ce qui te semble le mieux. ☐

10.3. Rendu unique avec v-once

Tant que nous parlons de performance, évoquons la directive `v-once`, qui peut être bénéfique. Lorsqu'elle est appliquée à un élément, cet élément et tous ses descendants ne seront rendus qu'une seule fois. Ensuite, Vue ne s'occupera plus du tout de toute cette partie du DOM.

Cette directive peut être utile pour des longues pages affichant de nombreuses informations, mais qui ne changent pas :

```
<div v-once>
  <!-- never updates, even if 'user.name' changes-->
  <h2>Admin {{ user.name }}</h2>
</div>
```

10.4. Répéter des éléments avec v-for

C'est inévitable : ton application finira forcément par devoir afficher des listes de quelque chose.

C'est pour résoudre ce problème ultra-commun que `v-for` est prévu : elle permet d'instancier un élément dans le DOM pour chaque élément d'une collection. Notre composant `Races` contient une propriété `races`, renournée par la fonction `setup()`, qui, comme tu t'en doutes, est un tableau de courses à afficher.

```
setup() {
  const races = ref([
    { id: 1, name: 'Lyon' },
    { id: 2, name: 'Amsterdam' }
  ]);
  return {
    races,
  };
}
```

Nous pouvons afficher ce tableau de courses avec `v-for` :

```
<ul>
  <li v-for="race in races">{{ race.name }}</li>
</ul>
```

Note que tu peux utiliser `in` ou `of` indifféremment :

```
<ul>
  <li v-for="race of races">{{ race.name }}</li>
</ul>
```

Et cela nous donne une jolie liste à puces, avec un élément `li` pour chaque course de notre tableau !

Si un nouvel élément est ajouté à la collection, Vue le détectera et ajoutera automatiquement un nouveau `li`. Et évidemment, la liste sera aussi mise à jour si le tableau est réordonné ou si des éléments sont supprimés du tableau.

Note qu'il est recommandé d'aider Vue à traquer quel élément du tableau correspond à quel élément du DOM. Nous reviendrons là-dessus en détails plus tard, mais pour l'instant, sache que tu peux utiliser un `binding :key` pour aider Vue à identifier fonctionnellement un élément du tableau. Dans notre cas, l'identifiant de la course est le meilleur candidat :

```
<ul>
  <li v-for="race in races" :key="race.id">{{ race.name }}</li>
</ul>
```

```
<ul>
  <li>Lyon</li>
  <li>Amsterdam</li>
</ul>
```

Il est possible de connaître l'index de l'élément courant à l'intérieur d'un **v-for** :

```
<ul>
  <li v-for="(race, index) in races" :key="race.id">{{ index }} - {{ race.name }}</li>
</ul>
```

Les parenthèses sont optionnelles, mais je trouve ça plus clair lorsqu'elles sont présentes. **index** contient l'index de l'élément courant, commençant à 0.

```
<ul>
  <li>0 - Lyon</li>
  <li>1 - Amsterdam</li>
</ul>
```

Itérer sur un tableau est de loin l'usage le plus courant de **v-for**, mais tu peux aussi t'en servir pour itérer sur les propriétés d'un objet :

```
<ul>
  <li v-for="(pony, position) in podium" :key="position">{{ position }}: {{ pony
}}</li>
</ul>
```

où **podium** est défini ainsi dans le **setup** du composant :

```
setup() {
  const podium = ref({
    gold: 'Rainbow Dash',
    silver: 'Pinkie Pie',
    bronze: 'Sweet Milk'
  });
  return {
    podium,
  };
}
```

L'index de la propriété est également accessible :

```
<ul>
  <li v-for="(pony, position, index) in podium" :key="position">{{ index + 1 }} - {{
```

```
position }}: {{ pony }}</li>
</ul>
```

```
<ul>
  <li>1 - gold: Rainbow Dash</li>
  <li>2 - silver: Pinkie Pie</li>
  <li>3 - bronze: Sweet Milk</li>
</ul>
```

Sois prudent cependant : si l'objet vient du serveur, en JSON, il n'y a pas de garantie sur l'ordre des propriétés de l'objet. En général, il est préférable d'utiliser un tableau lorsque l'ordre a de l'importance.

Tu peux aussi itérer sur un intervalle très facilement :

```
<ul>
  <li v-for="number in 3" :key="number">{{ number }}</li>
</ul>
```

Note que l'intervalle commence à 1 :

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

10.5. Du contenu HTML avec `v-html`

Si tu veux afficher du contenu HTML, ta première intuition sera sans doute de faire :

```
<div>{{ dynamicHTML }}</div>
```

avec :

```
setup() {
  return {
    dynamicHTML: ref('<strong>Cédric</strong>')
  };
}
```

Mais cela ne fonctionnera pas : le *code source* du HTML sera affiché... On peut cependant faire un *rendu* d'un contenu HTML en utilisant la propriété `innerHTML` :

```
<div :innerHTML="dynamicHTML"></div>
```

Ou, plus simplement, en utilisant la directive `v-html` :

```
<div v-html="dynamicHTML"></div>
```

Attention cependant ! Le contenu n'est pas interprété par Vue. Donc les interpolations ou les directives que pourraient contenir ce contenu HTML seront complètement ignorées par Vue. Il s'agit aussi évidemment d'être particulièrement prudent avec ce genre de contenu dynamique HTML : s'il ne provient pas d'une source fiable, tu risques de gros problèmes de sécurité.

10.6. Du contenu non interprété `v-pre`

Presque à l'inverse de `v-html`, `v-pre` permet précisément d'inclure dans le template du HTML statique qui ne doit pas être interprété par Vue, par exemple de l'interpolation.

```
<div v-pre>{{ hello }}</div>
```

affichera :

```
<div>{{ hello }}</div>
```

10.7. Autres directives

Il existe 3 autres directives natives. La première est `v-cloak`, qui permet de masquer un template tant que Vue est en train de le compiler. Mais cette directive n'est pas nécessaire si on utilise Vite ou la CLI, parce qu'aucun template ne sera de toute façon affiché avant d'être compilé.

Les deux autres, `v-model` et `v-slot`, sont plus importantes. Nous y reviendrons dans les chapitres suivants : elles méritent des explications détaillées. Suspense, suspense...

En attendant, nous avons un exercice pour toi !



Essaie notre troisième exercice [List of races](#) 🐾 ! Il est gratuit, et fait partie de notre Pack Pro, où tu apprendras par la pratique à construire une application complète pas à pas. Cet exercice te guide dans la construction d'un autre composant : la liste des courses. Ce qui semble être un bon cas d'usage pour `v-for` !

Chapter 11. Créer des composants



Depuis Vue 3, il y a plusieurs façons de créer des composants. Ce livre utilise la nouvelle *Composition API* introduite par Vue 3. Si tu as utilisé Vue 1 ou Vue 2 auparavant, l'*Options API* t'est sûrement plus familière.

Jusqu'à présent, nous nous sommes principalement concentrés sur les templates, sans trop se soucier de comment on écrit le composant lui-même. Mais nous avons déjà mentionné l'importance des *propriétés réactives* pour que Vue mette à jour le DOM lorsque l'état du composant change.

Le principe fondamental de Vue est de se baser sur des templates déclaratifs, affichant les données exposées par le composant et rafraîchissant la vue chaque fois que les données changent. Pour que la magie de Vue puisse opérer, le framework doit être conscient des changements apportés aux données affichées par le template. Certains frameworks, comme Angular, détectent les changements chaque fois qu'un événement se produit en comparant les nouvelles valeurs des expressions utilisées dans le template avec leur valeur précédente. Vue utilise une stratégie différente. Il utilise ce qu'il appelle des "propriétés réactives" (*reactive properties*).

Les propriétés réactives sont des propriétés qui sont capables de signaler qu'elles ont été modifiées. En rendant les propriétés réactives, on permet à Vue d'être averti des changements apportés à ces propriétés, ce qui lui permet de mettre à jour le DOM quand c'est nécessaire.

Voyons cela un peu plus en détails.

11.1. Une propriété réactive avec `ref`

Vue 3 introduit une fonction `ref` que tu peux utiliser pour définir une propriété réactive. Vue mettra ainsi la vue à jour lorsque la `value` de la `ref` change, sans que tu aies à faire quoi que ce soit. On peut utiliser la propriété `ref` dans le template directement : Vue en extraira la `value` automatiquement :

App.vue

```
<template>
  <div>
    <h1>Ponyracer</h1>
    <h2>{{ number0fUsers }} users</h2>
  </div>
</template>

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

  export default defineComponent({
    name: 'App',

    setup() {
      const number0fUsers = ref(146);
      // update the counter after 3 seconds
    }
  })
</script>
```

```

    setTimeout(() => (numberOfUsers.value = 147), 3000);
    return { numberOfUsers };
}
});
</script>

```

L'intérêt de TypeScript et de son inférence de type est notable ici : `numberOfUsers` est de type `Ref<number>`. Donc si tu tentes d'écrire `numberOfUsers = 147` ou `numberOfUsers.value = 'hello'`, TypeScript te préviendra gentiment que ça n'a pas de sens 🚫.

11.2. Une propriété réactive avec `reactive`

`ref` est pratique pour travailler avec des valeurs primitives. Il n'y a pas d'autre moyen pour rendre une valeur primitive réactive que de l'envelopper dans un objet. Mais lorsqu'on travaille déjà avec des objets, c'est un peu pénible d'avoir à toujours écrire `.value.property` pour modifier la valeur d'une propriété de l'objet :

Pony.vue

```

export default defineComponent({
  name: 'Pony',

  setup() {
    const pony = ref({
      name: 'Rainbow Dash',
      color: 'GREEN'
    });
    // update the name after 3 seconds
    setTimeout(() => {
      pony.value.name = 'Pinkie Pie';
    }, 3000);
    return { pony };
  }
});

```

L'idéal serait de rendre l'objet lui-même réactif. C'est ici que la fonction `reactive()` aide ! Elle est similaire à `ref` dans le sens qu'elle permet de définir des propriétés réactives. Et Vue va donc observer les changements et rafraîchir la vue en conséquence.

Pony.vue

```

export default defineComponent({
  name: 'Pony',

  setup() {
    const pony = reactive({
      name: 'Rainbow Dash',
      color: 'GREEN'
    });
  }
});

```

```
// update the name after 3 seconds
setTimeout(() => {
  pony.name = 'Pinkie Pie';
}, 3000);
return { pony };
});
```

Comme tu peux le voir, plus besoin de l'indirection `.value` ici. Tu peux modifier l'état de l'objet de manière naturelle, et tu peux toujours utiliser `{{ pony.name }}` dans le template !

Le truc cool, c'est que l'effet est récursif : les sous-propriétés des propriétés de l'objet sont elles aussi réactives !

Pony.vue

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

  setup() {
    const pony = reactive({
      name: 'Rainbow Dash',
      color: 'GREEN',
      origin: {
        country: 'FRANCE'
      }
    });
    // update the country of origin after 3 seconds
    setTimeout(() => {
      pony.origin.country = 'GERMANY';
    }, 3000);
    return { pony };
  }
});
```

Cela a le même effet sur les collections et leurs éléments, aussi bien présents que futurs : si on ajoute une nouvelle propriété à un objet réactif ou un nouvel élément à l'une de ses collections, le nouvel objet sera, lui aussi, réactif. Cela n'était pas le cas dans Vue 2.x, et en constituait l'une de ses limitations. Nous expliquerons comment c'est devenu possible avec Vue 3 dans un prochain chapitre.

Il peut dès lors être tentant d'envelopper les propriétés primitives du composant dans un objet, et d'éviter ainsi les usages de `ref()` and `.value`. On utilise alors `reactive()` dans tous les cas. C'est un choix parfaitement valide.

Construisons un composant `Product`.

Au lieu de :

Product.vue

```
<template>
  <div>{{ price }} * {{ quantity }}</div>
</template>

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

export default defineComponent({
  name: 'Product',

  setup() {
    const price = ref(10);
    const quantity = ref(1);
    // update the price and quantity after 3 seconds
    setTimeout(() => {
      price.value = 9;
      quantity.value = 3;
    }, 3000);
    return { price, quantity };
  }
});
</script>
```

tu peux utiliser un objet enveloppant et le rendre réactif avec `reactive()` :

Product.vue

```
<template>
  <div>{{ state.price }} * {{ state.quantity }}</div>
</template>

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

export default defineComponent({
  name: 'Product',

  setup() {
    const state = reactive({
      price: 10,
      quantity: 1
    });
    // update the price and quantity after 3 seconds
    setTimeout(() => {
      state.price = 9;
      state.quantity = 3;
    }, 3000);
    return { state };
  }
});
</script>
```

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



Tu pourrais alors être tenté de déstructurer l'objet `state` pour utiliser `price` et `quantity` directement dans le template au lieu de `state.price` and `state.quantity`. Mais cela casserait la réactivité de ces propriétés, et le template ne verrait plus aucun des changements apportés à `state`.

Ainsi, ceci ne fonctionne pas :

Product.vue

```
<template>
  <div>{{ price }} * {{ quantity }}</div>
</template>

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

export default defineComponent({
  name: 'Product',

  setup() {
    const state = reactive({
      price: 10,
      quantity: 1
    });
    // update the price and quantity after 3 seconds
    setTimeout(() => {
      state.price = 9;
      state.quantity = 3;
    }, 3000);
    // destructure the state
    // to use price and quantity directly in the template
    return { ...state };
  }
});
</script>
```

Mais tout n'est pas perdu : Vue contient une fonction `toRefs` qui permet de transformer un objet réactif en références réactives :

Product.vue

```
<template>
  <div>{{ price }} * {{ quantity }}</div>
</template>
```

```

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

export default defineComponent({
  name: 'Product',

  setup() {
    const state = reactive({
      price: 10,
      quantity: 1
    });
    // update the price and quantity after 3 seconds
    setTimeout(() => {
      state.price = 9;
      state.quantity = 3;
    }, 3000);
    // use toRefs to destructure the state
    return { ...toRefs(state) };
  }
});
</script>

```

11.3. ref ou reactive

Alors lequel choisir ? Comme ils sont assez similaires, c'est, dans la plupart des cas, plutôt une affaire de goût personnel. Voici cependant quelque chose qui nous a aidé à comprendre :

```
ref(obj) ~= reactive({ value: obj })
```

Une `ref` à un objet est pratiquement la même chose qu'une propriété `reactive` contenant une `value` avec cet objet. Si c'est difficile à saisir sous cette forme, pas de souci : cela nous a pris longtemps pour comprendre nous-même 🤪.

Voici quelques exemples, qui pourront peut-être t'aider.

11.3.1. Avec des primitives

Pas besoin de réfléchir à deux fois, il faut ici une `ref` :

```

const name = ref('Cédric');
name.value = 'Cyril';

const isAdmin = ref(false);
isAdmin.value = true;

```

11.3.2. Avec des tableaux

Les tableaux fonctionnent avec les deux, mais ne peuvent être ré-assignés qu'avec une `ref`.

```
const users = ref<Array<string>>([]);  
users.value.push('Cédric');  
users.value = ['JB']; // can be re-assigned
```

Avec `reactive`:

```
const otherUsers = reactive<Array<string>>([]);  
// you don't have to write '.value' anymore  
otherUsers.push('Cédric');  
// but we can't re-assign the array
```

11.3.3. Avec des objets

C'est très similaire à ce que nous avions avec des tableaux.

```
const admin = ref({ name: 'Cédric' });  
admin.value.name = 'JB';  
admin.value = { name: 'Agnès' };
```

Avec `reactive`:

```
const otherAdmin = reactive({ name: 'Cédric' });  
// you don't have to write '.value' anymore  
otherAdmin.name = 'JB';  
// but we can't re-assign the object
```

Comme tu le vois, tu ne peux pas vraiment te tromper. Dans le futur, il y aura peut-être une recommandation officielle, mais pour l'instant, choisi celui que tu préfères 😊.

11.4. Dériver une valeur avec `computed`

Il est assez fréquent de devoir recalculer une valeur à chaque fois qu'une autre valeur change. Par exemple, dans le composant `Product` que nous venons d'utiliser, il serait pratique de calculer `total`, chaque fois que `price` ou `quantity` change.

Vue rend cela très simple grâce à la fonction `computed()`. `computed` prend une fonction `getter` en argument et renvoie une référence (en lecture seule) :

Product.vue

```
<template>
```

```

<div>{{ price }} * {{ quantity }} = {{ total }}</div>
</template>

<script lang="ts">
import { computed, defineComponent, reactive, toRefs } from 'vue';

export default defineComponent({
  name: 'Product',

  setup() {
    const state = reactive({
      price: 10,
      quantity: 1
    });

    const total = computed(() => state.price * state.quantity);

    // update the price and quantity after 3 seconds
    setTimeout(() => {
      state.price = 9;
      state.quantity = 3;
    }, 3000);

    return { ...toRefs(state), total };
  }
});
</script>

```

Vue détecte les propriétés réactives dont dépend la valeur de `total`, et les observe afin de recalculer `total`. Chaque fois que l'une d'elles change, il exécute la fonction *getter* pour assigner une nouvelle valeur à `total`.

Cela ne fonctionne qu'avec des propriétés réactives : si `state` n'était pas un objet réactif, mais un simple objet, la *computed property* ne serait jamais réévaluée.

Les propriétés calculées sont évaluées à la demande : si une propriété calculée n'est jamais affichée ou utilisée ailleurs, alors Vue ne la recalcule même pas.

11.5. Produire un effet de bord avec `watchEffect` et `watch`

Parfois, il est nécessaire de savoir quand une propriété change, mais pas nécessairement pour en calculer une autre.

Par exemple, on pourrait parfois vouloir produire un effet de bord quand une propriété change.

L'exemple typique est une page de recherche, qui doit demander de nouveaux résultats au serveur chaque fois que l'utilisateur saisit une nouvelle requête. Ou, si on reprend l'exemple du composant `Product`, nous pourrions vouloir tracer chaque changement subi par la propriété `total`.

C'est ce que permet de faire la fonction `watchEffect` :

Product.vue

```
const total = computed(() => state.price * state.quantity);

const totalHistory = ref<Array<string>>([]);

watchEffect(
  // this is the effect called when one of its dependencies changes
  () => {
    totalHistory.value.push(`Total changed to ${total.value}`);
  }
);
```

Comme tu peux le voir, on peut définir un effet de bord à exécuter, et Vue va automatiquement surveiller les dépendances de l'effet, et le ré-exécuter dès qu'une des dépendances change.

On peut également accéder à la nouvelle valeur de la propriété, mais aussi à l'ancienne si nécessaire, en utilisant `watch` :

Product.vue

```
const total = computed(() => state.price * state.quantity);

const totalHistory = ref<Array<string>>([]);

watch(
  // this is the source of the watcher
  () => total.value,
  // this is the callback called when the source changes
  (newTotal, oldTotal) => {
    totalHistory.value.push(`Total changed from ${oldTotal} to ${newTotal}`);
  },
  // a watcher with a source is lazy by default
  // whereas a watcher with just an effect and no source is not
  // but we can make it run immediately with this option
  { immediate: true }
);
```

Il est aussi possible d'observer un tableau de valeur, pour produire un effet de bord lorsqu'une quelconque des propriétés du tableau change. Note que, par défaut, `watch` est *lazy* (fainéant) et ne s'exécutera pas immédiatement, alors que `watchEffect` sera lui exécuté directement (car Vue a besoin d'exécuter l'effet afin de déterminer les dépendances réactives qu'il a besoin de surveiller, alors qu'on les définit manuellement avec `watch`).

`watch` est du coup aussi très pratique si tu as besoin de n'exécuter l'effet que pour certains changements, et non pas dès que n'importe laquelle des dépendances change (comme c'est le cas avec `watchEffect`).

Si nécessaire, tu peux arrêter le *watcher* en appelant la fonction retournée par `watch` :

Product.vue

```
const stopRecording = watchEffect(() => {
  totalHistory.value.push('Total changed to ${total.value}');
});

function stop() {
  stopRecording();
}
```

Il est aussi possible de demander au watcher de faire un peu de ménage lorsqu'il s'arrête :

Product.vue

```
const stopRecording = watchEffect(
  // the effect takes a parameter
  // which is a function called when the watcher is stopped
  onCleanup => {
    totalHistory.value.push('Total changed to ${total.value}');
    // clean the history when stop watching
    onCleanup(() => (totalHistory.value = []));
  }
);
```

11.6. Passer des `props` aux composants

Construisons un composant `Race`, qui affiche les poneys engagés dans la course.

Un tel composant peut ressembler à ça :

Race.vue

```
<template>
<div>
  <h1>{{ name }}</h1>
  <div>
    <h2>{{ pony1.name }}</h2>
    <small>{{ pony1.color }}</small>
  </div>
  <div>
    <h2>{{ pony2.name }}</h2>
    <small>{{ pony2.color }}</small>
  </div>
  <div>
    <h2>{{ pony3.name }}</h2>
    <small>{{ pony3.color }}</small>
  </div>
</div>
```

```

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

export default defineComponent({
  name: 'Race',

  setup() {
    const name = ref('Paris');
    const pony1 = reactive({
      name: 'Rainbow Dash',
      color: 'BLUE'
    });
    const pony2 = reactive({
      name: 'Pinkie Pie',
      color: 'PINK'
    });
    const pony3 = reactive({
      name: 'Fluttershy',
      color: 'YELLOW'
    });
    return { name, pony1, pony2, pony3 };
  }
});
</script>

```

Le template est plein de copié-collés... Si on décide de changer la manière dont on affiche la couleur d'un poney, il faut penser à le faire 3 fois. Dans une application plus complexe, tu comprends vite que la situation est loin d'être idéale.

Tu m'objecteras sans doute qu'il suffirait d'utiliser un tableau de poneys, et de les afficher tous les trois avec `v-for`. Et tu aurais raison. Mais supposons maintenant qu'il faille afficher le gagnant de la course en haut du template, dans une section à part ? `v-for` ne nous est plus daucun secours ici. Il nous faut donc une meilleure solution pour garder notre code *DRY*.

Mais nous avons appris qu'on pouvait utiliser un composant à l'intérieur d'un autre composant. Extraire un plus petit composant réutilisable est la bonne manière de résoudre ce problème. On comprend vite ici qu'un composant `Pony` nous sauverait. Une fois créé, on n'aurait plus qu'à utiliser un élément `<Pony />` dans le template du composant `Race` chaque fois qu'on veut afficher un poney, et donc éviter les copié-collés :

Race.vue

```

<template>
<div>
  <h1>{{ name }}</h1>
  <Pony />
  <Pony />
  <Pony />
</div>

```

```
</template>
```

Mais... comment dire à chaque poney quel nom et quelle couleur il doit afficher ? ☺

C'est ici que les **props** entrent en jeu !

Dans un composant, tu peux définir des **props** afin de pouvoir recevoir des données du composant parent. Ces **props** deviennent partie intégrante de l'état du composant.

Ainsi, le composant **Pony** devrait ressembler à cela :

Pony.vue

```
<template>
  <div>
    <h2>{{ name }}</h2>
    <small>{{ color }}</small>
  </div>
</template>

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

  export default defineComponent({
    name: 'Pony',

    props: {
      name: String,
      color: String
    }
  });
</script>
```

Comme tu peux le voir, l'objet définissant le composant a une propriété **props**, qui est un objet définissant tout ce que le composant accepte de recevoir de son parent. Pour chaque *prop* , on spécifie le type de la propriété. Bizarrement, on ne peut pas utiliser les types TypeScript ici : cette API existe depuis longtemps, avant que le support de TypeScript soit introduit, et Vue ne supporte que les "types" JavaScript : **String**, **Number**, **Boolean**, **Date**, **Function**, **Symbol**, **Object** et **Array**. Nous verrons plus tard comment améliorer cela.

À l'exécution, Vue vérifie les valeurs passées par le parent pour chaque *prop* . Si une valeur ne correspond pas au type attendu, La console affiche un avertissement comme celui-ci :

```
'[Vue warn]: Invalid prop: type check failed for prop "ponyModel". Expected Object, got Number with value 2.'
```

Si tu examines le code source du composant **Pony**, tu verras qu'on peut utiliser les **props** directement dans le template.

Maintenant, comment le composant parent peut-il passer des valeurs au composant `Pony`? Revenons au composant `Race`: on peut utiliser simplement `<Pony name="Rainbow Dash" color="BLUE" />` si les valeurs des `props` sont statiques. Si elles sont dynamiques :

Race.vue

```
<template>
  <div>
    <h1>{{ name }}</h1>
    <Pony :name="pony1.name" :color="pony1.color" />
    <Pony :name="pony2.name" :color="pony2.color" />
    <Pony :name="pony3.name" :color="pony3.color" />
  </div>
</template>
```



Si tu veux passer un nombre, un booléen, un tableau, etc., tu *dois* utiliser `v-bind`: même pour une valeur statique. Autrement, Vue passera toujours une string : `<Pony speed="16" isRunning="false" />` passerait "16" pour la vitesse et "false" pour le booléen `isRunning`, au lieu de 16 et `false`. Pour passer le booléen `true` en revanche, il suffit de ne pas spécifier de valeur pour l'attribut : `<Pony isRunning />`

La situation est déjà bien meilleure, mais on peut faire mieux. Au lieu de passer chaque valeur individuellement au composant `Pony`, on peut passer directement l'objet contenant toutes ces valeurs :

Race.vue

```
<template>
  <div>
    <h1>{{ name }}</h1>
    <Pony :ponyModel="pony1" />
    <Pony :ponyModel="pony2" />
    <Pony :ponyModel="pony3" />
  </div>
</template>
```

Il nous suffit pour cela de spécifier une unique `prop` dans le composant `Pony`:

Pony.vue

```
import { defineComponent } from 'vue';

export default defineComponent({
  name: 'Pony',

  props: {
    ponyModel: Object
  }
})
```

```
});
```

C'est pratique et logique, mais on perd ainsi une information très utile : le type de l'objet. On *sait* que `ponyModel` n'est pas n'importe quel objet, mais qu'il s'agit d'une entité bien précise de notre application, définie par l'interface `PonyModel` :

PonyModel.ts

```
export interface PonyModel {  
    id: number;  
    name: string;  
    color: string;  
}
```

Pour résoudre ce problème de typage, on peut utiliser le type `PropType<T>` pour spécifier le type de la *prop* :

Pony.vue

```
import { defineComponent, PropType } from 'vue';  
import { PonyModel } from '@/models/PonyModel';  
  
export default defineComponent({  
    name: 'Pony',  
  
    props: {  
        ponyModel: Object as PropType<PonyModel>  
    }  
});
```

À présent que cette *prop* est correctement typée, il est beaucoup plus simple et robuste de l'utiliser dans le code du composant, comme on va le voir bientôt.

11.6.1. Props obligatoires

Vue permet de définir qu'une *prop* est obligatoire :

Pony.vue

```
props: {  
    ponyModel: {  
        type: Object as PropType<PonyModel>,  
        required: true  
    }  
}
```

La déclaration de la *prop* est un peu plus verbeuse : au lieu d'utiliser `prop: type` on utilise à présent `prop: { type: type, required: true }`. En marquant la *prop* obligatoire, on demande à Vue de faire

une vérification supplémentaire à l'exécution, lorsque le composant est utilisé, et de lever une erreur pour signaler qu'on a oublié de passer une valeur requise. Le message d'erreur prend cette forme :

```
'[Vue warn]: Missing required prop: "ponyModel"'
```

11.6.2. Props avec des valeurs par défaut

On peut aussi donner une valeur par défaut à une *prop* :

Pony.vue

```
props: {
  ponyModel: {
    type: Object as PropType<PonyModel>,
    default: () => ({
      name: 'Rainbow Dash',
      color: 'BLUE'
    })
  }
}
```

Comme cette *prop* est un objet, il faut en réalité fournir une fonction de fabrique comme valeur par défaut. Pour une valeur primitive comme un nombre, on utiliserait simplement `default: 42` par exemple.

11.6.3. Props avec validateurs

Dernière possibilité, mais pas la moindre : on peut définir un validateur pour une *prop*. Par exemple, pour s'assurer qu'une valeur supérieure à 5 est toujours passée pour la *prop* `speed` :

Pony.vue

```
speed: {
  type: Number,
  validator: (speed: number) => speed > 5
}
```

À l'exécution, Vue valide que la *prop* obéit bien à la règle de validation, et émet un avertissement si ce n'est pas le cas :

```
'[Vue warn]: Invalid prop: custom validator check failed for prop "speed".'
```



Toutes ces vérifications (typage, présence obligatoire et validation) sont omises quand l'application est exécutée en mode production.

11.6.4. Utiliser des props dans la fonction `setup`

Nous avons vu que les `props` pouvaient être utilisées dans le template, mais comment les utiliser dans la fonction `setup()` dont nous avons parlé plus tôt ?

Eh bien en fait, le premier paramètre de la fonction `setup()` peut être `props` ! Et on peut alors utiliser les `props` comme n'importe quelle autre propriété réactive :

Pony.vue

```
props: {
  ponyModel: {
    type: Object as PropType<PonyModel>,
    required: true
  }
},

setup(props) {
  const ponyImageUrl = computed(() => `images/pony-${props.ponyModel.color.
toLowerCase()}.gif`);
  return { ponyImageUrl };
}
```

Dans cet exemple, on définit une *computed property* `ponyImageUrl` qui se base sur la `color` de la *prop* `ponyModel`. Chaque fois que la valeur de la *prop* change, l'URL de l'image change aussi.

Et comme on a spécifié le type de la *prop*, le compilateur TypeScript sait que `props.ponyModel` est de type `PonyModel`. 🎉

Note que les `props` sont en lecture seule : tu ne peux pas les modifier dans le composant qui reçoit la *prop*. En revanche, le composant parent, qui fournit la *prop*, peut en changer sa valeur, bien sûr. Alors, comment faire pour modifier une *prop* qui nous est fournie par le parent ? On utilise un événement !

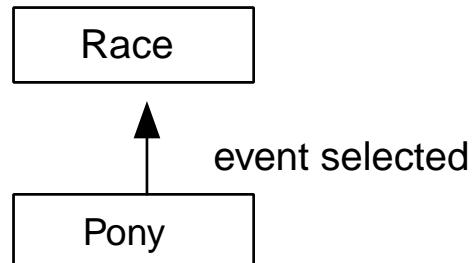
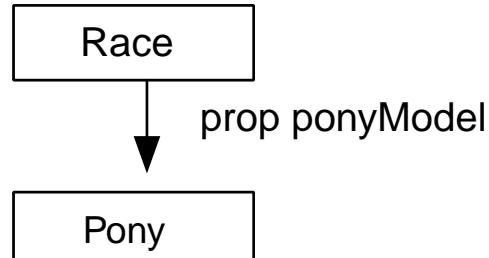


Essaye nos exercices [Détail d'une course](#) 🐾 et [Propriétés calculées](#) 🐾 ! Ces exercices vont te guider dans la construction d'un composant avancé avec des `props` et *computed properties*.

11.7. Des événements spécifiques avec `emit`

L'objet `props` ne peut pas être modifié par le composant enfant, qui le reçoit. Tu *peux* modifier le *contenu* d'une *prop* si c'est un objet ou un tableau, mais c'est considéré comme une mauvaise pratique. Lorsqu'un composant veut notifier son parent que quelque chose doit être changé, cependant, il peut émettre un événement. Le parent peut écouter cet événement et mettre à jour son état en conséquence.

Par exemple, le composant `Pony` peut recevoir une *prop* `ponyModel`, et peut émettre un événement `selected`.



Comment émet-on un événement ? Il s'avère que `setup(props)` peut recevoir un deuxième paramètre, le `context: setup(props, context)`. `context` est un objet qui contient plusieurs éléments, sur lesquels nous reviendrons plus tard. Pour l'instant, ce qui nous importe, c'est qu'il contient une fonction `emit`. Traditionnellement, on y accède en utilisant la déstructuration :

Pony.vue

```

<template>
  <div @click="selectPony()">
    <h2 :style="{ color: ponyModel.color }>{{ ponyModel.name }}</h2>
  </div>
</template>

<script lang="ts">
import { defineComponent, PropType } from 'vue';
import { PonyModel } from '@/models/PonyModel';

export default defineComponent({
  name: 'Pony',

  props: {
    ponyModel: {
      type: Object as PropType<PonyModel>,
      required: true
    }
  },

```

```

emits: {
  // We declare that the only event emitted by the component will be 'selected'
  // and that the emitted value is of type 'PonyModel'
  // The function is a validator of the event argument, checked by Vue in
development mode
  selected: (pony: PonyModel) => pony.color !== 'PURPLE'
},

setup(props, { emit }) {
  function selectPony() {
    // we emit a 'selected' event containing the pony entity
    emit('selected', props.ponyModel);
  }

  return { selectPony };
}
);
</script>

```



La propriété `emits` est optionnelle, mais pratique. À la compilation, TypeScript vérifie que tu n'émetts que des événements déclarés dans la propriété `emits` et que la valeur de l'événement est du type adéquat. Dans cet exemple, la compilation échouerait si tu tentais d'émettre un événement qui n'est pas nommé `selected`, ou qui n'est pas un `PonyModel`. À l'exécution, mais seulement pendant le développement, Vue vérifie aussi que le prédictat fourni accepte les événements émis. Si ce n'est pas le cas, l'événement sera émis, mais tu verras un avertissement dans la console. Dans cet exemple, un avertissement sera donc affiché si tu émets un poney violet. Si tu ne veux pas valider la valeur de l'événement, il suffit de retourner `true` dans cette fonction prédictat.

Comme tu peux le voir, `emit` nous permet de passer le nom et la valeur (le *payload*) de l'événement à émettre. Il ne nous reste plus qu'à écouter cet événement dans le template du composant parent, et à obtenir sa valeur si nécessaire en utilisant `$event`, exactement comme on le fait avec des événements standards du DOM.

Le composant `Race` peut changer la couleur du poney lorsqu'il est sélectionné, par exemple :

Race.vue

```
<Pony :ponyModel="pony" @selected="changeColor(pony)" />
```

avec la fonction `changeColor` suivante, qui assigne une couleur aléatoire au poney :

Race.vue

```

function changeColor(pony: PonyModel) {
  pony.color = randomColor();
}

```

```
}
```

Maintenant, chaque fois que les utilisateurs cliquent sur un poney, il émet un événement `selected`, auquel le composant `Race` réagit en appelant `changeColor()`. La couleur du poney sélectionné change, et comme c'est une *prop* du composant `Pony`, le composant se rafraîchit et affiche le poney dans sa nouvelle couleur !



Essaye notre exercice [Composant Pony 🐴](#) pour construire un nouveau composant avec des événements spécifiques.

11.8. Fonctions du cycle de vie

Très souvent, il est nécessaire de faire un peu de travail lorsque le composant est initialisé. La fonction `setup` est appelée lorsque le composant est créé, et définit la majeure partie du processus d'initialisation. Comment alors appeler une API afin de peupler une propriété après l'affichage du template par exemple ? Les fonctions de cycle de vie peuvent aider dans ce cas. On les utilise à l'intérieur de la fonction `setup`, toutes de la même manière : elles acceptent une fonction callback, qui peut être `async`, et qui sera appelée par Vue à un moment précis du cycle de vie du composant. Nous verrons plus tard que la fonction `setup` elle-même peut être asynchrone dans un cas précis.

Les principales fonctions, correspondant chacune à un moment de la vie du composant, sont :

- `onBeforeMount/onMounted`, appelée avant/après que le composant a été "monté", c'est-à-dire inséré dans le DOM (y'a probablement une blague avec "monter le poney", mais restons concentrés) ;
- `onBeforeUpdate/onUpdated`, appelée avant/après que le composant a été modifié (quelque chose a changé, et le DOM a été mis à jour) ;
- `onBeforeUnmount/onUnmounted`, appelée avant/après que le composant a été détruit (enlevé du DOM)
- `onErrorCaptured` appelée si une erreur survient dans l'un des composants descendants.

Voyons quelques exemples.

11.8.1. Initialiser un composant avec `onMounted()`

Un usage très fréquent de `onMounted` est d'appeler une API asynchrone afin d'initialiser une propriété réactive du composant. Si `fetchRace()` est une fonction asynchrone (et c'est le cas, puisqu'elle fait un appel HTTP au backend pour obtenir une course), alors on peut écrire :

Race.vue

```
props: {  
  raceId: {  
    type: Number,  
    required: true  
  }  
},
```

```

setup(props) {
  const race = ref<RaceModel | null>(null);

  // we fetch the details of the race based on its ID
  onMounted(async () => {
    race.value = await fetchRace(props.raceId);
  });

  return { race };
}

```

onMounted peut aussi être utile si on veut manipuler le DOM créé pour le composant, comme nous le verrons plus tard.

11.8.2. Faire le ménage avec **onUnmounted()**

Une source fréquente de fuites mémoires dans les applications est d'oublier d'arrêter des tâches récurrentes, qui tournent en tâche de fond.

Imaginons que nous voulions que le composant **Race** rafraîchisse la course affichée toutes les 10 secondes. Nous pourrions écrire :

Race.vue

```

setup(props) {
  const race = ref<RaceModel | null>(null);

  onMounted(( ) => {
    // we fetch the details of the race based on its ID every 10s
    window.setInterval(async () => (race.value = await fetchRace(props.raceId)),
10000);
  });

  return { race };
}

```

Mais nous venons d'introduire un bug : chaque fois que le composant est instancié, il démarre une nouvelle tâche de fond, qui ne s'arrête jamais...

Au bout d'un moment, si les utilisateurs naviguent vers ce composant, puis vers une autre page de l'application, puis de nouveau vers ce composant, et ainsi de suite, le serveur va finir par s'écrouler sous le poids des requêtes incessantes de mise à jour, et le navigateur va consommer bien plus de mémoire et de bande passante que nécessaire.

On peut arrêter le processus avec **onUnmounted** :

Race.vue

```

setup(props) {

```

```
const race = ref<RaceModel | null>(null);
let intervalRef: number | undefined;

onMounted(() => {
  intervalRef =
    // we fetch the details of the race based on its ID every 10s
    window.setInterval(async () => (race.value = await fetchRace(props.raceId)), 10000);
});

onUnmounted(() => {
  //we cancel the timeout
  window.clearInterval(intervalRef);
});

return { race };
}
```



Essaye notre [quiz](#) 🎯 pour voir si tu as tout compris !

Chapter 12. API Composition

Nous avons déjà introduit l'API Composition dans notre précédent chapitre sur la construction des composants.

Mais nous n'avons pas réellement expliqué en quoi elle permettait d'améliorer la conception de nos composants. C'est ce que nous allons faire maintenant. Et ensuite, nous verrons comment cette API permet aussi de partager du code entre plusieurs composants.

12.1. Une conception propre grâce à l'API Composition

Imaginons un composant `Order` permettant à un utilisateur de commander un produit avec une quantité donnée. Nous voulons afficher les informations concernant le produit, ainsi que le prix total de la commande. Nous voulons aussi permettre à l'utilisateur d'augmenter ou de baisser la quantité. Et enfin, nous voulons tracer quand un composant est affiché et quand il est détruit, pour des raisons analytiques.

Un tel composant peut ressembler à cela :

Order.vue

```
setup(props) {
  const product = ref<ProductModel | null>(null);
  const quantity = ref(0);
  const price = computed(() => quantity.value * (product.value?.unitPrice ?? 0));

  onMounted(async () => {
    // log a trace that the user entered this page
    await trace('enter');
  });

  // fetch the product data on initialization and refresh them if the props change
  watchEffect(async () => {
    product.value = await getProduct(props.productId);
  });

  function increaseQuantity() {
    quantity.value += 1;
  }

  function decreaseQuantity() {
    if (quantity.value > 0) {
      quantity.value -= 1;
    }
  }

  async function order() {
    // place the order
  }
}
```

```

onUnmounted(async () => {
  // log a trace that the user left this page
  await trace('exit');
});

return { product, quantity, price, increaseQuantity, decreaseQuantity, order };
}

```

Le code est compréhensible, mais les différents aspects sont tous mélangés les uns avec les autres.

Dans Vue 2.x, cela aurait été pire. En effet, l'API Composition, a été introduite avec Vue 3.0. Auparavant, c'était l'API Options qu'il fallait utiliser (on peut toujours l'utiliser d'ailleurs), et elle nécessitait de déclarer plusieurs options :

- les propriétés réactives dans l'option `data` ;
- les méthodes du composant dans l'option `methods` ;
- les *watchers* dans l'option `watch` ;
- les propriétés calculées dans l'option `computed` ;
- les *hooks* chacun dans une option (`mounted`, `destroyed`, etc.)

Ainsi, pour un composant complexe, la logique se retrouvait éclatée entre toutes ces différentes options.

L'un des aspects intéressants de l'API Composition est qu'elle permet de grouper tout le code concernant un aspect fonctionnel à un seul endroit, séparé du code concernant les autres aspects. Et on peut même extraire du code partagé *en dehors* des composants.

Refactorisons notre composant `Order`, en extrayant une fonction séparée pour chaque aspect.

Vue propose une convention que tu peux respecter : implémenter une fonction `use…` qui retourne un objet contenant les différentes propriétés réactives et fonctions liées à un aspect donné. Nous verrons dans les prochains chapitres que de nombreuses bibliothèques utilisent cette convention. Le routeur, par exemple, expose une fonction `useRouter`.

Dans notre composant, nous pouvons donc commencer par extraire une fonction `useTracking`, hors du composant, dédiée au traçage de l'utilisation du composant :

Order.vue

```

function useTracking() {
  onMounted(() => {
    // log a trace that the user entered this page
    trace('enter');
  });

  onUnmounted(() => {
    // log a trace that the user left this page
    trace('exit');
}

```

```
});  
}
```

Extrayons ensuite une fonction `useProduct`, contenant la logique relative au produit :

Order.vue

```
function useProduct(productId: Ref<number>) {  
  const product = ref<ProductModel | null>(null);  
  // fetch the product data on initialization and refresh them if the props change  
  watchEffect(async () => {  
    product.value = await getProduc(productId.value);  
  });  
  return product;  
}
```

Et enfin une fonction `useOrderService`, pour tout ce qui concerne la commande, sa quantité et son prix :

Order.vue

```
function useOrderService(product: Ref<ProductModel | null>) {  
  const quantity = ref(0);  
  const price = computed(() => quantity.value * (product.value?.unitPrice ?? 0));  
  
  function increaseQuantity() {  
    quantity.value += 1;  
  }  
  
  function decreaseQuantity() {  
    if (quantity.value > 0) {  
      quantity.value -= 1;  
    }  
  }  
  
  async function order() {  
    // place the order  
  }  
  
  return { quantity, price, increaseQuantity, decreaseQuantity, order };  
}
```

Chacune des fonctions extraites est plus simple et lisible, et le code du composant se résume à présent à :

Order.vue

```
setup(props) {  
  useTracking();
```

```

// we need `toRefs` to destructure the props
// and keep the reactivity
const { productId } = toRefs(props);
const product = useProduct(productId);

const orderService = useOrderService(product);

return { product, ...orderService };
}

```

Le code du composant est donc bien plus concis. Il est plus simple de voir d'un seul coup d'œil ce qui concerne une fonctionnalité particulière.

Et, cerise sur le gâteau, ces fonctions `use...`, que l'on appelle souvent des *composables*, peuvent être partagées entre composants.

12.2. Extraire de la logique commune

Assez souvent dans une application, plusieurs composants devront partager de la logique commune, par exemple appeler la même API HTTP.

Au lieu de dupliquer le code, on peut l'extraire dans un fichier séparé. Par exemple, si plusieurs composants doivent appeler des fonctions relatives à l'utilisateur telles que `authenticate`, `logout`, `register`, etc. on peut placer toutes ces fonctions dans un fichier dédié.

```

import { UserModel } from '@/models/UserModel';

export function authenticate(login: string, password: string): Promise<UserModel> {
    // call the HTTP API
}

export function logout(): void {
    // ...
}

// ...

```

Pour tirer profit de l'API Composition, nous devrions retourner ces fonctions en tant que propriétés d'un objet, retourné par une fonction `useUserService` :

UserService.ts

```

import { UserModel } from '@/models/UserModel';

async function authenticate(login: string, password: string): Promise<UserModel> {
    // call the HTTP API
}

```

```

function logout(): void {
    // ...
}

// ...

export function useUserService() {
    return {
        authenticate,
        logout
    };
}

```

On peut ensuite utiliser ce *composable* dans un composant comme suit :

Login.vue

```

setup() {
    const credentials = {
        login: '',
        password: ''
    };
    const userService = useUserService();
    async function login() {
        await userService.authenticate(credentials.login, credentials.password);
    }
    return { credentials, login };
}

```

12.3. L'API Composition hors des composants

Jusqu'ici, envelopper les fonctions de gestion de l'utilisateur dans un objet retourné par `useUserService()` n'apporte pas grand-chose, parce que cet objet n'a aucun état, ni aucune logique liée à un quelconque composant. Mais il serait aussi intéressant de retourner, en plus des seules fonctions, l'utilisateur authentifié : plusieurs composants peuvent avoir besoin de l'afficher ou de connaître ses propriétés.

Comme l'API Composition n'est pas liée aux composants, on peut l'utiliser directement dans la fonction `useUserService` :

UserService.ts

```

import { ref } from 'vue';
import { UserModel } from '@/models/UserModel';

const userModel = ref<UserModel | null>(null);

async function authenticate(login: string, password: string): Promise<UserModel> {

```

```

    // call the HTTP API
    // in case of success, store the logged-in user
    userModel.value = response.data;
}

function logout(): void {
    // ...
    userModel.value = null;
}

// ...

export function useUserService() {
    return {
        userModel,
        authenticate,
        logout
    };
}

```

À présent, tout composant peut accéder à l'utilisateur courant et l'utiliser dans son template directement, puisque c'est une valeur réactive !

Home.vue

```

<template>
    <div v-if="userModel">Hello {{ userModel.name }}!</div>
    <div v-else>Welcome, anonymous comrade!</div>
</template>

<script lang="ts">
export default defineComponent({
    name: 'Home',

    setup() {
        const { userModel } = useUserService();
        return { userModel };
    }
});
</script>

```

Revenons à notre cas d'utilisation de traçage de l'utilisation d'un composant : on voudrait tracer que l'utilisateur entre ou sort d'un composant.

On peut bien sûr gérer ça dans chaque composant en y définissant les *hooks* `onMounted` et `onUnmounted` :

Home.vue

```
<template>
  <div v-if="userModel">Hello {{ userModel.name }}!</div>
  <div v-else>Welcome</div>
</template>

<script lang="ts">
import { defineComponent, onMounted, onUnmounted } from 'vue';
import { useUserService } from './UserService';

export default defineComponent({
  name: 'Home',

  setup() {
    const { userModel, trace } = useUserService();
    onMounted(() => trace('home:enter'));
    onUnmounted(() => trace('home:exit'));
    return { userModel };
  }
});
</script>
```

avec la fonction `trace` définie comme suit :

UserService.ts

```
function trace(event: string): void {
  // call the tracing HTTP API with userModel.id
  // and the event
}

export function useUserService() {
  return {
    userModel,
    trace,
    authenticate,
    logout
  };
}
```

Mais c'est un peu pénible de devoir ajouter ces *hooks* dans chaque composant. On peut faire plus élégant : déplacer les appels à `onMounted` et `onUnmounted` dans `useUserService` :

UserService.ts

```
export function useUserService(view?: string) {
  if (view) {
    onMounted(() => trace(`#${view}:enter`));
    onUnmounted(() => trace(`#${view}:exit`));
```

```

    }
    return {
      userModel,
      authenticate,
      logout
    };
}

```

Ce simple changement permet de faire en sorte que, chaque fois qu'un composant appelle `useUserService` et lui passe un nom de vue, les appels à `trace` sont faits automatiquement ! C'est pas beau ça ?

Home.vue

```

export default defineComponent({
  name: 'Home',

  setup() {
    const { userModel } = useUserService('home');
    return { userModel };
  }
});

```

Pour résumer, l'API Composition permet d'extraire de la logique dans une fonction `use…` qui retourne un objet. Cet objet peut contenir des fonctions (synchrone ou asynchrone) mais aussi des propriétés réactives (avec `ref` ou `reactive`). Les fonctions `use…` permettent aussi de définir des *watchers*, des propriétés calculées ou des *hooks* de cycle de vie. Elles peuvent être utilisées pour améliorer la lisibilité d'un composant en centralisant les parties du code concernant un aspect spécifique, ou être utilisées pour partager du code entre plusieurs composants et les faire communiquer entre eux.



Essaie l'exercice [Lifecycle hooks](#) pour créer ta première fonction `use…` et mettre en application le hook `onMounted`.

12.4. Un exemple de la communauté : VueUse

Un des trucs cools avec ce pattern est que ces *composables* peuvent être extraits et réutilisés. [VueUse](#) est une bibliothèque open-source de la communauté Vue qui offre des *tonnes* de fonctions utilitaires.

Voici quelques-unes de mes préférées :

- `useTitle` : pour changer le titre d'une page
- `useGeolocation` : pour récupérer la position d'un utilisateur
- `useQRCode` : pour générer des QR codes
- `useBreakpoints` : pour être notifié d'un changement de la taille de l'écran dans ton code (comme le fait une media query en CSS)

- **useClipboard** : pour lire ou écrire du contenu dans le copier/coller

VueUse.vue

```
<template>
  <div>{{ coords.latitude }} - {{ coords.longitude }}</div>
  
</template>

<script lang="ts">
import { defineComponent } from 'vue';
import { useTitle, useGeolocation } from '@vueuse/core';
import { useQRCode } from '@vueuse/integrations/useQRCode';

export default defineComponent({
  name: 'VueUse',
  setup() {
    // change the title
    useTitle('VueUse is awesome');
    // get the position of the user
    const { coords } = useGeolocation();
    // generate a QR code for the ebook
    const qrcode = useQRCode('https://books.ninja-squad.com/vue');
    return { coords, qrcode };
  }
});
</script>
```

Il y a aussi des intégrations avec firebase, axios, etc. Plein de petites pépites auxquelles penser la prochaine que tu as besoin de quelque chose !

Chapter 13. Style des composants

Faisons une pause pour parler un peu style et CSS quelques instants. Oh noooon, mais pourquoi parler de CSS ?! Parce que Vue (et Vite ou la CLI) fait beaucoup pour nous sur ce sujet.

En tant que développeur web, tu ajoutes souvent des classes CSS sur tes éléments. Et l'essence même de CSS, c'est que ce style va *cascader* (rappelons que CSS signifie *Cascading Style Sheet*). C'est fréquemment un comportement voulu (pour, par exemple, changer la police globalement dans toute l'application), mais parfois non. Imaginons que tu souhaites ajouter un style sur l'élément sélectionné d'une liste : tu vas probablement utiliser un sélecteur CSS très restreint, comme `li.selected`. Ou encore plus restreint, en utilisant une convention comme `BEM`, parce que tu veux styler l'élément sélectionné dans une partie bien spécifique de ton application.

Les styles que tu définis dans un composant (dans la partie `style` d'un SFC) sont par défaut appliqués globalement.

Et sur ce point Vite et la CLI peut se rendre utile. Les styles peuvent s'appliquer au composant courant et seulement à celui-ci. Mais comment est-ce possible ?

13.1. Styles `scoped`

Tout commence par l'écriture d'un style. Prenons par exemple notre composant `Pony`, que tu dois commencer à bien connaître. C'est une version simplifiée, affichant uniquement l'image du poney et son nom dans un élément `span`. Pour l'exemple, ajoutons une classe CSS `highlight` à ce `span` :

```
<div>
  
  <span class="highlight">{{ ponyModel.name }}</span>
</div>
```

Cette classe est définie dans les styles du composant, mais note que nous avons ajouté `scoped` à l'élément `style` :

```
<style scoped>
  .highlight {
    color: red;
  }
</style>
```

Comme tu le vois, on veut afficher le nom du poney en rouge.

Pour être sûr que notre style ne s'applique qu'au composant `Pony`, Vue va *inliner* le CSS dans le `<head>` de la page. Mais avant de l'*inliner*, Vue va réécrire le sélecteur CSS, pour lui ajouter un attribut unique. Cet attribut unique sera ensuite ajouté sur tous les éléments du template du composant ! Ainsi, le style s'appliquera uniquement à notre composant :

```

<html>
  <head>
    <style>.highlight[data-v-0a17a18c] {color: red}</style>
  </head>
  <body>
    ...
    <Pony>
      <div data-v-0a17a18c="">
        
        <span class="highlight" data-v-0a17a18c="">Rainbow Dash</span>
      </div>
    </Pony>
  </body>
</html>

```

Le sélecteur de classe `highlight` a été réécrit en `.highlight[data-v-0a17a18c]`, alors il ne s'appliquera que sur les éléments qui auront à la fois la classe `highlight` et l'attribut `data-v-0a17a18c`. Tu peux aussi vérifier que cet attribut a bien été ajouté automatiquement à notre `span`, pour que tout fonctionne parfaitement.

Cette fonctionnalité est très pratique. Cela permet par exemple d'utiliser des composants fournis par une librairie tierce, sans avoir à se soucier de potentiels conflits entre nos classes CSS et les leurs !

13.2. Styles module

Vue supporte aussi les styles `module`. Cette fonctionnalité est basée sur la spécification [CSS Modules](#) et permet d'utiliser `$style` dans ton template et dans ton code ! C'est une solution assez populaire dans certains écosystèmes (comme React), mais je ne l'ai jamais utilisée 🤔.

13.3. v-bind en CSS

Tu peux aussi directement utiliser `v-bind` dans le CSS pour lier une valeur CSS à une propriété de ton composant :

```

<script setup lang="ts">
  import { ref } from 'vue';
  const ponyModel = ref({
    name: 'Rainbow Dash',
    imageUrl: '',
    color: 'blue'
  });
</script>

<style scoped>
.highlight {
  color: v-bind('ponyModel.color');
}

```

```
}
```

```
</style>
```

C'est bien sûr réactif : si `ponyModel.color` change, alors la couleur se met à jour.

13.4. `v-deep`, `v-global` et `v-slotted`

Les styles `scoped` dans un SFC supportent aussi des extensions CSS spécifiques pour certains cas très précis.

Par exemple, on veut parfois ajouter un style explicitement à un composant enfant. Mais comment pouvons-nous faire ça, alors qu'un style `scoped` ne s'applique par essence qu'au composant courant ?

C'est là qu'entre en jeu `::v-deep()`, ou `:deep()` en raccourci :

```
/* deep selectors */
::v-deep(.foo) {}
/* shorthand */
:deep(.foo) {}
```

Au contraire, si tu veux rendre une règle globale, tu peux utiliser `::v-global()` ou `:global()` :

```
/* one-off global rule */
::v-global(.foo) {}
/* shorthand */
:global(.foo) {}
```

La dernière extension permet de cibler le contenu des slots. En Vue 3, le contenu des slots n'est pas modifié par les styles `scoped`. Si tu veux viser explicitement le contenu des slots, tu peux utiliser `::v-slotted()` ou `:slotted()` :

```
/* targeting slot content */
::v-slotted(.foo) {}
/* shorthand */
:slotted(.foo) {}
```

13.5. PostCSS

Vite et Vue CLI utilisent [PostCSS](#) sous le capot pour transformer notre CSS.

Mais ici, ils diffèrent légèrement : Vite considère que tu cibles des navigateurs modernes, et ne fait donc que peu de transformation par défaut. Mais tu peux lui fournir une configuration PostCSS si nécessaire, et elle sera appliquée automatiquement.

13.6. Pre-processeurs CSS

CSS s'améliore d'année en année. Il est maintenant possible de faire des variables CSS par exemple. C'est pour cela que Vite recommande de s'en tenir au CSS standard.

Mais si tu veux utiliser plus que du CSS brut, alors Vite et la CLI ont aussi un support natif des pré-processeurs CSS les plus courants :

- SCSS
- Less
- Stylus

Tu as simplement besoin d'installer le pré-processeur dont tu as besoin, (avec son "loader" Webpack si tu utilises la CLI) :

```
npm install --save-dev sass
```

Tu peux ensuite utiliser l'attribut `lang` sur l'élément `style` :

```
<style lang="scss">  
</style>
```

Et la CLI s'occupe du reste.

Pour résumer, Vite ou la CLI s'avèrent très précieux pour la partie CSS d'une application. Tu voudras probablement utiliser des styles `scoped` la plupart du temps et Vue fera marcher sa magie !

Chapter 14. Les nombreuses façons de définir des composants

Comme nous le disions dans le chapitre précédent, il y a plusieurs façons de définir des composants depuis Vue 3.

Ce livre utilise l'API de Composition, introduite en Vue 3, mais tu as peut-être déjà croisé l'API Options, utilisée en Vue 1 et 2, d'ailleurs toujours utilisable en Vue 3. Vue 3 offre aussi une autre approche, inspirée par le framework Svelte. Passons en revue ces différentes alternatives.

14.1. API Options

C'est la façon historique de construire des composants Vue. Tu utilises un objet, dans lequel tu définis les données réactives avec `data`, les méthodes avec `methods`, les données calculées avec `computed`, etc. C'est tout à fait utilisable et toujours supporté par Vue 3, mais principalement par souci de compatibilité. Elle n'offre pas la même flexibilité que l'API de Composition, et impose ainsi aux développeurs d'utiliser les Mixins pour partager du comportement.

Product.vue

```
<template>
  <div>{{ price }} * {{ quantity }} = {{ total }}</div>
  <CustomButton @click="addOne()">Add 1 unit</CustomButton>
</template>

<script lang="ts">
import { defineComponent } from 'vue';
import CustomButton from '@/chapters/many-ways/CustomButton.vue';

export default defineComponent({
  name: 'Product',
  components: {
    CustomButton
  },
  data() {
    return {
      price: 10,
      quantity: 1
    };
  },
  computed: {
    total(): number {
      return this.price * this.quantity;
    }
  },
  methods: {
    addOne(): void {
      this.quantity++;
    }
  }
})
```

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

14.2. API de Composition

Vue 3 a introduit l'API de Composition après une longue discussion et réflexion avec la communauté. L'idée est de fournir une API flexible avec un excellent support TypeScript. Cette API concentre la majorité du travail dans la propriété `setup` de la définition du composant.

Product.vue

```
<template>
  <div>{{ price }} * {{ quantity }} = {{ total }}</div>
  <CustomButton @click="addOne()">Add 1 unit</CustomButton>
</template>

<script lang="ts">
  import { computed, defineComponent, ref } from 'vue';
  import CustomButton from '@/chapters/many-ways/CustomButton.vue';

  export default defineComponent({
    name: 'Product',

    components: {
      CustomButton
    },

    setup() {
      const price = ref(10);
      const quantity = ref(1);
      const total = computed(() => price.value * quantity.value);
      const addOne = () => quantity.value++;
      return { price, quantity, total, addOne };
    }
  });
</script>
```

14.3. Script setup

Vue 3.2 introduit une autre façon de déclarer un composant, basée sur l'API de Composition, mais légèrement moins verbeuse, en ajoutant un attribut `setup` à l'élément `script` de ton SFC :

Product.vue

```
<template>
  <div>{{ price }} * {{ quantity }} = {{ total }}</div>
```

```

<CustomButton @click="addOne()">Add 1 unit</CustomButton>
</template>

<script setup lang="ts">
import { computed, ref } from 'vue';
import CustomButton from '@/chapters/many-ways/CustomButton.vue';
const price = ref(10);
const quantity = ref(1);
const total = computed(() => price.value * quantity.value);
const addOne = () => quantity.value++;
</script>

```

Comme tu peux le voir, cette syntaxe enlève pas mal de *boilerplate*, car tout ce qui dans la section `script` est automatiquement exposé dans le template. Cela s'inspire fortement de ce que fait le framework Svelte, et c'est la façon préférée de définir des composants actuellement. C'est la version que nous utiliserons dans ce livre et les exercices à partir de maintenant. On se penche plus en détails sur cette syntaxe dans le chapitre suivant : finissons notre tour avant.

14.4. API Classe

Vue 2 avait une API basée sur les classes pour définir les composants, qui était la meilleure solution quand on voulait coder en TypeScript, et qui est toujours supportée en Vue 3. Les différents outils de l'écosystème ne supportent pas aussi bien cette syntaxe, car la façon recommandée d'utiliser TypeScript est maintenant l'API de Composition avec `script setup`.

Product.vue

```

<template>
  <div>{{ price }} * {{ quantity }} = {{ total }}</div>
  <CustomButton @click="addOne()">Add 1 unit</CustomButton>
</template>

<script lang="ts">
import { Options, Vue } from 'vue-class-component';
import CustomButton from '@/chapters/many-ways/CustomButton.vue';

@Options({
  components: {
    CustomButton
  }
})
export default class Product extends Vue {
  price = 10;
  quantity = 1;

  get total(): number {
    return this.price * this.quantity;
  }

```

```
addOne(): void {
  this.quantity++;
}
</script>
```

Cette API a même failli devenir l'API par défaut en Vue 3, mais après une [longue discussion](#) avec la communauté, l'API de Composition a été préférée.

Chapter 15. script setup

Vue 3.2 a introduit la syntaxe `script setup`, une façon un peu moins verbeuse de déclarer les composants. On l'active en ajoutant l'attribut `setup` à la balise `script` d'un SFC, et on peut alors enlever un peu de boilerplate du composant.

Prenons un exemple pratique et migrions-le vers cette syntaxe !

15.1. Migrer un composant

Le composant `Pony` ci-dessous a deux props (le `ponyModel` à afficher, et une option `isRunning`). Basée sur ces deux props, une URL est calculée pour l'image du poney affichée dans le template (via un autre composant `CustomImage`). Le composant émet également un événement `selected` quand on clique dessus.

Pony.vue

```
<template>
  <figure @click="clicked()">
    <CustomImage :src="ponyImageUrl" :alt="ponyModel.name" />
    <figcaption>{{ ponyModel.name }}</figcaption>
  </figure>
</template>

<script lang="ts">
import { computed, defineComponent, PropType } from 'vue';
import CustomImage from './CustomImage.vue';
import { PonyModel } from '@/models/PonyModel';

export default defineComponent({
  components: { CustomImage },

  props: {
    ponyModel: {
      type: Object as PropType<PonyModel>,
      required: true
    },
    isRunning: {
      type: Boolean,
      default: false
    }
  },
  emits: {
    selected: () => true
  },
  setup(props, { emit }) {
    const ponyImageUrl = computed(() => `/pony-${props.ponyModel.color}${props
```

```

.isRunning ? '-running' : ''}.gif');

function clicked() {
  emit('selected');
}

return { ponyImageUrl, clicked };
}
});
</script>

```

Première étape : ajoutons l'attribut `setup` à l'élément `script`. On peut alors garder seulement le contenu de la fonction `setup`, et supprimer l'appel à `defineComponent` et `setup` :

Pony.vue

```

<script setup lang="ts">
import { computed, PropType } from 'vue';
import CustomImage from './CustomImage.vue';
import { PonyModel } from '@/models/PonyModel';

components: { CustomImage },

props: {
  ponyModel: {
    type: Object as PropType<PonyModel>,
    required: true
  },
  isRunning: {
    type: Boolean,
    default: false
  }
},
emits: {
  selected: () => true
},
const ponyImageUrl = computed(() => `/pony-${props.ponyModel.color}${props
.isRunning ? '-running' : ''}.gif`);

function clicked() {
  emit('selected');
}

return { ponyImageUrl, clicked };
</script>

```

15.2. Retour implicite

On peut également supprimer le `return` à la fin : toutes les déclarations de haut niveau à l'intérieur d'un `script setup` (ainsi que les imports) sont automatiquement disponibles dans le template. Ici `ponyImageUrl` et `clicked` sont disponibles sans avoir besoin de les renvoyer.

C'est la même chose pour les `components` ! Importer le composant `CustomImage` est suffisant, et Vue comprend qu'il est utilisé dans le template. On peut donc enlever la déclaration `components`.

Pony.vue

```
import { computed, PropType } from 'vue';
import CustomImage from './CustomImage.vue';
import { PonyModel } from '@/models/PonyModel';

props: {
  ponyModel: {
    type: Object as PropType<PonyModel>,
    required: true
  },
  isRunning: {
    type: Boolean,
    default: false
  }
},
emits: {
  selected: () => true
},

const ponyImageUrl = computed(() => `/pony-${props.ponyModel.color}${props.isRunning ? '-running' : ''}.gif`);

function clicked() {
  emit('selected');
}
```

On y est presque : il reste à migrer les déclarations `props` et `emits`.

15.3. `defineProps`

Vue nous donne une fonction `defineProps` que l'on peut utiliser pour déclarer les props. C'est une fonction disponible à la compilation (une macro), il n'y a donc pas besoin de l'importer dans le code : Vue comprend tout seul lorsqu'il compile le composant.

`defineProps` renvoie les props :

Pony.vue

```
const props = defineProps({
  ponyModel: {
    type: Object as PropType<PonyModel>,
    required: true
  },
  isRunning: {
    type: Boolean,
    default: false
  }
});
```

`defineProps` reçoit la même déclaration que `props` comme paramètre. Mais on peut faire encore mieux pour les utilisateurs TypeScript !

`defineProps` est typé de façon générique : on peut l'appeler sans paramètre, mais en spécifiant une interface comme "forme" des props. Plus besoin de l'horrible `Object as PropType<Something>` ! On peut utiliser de vrais types TypeScript, et ajouter `?` pour marquer une prop comme optionnelle 🎉.

Pony.vue

```
const props = defineProps<{
  ponyModel: PonyModel;
  isRunning?: boolean;
}>();
```

Nous avons perdu un bout d'information cependant. Dans la version précédente, nous pouvions spécifier que `isRunning` avait une valeur par défaut à `false`. Pour avoir le même comportement, on peut utiliser la fonction `withDefaults` :

Pony.vue

```
interface Props {
  ponyModel: PonyModel;
  isRunning?: boolean;
}

const props = withDefaults(defineProps<Props>(), { isRunning: false });
```

Ou encore plus simple, depuis Vue 3.2.20 et son option `reactivityTransform`, on peut déstructurer et donner une valeur par défaut à une prop directement :

Pony.vue

```
interface Props {
  ponyModel: PonyModel;
  isRunning?: boolean;
}
```

```
const { ponyModel, isRunning = false } = defineProps<Props>();
```

Cela permet d'utiliser les props directement, sans avoir à écrire `props.` :

Pony.vue

```
const ponyImageUrl = computed(() => `/pony-${ponyModel.color}${isRunning ? '-running' : ''}.gif`);
```

La dernière partie à migrer est la déclaration des événements.

15.4. `defineEmits`

Vue nous offre également une fonction `defineEmits`, très similaire à `defineProps`. `defineEmits` renvoie la fonction `emit` :

Pony.vue

```
const emit = defineEmits({
  selected: () => true
});
```

Ou encore mieux, avec TypeScript :

Pony.vue

```
const emit = defineEmits<{
  selected: [];
}>();
```

La déclaration complète est plus courte de 10 lignes, ce qui est pas mal pour un composant d'une trentaine de lignes ! Le composant est plus simple à lire, et plus facile à écrire en TypeScript. Il est cependant un peu étrange de voir tout exposé automatiquement dans le template, sans avoir à écrire de return, mais on s'habitue.

Pony.vue

```
<template>
  <figure @click="clicked()">
    <CustomImage :src="ponyImageUrl" :alt="ponyModel.name" />
    <figcaption>{{ ponyModel.name }}</figcaption>
  </figure>
</template>

<script setup lang="ts">
  import { computed } from 'vue';
  import CustomImage from './CustomImage.vue';
```

```

import { PonyModel } from '@/models/PonyModel';

interface Props {
  ponyModel: PonyModel;
  isRunning?: boolean;
}

const props = withDefaults(defineProps<Props>(), { isRunning: false });

const emit = defineEmits<{
  selected: [];
}>();

const ponyImageUrl = computed(() => `/pony-${props.ponyModel.color}${props.isRunning ? '-running' : ''}.gif`);

function clicked() {
  emit('selected');
}
</script>

```

15.5. defineOptions

Vue v3.3 a introduit une nouvelle macro appelée `defineOptions`. Elle est pratique pour déclarer certaines options du component tel son nom (si le nom inféré basé sur le nom du fichier ne convient pas) ou pour définir l'option `inheritAttrs` par exemple :

```
defineOptions({ name: 'Pony' });
```

15.6. Fermé par défaut et `defineExpose`

Il y a une différence subtile entre les deux façons d'écrire des composants : un composant `script setup` est "fermé par défaut". Cela veut dire que d'autres composants ne voit pas ce qui est défini à l'intérieur de celui-ci.

Par exemple, le composant `Pony` peut accéder au composant `CustomImage` qu'il utilise (en utilisant des refs, comme on le verra dans un chapitre plus loin). Si `CustomImage` est déclaré avec `defineComponent`, alors tout ce que renvoie sa fonction `setup` est aussi visible par le composant parent (`Pony`). En revanche, si `CustomImage` est défini avec `script setup`, alors *rien* n'est visible pour le composant parent. `CustomImage` peut cependant exposer ce qu'il souhaite en ajoutant `defineExpose({ key: value })`. Alors `value` sera accessible comme étant `key`.

Cette syntaxe est donc celle qui est recommandée pour construire tes composants. Les exemples de composants l'utiliseront donc à partir de maintenant, ainsi que les exercices de notre formation en ligne.



Essaye l'exercice [Script setup](#) pour migrer notre application à cette syntaxe.

Chapter 16. Tester ton application

16.1. Tester c'est douter

J'adore les tests automatisés. Ma vie professionnelle tourne autour de cette barre de progression qui devient verte dans mon IDE, et me félicite d'avoir bien travaillé. J'espère que les tests sont aussi importants pour toi, car ils sont le filet de sécurité ultime quand on écrit du code. Rien n'est plus pénible que tester du code manuellement.

Vue fait du bon travail en nous permettant d'écrire des tests assez simplement, avec l'aide de quelques outils.

Tu peux écrire plusieurs types de test :

- des tests unitaires ;
- des tests de bout en bout (*end-to-end*).

Les premiers sont là pour vérifier qu'une petite unité de code (un composant, un service...) fonctionne correctement en isolation, c'est-à-dire sans considérer ses dépendances. Écrire un test unitaire requiert d'exécuter chacune des fonctions d'un composant ou d'un service, et de vérifier que le résultat obtenu est conforme à nos attentes, en fonction des entrées que nous lui avons fournies. On peut aussi vérifier que les dépendances utilisées par cette unité de code sont correctement appelées. Par exemple, qu'un service envoie la requête HTTP qu'on attend de lui.

On peut aussi écrire des tests de bout en bout. Leur principe est plutôt de simuler un utilisateur qui interagit avec l'application, en démarrant une réelle instance de l'application et en pilotant programmatiquement le navigateur afin de saisir des valeurs dans des champs de formulaire, de cliquer sur des boutons, etc. On peut ensuite vérifier que la page affichée est dans l'état attendu, avec les bonnes informations affichées, une URL correcte, ou quoi que ce soit d'autre.

Nous allons couvrir tout cela. Commençons par les tests unitaires.

16.2. Tests unitaires

Comme nous venons de le voir, les tests unitaires permettent de tester une petite unité de code en isolation. Chaque test permet seulement de vérifier qu'une toute petite partie de l'application fonctionne comme attendu, mais ils ont de nombreux avantages :

- ils sont très rapides. Tu peux en exécuter plusieurs centaines en quelques secondes ;
- ils sont très efficaces pour tester (pratiquement) tout ton code, y compris les cas très particuliers, qu'il peut être difficile de tester avec l'application réelle.

L'un des principes fondamentaux des tests unitaires est l'**isolation** : on ne veut pas s'encombrer des dépendances de l'unité testée. Du coup, on utilise plutôt des "*mocks*" à la place des dépendances réelles. Ce sont des objets ou fonctions fictifs, créés spécifiquement pour les besoins du test.

Pour nous aider à écrire ces tests, nous allons nous appuyer sur quelques outils. D'abord, on a besoin d'une bibliothèque pour écrire les tests.

L'une des plus populaires, (sinon la plus populaire) est [Jest](#), mais nous allons utiliser ma préférée, [Vitest](#) !

16.3. Vitest

Vitest est une solution super intéressante : elle est très similaire à Jest (même API, et aussi basée sur jsdom) mais plus rapide et moderne.

Si tu génères ton projet avec [create-vue](#) au lieu de la Vue CLI, ce sera la solution de test unitaire qui te sera proposée. Les exercices du pack pro utilisent tous Vitest bien sûr : tu vas pouvoir en faire ta source d'inspiration (et de configuration) pour tester toutes les facettes de tes projets !



Vitest a été construit afin d'utiliser Vite pour s'occuper de toutes les transformations nécessaires pour lancer les tests. On n'a donc pas besoin de [ts-jest](#) ou [@vue/vue3-jest](#). Et Vitest marche super bien avec les Modules ECMAScript (ESM), alors que Jest... pas vraiment.

Toutes les options de configuration vont soit dans un fichier dédié [vitest.config.ts](#), soit directement dans la configuration Vite, dans une propriété [test](#).

Vitest nous met à disposition les fonctions suivantes pour structurer nos tests :

- [describe\(\)](#) définit une suite de tests (un groupe de tests) ;
- [test\(\)](#) définit un test ;
- [expect\(\)](#) définit une assertion.

Un test JavaScript basique pour une simple classe telle que celle-ci :

Pony.ts

```
class Pony {
  constructor(public name: string, public speed: number) {}

  isFasterThan(speed: number): boolean {
    return this.speed > speed;
  }
}
```

écrit avec Vitest ressemblerait à cela :

Pony.spec.ts

```
describe('Pony', () => {
  test('should be faster than a slow speed', () => {
    const pony = new Pony('Rainbow Dash', 10);
    expect(pony.isFasterThan(8)).toBe(true);
  });
});
```

L'appel à `expect()` peut être chaîné avec un tas de méthodes, appelée *matchers*, comme `toBe()`, `toBeLessThan()`, `toBeUndefined()`, etc. Chaque *matcher* peut être inversé avec la propriété `not` de l'objet retourné par `expect()`:

Pony.spec.ts

```
describe('Pony', () => {
  test('should not be faster than a fast speed', () => {
    const pony = new Pony('Rainbow Dash', 10);
    expect(pony.isFasterThan(20)).not.toBe(true);
  });
});
```

Le fichier de test est un fichier séparé de celui contenant le code à tester, en général avec une extension `.spec.ts`. Le test d'un composant *Pony* écrit dans le fichier *Pony.vue* sera donc typiquement écrit dans un fichier *Pony.spec.ts*. Où placer le fichier `.spec.ts` est une histoire de préférences : certains aiment placer le code et son test côté à côté, dans un sous-répertoire `__tests__` ; d'autres préfèrent placer le fichier de test dans une arborescence séparée, dédiée aux tests. J'ai tendance à me simplifier la vie en faisant ce que `create-vue` fait par défaut, à savoir un répertoire `__tests__` à côté des fichiers testés.

Souvent, tous les tests définis par un appel à `test()` nécessitent un contexte commun qui doit être préparé avant le test proprement dit. Tu peux utiliser la fonction `beforeEach()` pour mettre en place ce contexte avant chaque test. Par exemple, si j'ai plusieurs tests sur le même poney, ça a du sens d'utiliser `beforeEach()` pour créer et initialiser ce poney, au lieu de copier-coller le même code dans chacun des tests.

Pony.spec.ts

```
describe('Pony', () => {
  let pony: Pony;

  beforeEach(() => {
    pony = new Pony('Rainbow Dash', 10);
  });

  test('should be faster than a slow speed', () => {
    expect(pony.isFasterThan(8)).toBe(true);
  });
});
```

```

});  
  

test('should not be faster than the same speed', () => {
  expect(pony.isFasterThan(10)).toBe(false);
});  

});

```

Il existe aussi une fonction `afterEach`, mais je l'utilise rarement...

Une dernière astuce : Vitest permet de créer les objets fictifs que j'évoquais précédemment (*mocks* ou *spies*, comme tu préfères), ou même d'espionner les appels à une méthode d'un objet réel. On peut ensuite faire des assertions sur ces méthodes espions, telles que `toHaveBeenCalled()` qui vérifie que la méthode a été appelée, ou bien `toHaveBeenCalledWith()` qui vérifie que la méthode a été appelée avec des paramètres bien précis.

Par exemple, supposons qu'on veuille tester cette classe `Race`, qui a une méthode `start()`. Cette méthode appelle `run()` sur chacun des poneys de la course, et retourne les poneys qui ont bien voulu démarrer (ils peuvent être têteux parfois : `run()` retourne un booléen) :

Race.ts

```

class Race {
  constructor(private ponies: Array<Pony>) {}  
  

  start(): Array<Pony> {
    return (
      this.ponies
        // start every pony
        // and only keeps the ones that started running
        .filter(pony => pony.run(10))
    );
  }
}

```

Nous voulons tester la méthode `start()`, et vérifier qu'elle appelle bien `run()` sur chacun des poneys. Donc, on va espionner la méthode `run()` de tous les poneys de la course :

Race.spec.ts

```

describe('Race', () => {
  let rainbowDash: Pony;
  let pinkiePie: Pony;
  let race: Race;  
  

  beforeEach(() => {
    rainbowDash = new Pony('Rainbow Dash');
    // first pony agrees to run
    vi.spyOn(rainbowDash, 'run').mockReturnValue(true);  
  

    pinkiePie = new Pony('Pinkie Pie');
  });
});

```

```

    // second pony refuses to run
    vi.spyOn(pinkiePie, 'run').mockReturnValue(false);

    // create a race with these two ponies
    race = new Race([rainbowDash, pinkiePie]);
};

});

```

et vérifier qu'elle a bien été appelée :

Race.spec.ts

```

test('should make the ponies run when it starts', () => {
    // start the race
    const runningPonies: Array<Pony> = race.start();
    // should have called `run()` on the ponies
    expect(pinkiePie.run).toHaveBeenCalled();
    // with a speed of 10
    expect(rainbowDash.run).toHaveBeenCalledWith(10);
    // as one pony refused to start, the result should be an array of one pony
    expect(runningPonies).toEqual([rainbowDash]);
});

```

Quand tu écris des tests unitaires, veille à ce qu'ils soient courts et lisibles. Et commence par les faire échouer d'abord, pour être sûr qu'ils testent correctement ce qu'ils sont supposés tester.

L'étape suivante est d'exécuter nos tests. Vitest exécute les tests dans [jsdom](#), un environnement ressemblant à celui fourni par un navigateur, mais qui tourne dans Node.js. La Vue CLI fournit la commande `npm run test:unit` pour exécuter les tests unitaires. Elle permet aussi de surveiller tes fichiers et de réexécuter les tests à chaque sauvegarde avec :

```
npm run test:unit -- --watch
```

Comme exécuter les tests est très rapide, c'est pratique d'utiliser cette option et d'avoir un feedback (presque) immédiat sur le code. Si nécessaire, il est aussi possible de n'exécuter que certains tests.

Je ne vais pas rentrer dans le détail de la configuration de Vitest, mais c'est un projet très intéressant, avec de nombreux plugins utilisables, pour le faire fonctionner avec les bibliothèques que tu veux, obtenir de la couverture de code, etc. Et bien sûr, si tu écris ton code, comme moi, en TypeScript, le typage est supporté nativement.

À présent que nous savons comment écrire un test unitaire, ajoutons Vue au cocktail.

16.4. `@vue/test-utils`

Vue a une bibliothèque officielle dédiée aux tests appelée `@vue/test-utils`.

Elle permet d'aller plus loin que de simplement tester du code, en permettant de tester des composants Vue. On peut *monter* (*mount*) un composant dans un test, et tester non seulement son état et ses fonctions, mais aussi sa vue et ses événements.

Supposons que nous ayons écrit un joli composant **Pony**. Il a des *props*, et il affiche une image qui dépend de sa couleur. Il émet aussi un événement lorsque l'utilisateur clique dessus :

Pony.vue

```
<template>
  <figure @click="clicked()">
    
    <figcaption>{{ ponyModel.name }}</figcaption>
  </figure>
</template>

<script setup lang="ts">
  import { computed } from 'vue';
  import { PonyModel } from '@/models/PonyModel';

  const props = defineProps<{
    ponyModel: PonyModel;
  }>();

  const emit = defineEmits<{ ponySelected: [] }>();

  const ponyImageUrl = computed(() => `/images/pony-${props.ponyModel.color.toLowerCase()}.gif`);

  function clicked() {
    emit('ponySelected');
  }
</script>
```

Nous voulons tester que le composant affiche une image avec l'attribut **src** correct. Nous allons donc *monter* le composant, c'est-à-dire instancier le composant, avec sa vue, dans notre test unitaire :

Pony.spec.ts

```
describe('Pony.vue', () => {
  test('should display an image and a legend', () => {
    const ponyModel: PonyModel = {
      id: 1,
      name: 'Fast Rainbow',
      color: 'PURPLE'
    };

    const wrapper = mount(Pony, {
      props: {
```

```
    ponyModel
  }
});
});
});
```

La fonction `mount()` retourne un `wrapper`, qui expose des propriétés et méthodes intéressantes.

Par exemple, le wrapper a une propriété `vm`, qui représente l'instance du composant. Tu peux l'utiliser pour vérifier la valeur des différentes propriétés du composant :

Pony.spec.ts

```
const url = (wrapper.vm as unknown as { ponyImageUrl: string }).ponyImageUrl;
expect(url).toBe('/images/pony-purple.gif');
```

Il a aussi les méthodes `get`, `find` et `findAll`, pour trouver des éléments du DOM à l'intérieur de la vue du composant :

Pony.spec.ts

```
// you can use `wrapper.get` if you want the test to fail if the element is not found
const image = wrapper.get('img');
expect(image.attributes('src')).toBe('/images/pony-purple.gif');
expect(image.attributes('alt')).toBe('Fast Rainbow');

// or you can use `wrapper.find` if you want to check if the element exists or not
const legend = wrapper.find('figcaption');
expect(legend.exists()).toBeTruthy();
expect(legend.text()).toContain('Fast Rainbow');
```

Notre premier test de composant ressemble donc finalement à ça :

Pony.spec.ts

```
test('should display an image and a legend', () => {
  const ponyModel: PonyModel = {
    id: 1,
    name: 'Fast Rainbow',
    color: 'PURPLE'
  };

  const wrapper = mount(Pony, {
    props: {
      ponyModel
    }
  });

  // you can use `wrapper.get` if you want the test to fail if the element is not
  // found
```

```

const image = wrapper.get('img');
expect(image.attributes('src')).toBe('/images/pony-purple.gif');
expect(image.attributes('alt')).toBe('Fast Rainbow');

// or you can use `wrapper.find` if you want to check if the element exists or not
const legend = wrapper.find('figcaption');
expect(legend.exists()).toBeTruthy();
expect(legend.text()).toContain('Fast Rainbow');
});

```

Des méthodes similaires (`getComponent`, `findComponent`, `findAllComponents`) permettent également de trouver les sous-composants Vue utilisés par notre template. Par exemple, si notre composant `Pony` utilisait un sous-composant `CustomImage`, `wrapper.getComponent(CustomImage)` nous permettrait de le localiser.

Testons à présent l'émission de l'événement lorsqu'on clique sur le poney. La méthode `trigger` de vue-test-utils permet de déclencher un événement :

Pony.spec.ts

```

test('should emit an event on click', () => {
  const ponyModel: PonyModel = {
    id: 1,
    name: 'Fast Rainbow',
    color: 'PURPLE'
  };

  const wrapper = mount(Pony, {
    props: {
      ponyModel
    }
  });

  const figure = wrapper.get('figure');
  figure.trigger('click');
  // Check that the click handler on the `figure` element works
  // and emits the `ponySelected` event
  expect(wrapper.emitted().ponySelected).toBeTruthy();
});

```

Si tu veux vérifier le DOM après avoir changé l'état du composant, ou après avoir déclenché un événement, il te faut attendre que le DOM soit mis à jour. Pour cela, tu peux:

- soit appeler `nextTick()` pour rafraîchir le DOM ;
- soit directement utiliser `await` sur l'action comme dans l'exemple suivant (ce qui appelle `nextTick()` pour nous).

```

test('should update image when the color changes', async () => {
  const ponyModel: PonyModel = {
    id: 1,
    name: 'Fast Rainbow',
    color: 'PURPLE'
  };

  const wrapper = mount(Pony, {
    props: {
      ponyModel
    }
  });

  const image = wrapper.get('img');
  expect(image.attributes('src')).toBe('/images/pony-purple.gif');

  // Update the prop
  await wrapper.setProps({
    ponyModel: {
      id: 1,
      name: 'Fast Rainbow',
      color: 'BLUE'
    }
  });

  // Check that the image has been updated
  expect(image.attributes('src')).toBe('/images/pony-blue.gif');
});

```

Comme tu peux le voir, on appelle la méthode `wrapper.setProps` pour mettre à jour les props, et on utilise `await` pour attendre le résultat. Tu peux faire de même avec `trigger`.

`@vue/test-utils` fournit aussi des options pour monter un composant. Les plus utiles sont les suivantes :

- l'option `stubs` pour monter un composant et remplacer certains sous-composants ;
- l'option `plugins` pour ajouter un plugin au test, comme le routeur .

Nos exercices en ligne contiennent des tests qui mettent en application toutes ces fonctions, bien sûr.

16.5. Snapshot testing

Un moyen différent (ou complémentaire) de tester les templates est d'utiliser le *snapshot testing*. Vitest permet d'enregistrer un *snapshot* (un instantané) du DOM généré par un test de composant (dans un format lisible) et de le stocker dans un fichier accompagnant les tests, afin qu'il soit examiné manuellement.

Pony.spec.ts

```
test('should match the snapshot', () => {
  const ponyModel: PonyModel = {
    id: 1,
    name: 'Fast Rainbow',
    color: 'PURPLE'
  };

  const wrapper = mount(Pony, {
    props: {
      ponyModel
    }
  });

  expect(wrapper.element).toMatchSnapshot();
});
```

Lorsque tu exécutes le test pour la première fois, le test ne fait aucune vérification, mais il génère le snapshot. Tu peux ensuite vérifier que le snapshot contient la structure du DOM attendue (et corriger le code jusqu'à ce que ce soit le cas).

Une fois que tu es satisfait du résultat, tu peux enregistrer le snapshot dans ton système de gestion de versions (il devient une partie des sources du projet), de façon à ce que chaque exécution suivante du test vérifie que le DOM généré par le test corresponde toujours au snapshot enregistré.

Si tu utilises cette fonctionnalité, tu verras un répertoire `__snapshots__` apparaître, contenant le snapshot :

Pony.spec.ts.snap

```
// Vitest Snapshot v1, https://vitest.dev/guide/snapshot.html

exports['Pony.vue > should match the snapshot 1'] = `

<figure>
  
  <figcaption>
    Fast Rainbow
  </figcaption>
</figure>
`;
```

Si tu modifies ensuite le template, le test échoue et montre les différences par rapport au snapshot enregistré. Dans cet exemple, j'ai ajouté un point d'exclamation après le nom du poney :

```
Pony.vue > should match the snapshot
```

```

expect(received).toMatchSnapshot()

- Expected - 1
+ Received + 1

    alt="Fast Rainbow"
    src="/images/pony-purple.gif"
  />
  <figcaption>
-   Fast Rainbow
+   Fast Rainbow!
  </figcaption>
</figure>"
```

FAIL Tests failed. Watching for file changes...
press u to update snapshot, press h to show help

À toi ensuite de choisir si le changement est un bug (et donc de le corriger) ou bien si c'est un changement voulu, et donc d'accepter les changements et d'enregistrer le nouveau snapshot.

Même si l'approche est séduisante, je ne suis pas un grand fan de cette pratique, parce que les développeurs acceptent parfois les changements sans y prêter une grande attention. Je considère donc le snapshot testing comme un complément aux tests conventionnels, mais pas comme un remplacement. Cela peut néanmoins être utile parfois, et c'est très simple à écrire !

16.6. Tests de bout en bout

Les tests de bout en bout (*end-to-end*, or *e2e*) sont l'autre type de tests que tu peux écrire et exécuter. Ils consistent à réellement lancer ton application dans un navigateur, et à simuler un utilisateur interagissant avec l'application (en cliquant sur des boutons, remplissant des formulaires, etc.). Ils ont l'avantage de tester réellement l'application dans son ensemble plutôt que des petites unités isolées, mais :

- ils sont plus lents (plusieurs secondes par test) ;
- il est plus difficile de tester les cas à la marge.

Comme tu peux le deviner, tu n'as pas à choisir entre tests unitaires et tests e2e. Combiner les deux est l'idéal pour avoir une bonne couverture et une assurance raisonnable du bon fonctionnement de l'application.

Vue CLI propose l'utilisation de deux frameworks de tests : [Cypress](#) et [Nightwatch.js](#), et `create-vue` donne également le support de Cypress. Et je suis en pâmoison devant Cypress 😊.

Tu peux lancer les tests e2e avec :

```
npm run test:e2e
```

16.7. Cypress



Lorsque tu exécutes les tests e2e, Cypress lance Electron, Chrome, Firefox ou Edge. Tu peux aussi les lancer en mode *headless* (c'est-à-dire sans afficher la moindre fenêtre) avec :

```
npm run test:e2e -- --headless
```

ce qui est pratique pour exécuter ces tests sur une plateforme d'intégration continue.

Cypress a plein de fonctionnalités intéressantes :

- il est facile à mettre en place ;
- il permet de *mocker* des réponses HTTP ;
- il permet de tester l'application avec différentes tailles de fenêtre (pratique pour les applications *responsive*) ;
- il offre un DSL relativement simple.

Ce qui m'a conquis cependant est ce qu'il appelle le *time-travel debugging*. À chaque étape du test, Cypress fait un snapshot, qui permet donc de visualiser, directement dans le navigateur, l'aspect graphique et le contenu du DOM. En plaçant simplement sa souris sur une étape d'un test en échec, on peut voir le contenu de la page et interagir avec elle pour rapidement comprendre ce qui ne va pas.

Les tests Cypress sont écrits avec Mocha. Même si c'est une bibliothèque différente de Vitest, tu verras qu'elle est très similaire. Ce qui importe plus, c'est l'API spécifique de Cypress nous permettant d'interagir avec l'application.

Cypress nous met à disposition un objet `cy`, qui nous permet par exemple d'appeler la méthode `visit()` pour... visiter une page. Ensuite, `get(selector)` permet de sélectionner tous les éléments qui correspondent à un sélecteur CSS. Ou encore `contains(selector, text)` permet de cibler les éléments contenant un texte donné.

Les objets rentrés par ces méthodes offrent eux-mêmes des méthodes permettant de vérifier leur état, comme `should('contain', text)` or `should('be', 'not.displayed')`. Ou encore des méthodes comme `click()` et `type()` qui permettent d'interagir avec eux.

Comme je l'évoquais précédemment, tu peux aussi isoler les tests du backend, en mockant les appels HTTP, et vérifier que les requêtes sont envoyées aux moments opportuns.

Par exemple, supposons que la page de recherche de l'application affiche un titre, et un champ de

recherche permettant de trouver une course en appelant une API REST lorsque l'utilisateur clique sur un bouton.

Un test de cette page ressemblerait à ça :

```
describe('Search', () => {
  it('should display a title', () => {
    cy.visit('/search');
    cy.contains('h1', 'Ponyracer');
  });

  it('should search a race', () => {
    // mock the API call
    cy.intercept('GET', 'api/races?query=London', []).as('searchRace');

    cy.visit('/search');
    // search the London race
    cy.get('input').type('London');
    // click on the search button
    cy.get('button').click();
    // we should have an API call
    cy.wait('@searchRace');
  });
});
```

Ces tests peuvent être assez longs à écrire, mais ils sont vraiment utiles. On peut les utiliser à d'autres fins, par exemple enregistrer des vidéos de démonstration, prendre des copies d'écran, les comparer au pixel prêt d'une exécution à l'autre avec [Percy](#), etc.

Avec les tests unitaires et les tests e2e, tu as toutes les clés en main pour construire une application robuste et maintenable !



Tous les exercices de notre Pro Pack viennent avec des tests unitaires et e2e. Si tu veux en apprendre plus, nous t'encourageons à y jeter un œil : nous avons testé toute l'application (100 % de couverture) ! À la fin, tu auras des douzaines d'exemples dont tu pourras t'inspirer pour tes propres projets. Même les cas plus avancés sont couverts, comme la manipulation du temps dans l'exercice [Boost a pony](#) 🦄, ou la communication par Websockets dans l'exercice [Reactive score](#) 🦄.

Chapter 17. Envoyer et recevoir des données avec HTTP

Cela ne vous surprendra pas : une bonne part de notre travail consiste à demander à un serveur (le *backend*) de fournir des données à notre application (le *frontend*), et à lui fournir des informations en retour.

En règle générale, on utilise HTTP pour cela, même s'il existe des alternatives de nos jours, comme les *WebSockets*. Vue ne fournit pas de module http, mais il existe de nombreuses bibliothèques ou APIs qui nous permettent d'envoyer des requêtes HTTP.

L'un des choix possibles est l'API `fetch`, qui est directement mise à disposition par la plupart des navigateurs. Vous pouvez parfaitement construire votre application en vous basant sur `fetch`.

Mais de nombreux développeurs utilisent plutôt la bibliothèque `axios`. Axios offre une API basée sur les promesses, et est utilisable aussi bien dans Node.js que dans les navigateurs.

Et elle offre quelques fonctionnalités pratiques que `fetch` ne fournit pas. En route !

17.1. Obtenir des données

Axios fournit un service sous la forme d'un objet que tu peux importer dans n'importe quel fichier.

```
import axios from 'axios';
```

Il expose plusieurs méthodes, qui correspondent aux *verbes* du protocole HTTP :

- `get`
- `post`
- `put`
- `delete`
- `patch`
- `head`

Toutes ces méthodes retournent une `Promise`, que tu peux donc utiliser en appelant `then`, ou en utilisant `async/await`, qui est ma technique favorite.

Commençons par rapatrier les courses disponibles dans Ponyracer. Nous supposerons qu'un *backend* est déjà en place, et qu'il expose une API *RESTful*. Pour obtenir les courses, nous enverrons une requête GET à une URL telle que 'http://backend.url/api/races'.

En général, l'URL de base de tes requêtes HTTP sera stockée dans une variable ou un service, que tu configureras en fonction de ton environnement. Si les ressources REST sont servies par le même serveur que celui qui sert ton application Vue, tu peux utiliser un chemin plutôt qu'une URL complète : '/api/races'.

Utiliser `axios` pour envoyer une telle requête est simplissime :

```
const response = await axios.get('/api/races');
```

`axios.get` retourne une `Promise` contenant la réponse en cas de succès, donc on peut utiliser `await` pour obtenir le résultat.

Le contenu (*body*) de la réponse est en général la partie qui nous intéresse. Il se trouve dans la propriété `data` de la réponse :

```
// races is of type 'any'  
const races = response.data;
```

Comme `axios` n'a aucune idée des données que tu essaies d'obtenir, la méthode `get` est générique, et tu peux (et en général, devrait) spécifier le type de données retournées par le serveur.

```
const response = await axios.get<Array<RaceModel>>('/api/races');  
// races is of type 'Array<RaceModel>'  
const races = response.data;
```

Remarque que tu n'as pas besoin de déserialiser le *body* de la réponse pour transformer son contenu en objet ou en tableau JavaScript. C'est fait automatiquement par Axios. Cependant, Axios ne fait aucune vérification pour s'assurer que le document JSON obtenu est conforme au type générique spécifié lors de l'appel. Il t'appartient de choisir le type correct, et de concevoir l'interface `RaceModel` de manière à ce qu'elle corresponde effectivement à la structure du document JSON envoyé par le serveur.

Bien entendu, tu as aussi accès au reste des informations contenues dans la réponse HTTP, grâce aux propriétés `status`, `headers`, etc.

```
const response = await axios.get<Array<RaceModel>>('/api/races');  
// status contains 200  
const status = response.status;  
// headers contains []  
const headers = response.headers;
```

La promesse retournée sera en erreur si le statut de la réponse est différent de 2xx or 3xx. Dans ce cas, l'erreur contenue dans la promesse est de type `AxiosError`.

Envoyer des données est également assez simple. Il suffit d'appeler `post()` ou `put()`, avec l'URL de destination et l'objet à envoyer :

```
const response = await axios.post<RaceModel>('/api/races', newRace);
```

Encore une fois, nul besoin de sérialiser l'objet en JSON. Axios s'en charge. Le type générique `RaceModel`, comme pour l'appel à `get()`, représente le type attendu dans la réponse. Donc, dans cet exemple, le serveur attend un `RaceModel` en entrée, et fournit en sortie le `RaceModel` créé.

Je ne te montrerai pas d'exemple pour les autres méthodes : je suis sûr que tu as compris le principe.

17.2. Options avancées

Bien sûr, il est possible de configurer tes requêtes plus finement. Chaque méthode prend un argument optionnel `config`, qui permet de configurer la requête comme tu l'entends. Voici quelques-unes des options, les plus utiles.

`params` représente les paramètres de recherche de la requête à ajouter en fin d'URL (qu'on connaît sous le nom de *query string*).

```
const params = {
  sort: 'ascending',
  page: '1'
};
const response = await axios.get<Array<RaceModel>>('/api/races', { params });
// will call the URL /api/races?sort=ascending&page=1
```

L'option `headers` permet d'ajouter des entêtes à la requête. C'est souvent nécessaire, par exemple pour l'authentification basée sur des *JSON Web Tokens* :

```
const headers = { Authorization: `Bearer ${token}` };
const response = await axios.get<Array<RaceModel>>('/api/races', { headers });
```

Nous mettrons cela en pratique dans nos exercices très bientôt.

17.3. Intercepteurs

L'une des raisons de vouloir utiliser Axios plutôt que l'API Fetch est le support des intercepteurs. Ils sont intéressants pour... intercepter les requêtes envoyées ou les réponses reçues par ton application.

Supposons par exemple que tu veuilles ajouter un entête spécifique à tout un ensemble de requêtes HTTP. C'est possible, sans répéter le même code pour chacune des requêtes, en écrivant un intercepteur tel que celui-ci :

```
axios.interceptors.request.use((config: InternalAxiosRequestConfig) => {
  if (user) {
    config.headers!.Authorization = `Bearer ${user.token}`;
  }
  return config;
```

```
});
```

À présent, chaque requête passera par l'intercepteur, qui ajoutera l'entête si nécessaire (dans notre exemple, si l'utilisateur courant est identifié).

Tu peux aussi intercepter les réponses. C'est extrêmement pratique pour gérer les erreurs de manière générique et centralisée :

```
axios.interceptors.response.use(  
  (response: AxiosResponse) => response,  
  error => {  
    // do whatever you want with the error  
    return Promise.reject(error);  
  }  
);
```

17.4. Tests

Nous avons maintenant un composant ou un service qui envoie une requête HTTP pour obtenir les courses de poneys. Comment le tester ?

Dans un test unitaire, nous voulons éviter de réellement envoyer une requête au serveur : ce n'est pas ce que nous voulons tester. Nous voudrions donc simuler un serveur et lui faire renvoyer des données de test fictives. Il nous suffit pour cela d'utiliser `vi.spyOn` pour "remplacer" les méthodes d'Axios et leur faire retourner ce qu'on désire :

```
const fakeRace = { id: 1 } as RaceModel;  
const fakeResponse = { status: 200, data: fakeRace } as AxiosResponse;  
vi.spyOn(axios, 'get').mockResolvedValue(fakeResponse);
```

Et voilà !



Essaye notre exercice [HTTP 🐾](#) ! Nous avons préparé une API REST complète, prête à être utilisée. Récupérons donc les courses avec Axios. Plus tard, nous apprendrons comment appeler une API sécurisée avec un système d'authentification et des intercepteurs dans l'exercice [HTTP avec authentification 🐾](#). Dans le même genre, nous utiliserons aussi les [WebSockets 🐾](#).

Chapter 18. Slots

18.1. Projection de contenu avec `slot`

Nous, les développeurs, n'aimons pas nous répéter. Il est donc assez commun de vouloir créer des composants réutilisables, dont le contenu est dynamique.

Par exemple, supposons que nous voulions utiliser [un composant "card"](#) en utilisant le framework CSS Bootstrap. Le modèle d'une carte est le suivant :

```
<div class="card">
  <div class="card-body">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some quick example text</p>
  </div>
</div>
```

On peut bien sûr copier-coller ce *boilerplate* chaque fois qu'on veut afficher une carte dans l'application. Mais à ce moment de ta lecture, tu dois sans doute te dire que ce serait une bonne idée d'en faire un composant. Deux parties de ce composant sont dynamiques (le titre et le contenu), donc ta première idée consistera sans doute à utiliser des *props* :

Card.vue

```
<template>
  <div class="card">
    <div class="card-body">
      <h4 class="card-title">{{ title }}</h4>
      <p class="card-text">{{ text }}</p>
    </div>
  </div>
</template>

<script setup lang="ts">
defineProps<{
  title: string;
  text: string;
}>();
</script>
```

et tu utiliseras le composant ainsi :

```
<Card title="Card title" text="Some quick example text"></Card>
```

Cela fonctionne parfaitement. Mais, en examinant ton besoin plus attentivement, tu vas réaliser que le contenu de la carte est souvent en fait du HTML, contenant éventuellement des composants,

des directives, des *bindings*, et pas juste du texte.

Bien sûr, Vue supporte ce cas d'utilisation. C'est très simple de fournir un bout de template HTML à un composant, grâce à `<slot>`.

`slot` est un tag fourni par Vue, te permettant d'inclure, dans le template d'un composant enfant, un morceau de template fourni par le composant parent :

Card.vue

```
<template>
  <div class="card">
    <div class="card-body">
      <h4 class="card-title">{{ title }}</h4>
      <p class="card-text">
        <slot></slot>
      </p>
    </div>
  </div>
</template>

<script setup lang="ts">
defineProps<{
  title: string;
}>();
</script>
```

Un tel composant s'utilise comme ceci :

```
<Card title="Card title">Some quick <strong>example</strong> text</Card>
```

On peut également passer un composant Vue, par exemple :

```
<Card title="Card title">
  <Pony :ponyModel="pony" />
</Card>
```

Bien que, dans le DOM généré par Vue, le composant `Pony` soit affiché à l'intérieur du composant `Card`, toutes les expressions (comme `pony` dans l'exemple ci-dessus) sont évaluées sur le composant parent, et pas sur le composant `Card`. Le morceau de template passé au composant `Card` n'a pas accès à l'état du composant `Card`.

18.2. Slots nommés

Finalement, tu réaliseras aussi sans doute que le titre, lui aussi, devrait pouvoir contenir du HTML. Il est possible de passer plusieurs parties de contenu au composant `Card`, à condition de *nommer* les slots.

```
<template>
  <div class="card">
    <div class="card-body">
      <h4 class="card-title">
        <slot name="title"></slot>
      </h4>
      <p class="card-text">
        <slot></slot>
      </p>
    </div>
  </div>
</template>
```

Le slot sans nom est implicitement nommé `default`.

Pour fournir le titre et le contenu au composant `Card`, il te suffit de les envelopper dans un élément `template` et de donner à ces deux éléments le nom du slot qu'ils doivent remplir. Un `template` est juste un élément invisible.

```
<Card>
  <template v-slot:title>Card title</template>
  <template v-slot:default>Some quick <strong>example</strong> text</template>
</Card>
```

Tu peux aussi utiliser la syntaxe moins verbeuse (et en général préférée) `#name` :

```
<Card>
  <template #title>Card title</template>
  <template #default>Some quick <strong>example</strong> text</template>
</Card>
```

Note que le nom `#default` n'a même pas besoin d'être spécifié :

```
<Card>
  <template #title>Card title</template>
  <template>Some quick <strong>example</strong> text</template>
</Card>
```

Et tu peux même te passer de l'élément `template` autour du contenu du slot par défaut :

```
<Card>
  <template #title>Card title</template>
  Some quick <strong>example</strong> text
```

```
</Card>
```

Certains composants fournis par Vue utilisent ce mécanisme de slots. Nous en utiliserons certains dans les chapitres suivants.

En tant que développeur d'applications, tu finiras aussi par utiliser les slots dans tes propres composants.



Essaye notre exercice [Slots](#) pour construire un nouveau composant utilisant cette fonctionnalité.

18.3. Contenu par défaut

Si le composant parent ne fournit aucun contenu pour un slot, le composant enfant peut afficher un contenu par défaut au lieu de laisser le slot vide. On peut donc, par exemple, donner un titre par défaut à notre composant `Card` :

`Card.vue`

```
<template>
  <div class="card">
    <div class="card-body">
      <h4 class="card-title">
        <slot name="title">Default title</slot>
      </h4>
      <p class="card-text">
        <slot></slot>
      </p>
    </div>
  </div>
</template>
```

Si on utilise le composant `Card` et qu'on omet de spécifier un titre, le titre *Default title* sera affiché.

18.4. Slot props

Comme on l'a expliqué avant, le contenu du slot peut accéder à l'état du composant **courant**. Par exemple, si on utilise le composant `Card`, on peut écrire :

```
<Card>
  <template #title>Card title</template>
  Some quick <strong>{{ example }}</strong> text
</Card>
```

où `example` est une propriété du composant **courant**, et non une propriété de `Card`.

Mais parfois, c'est très pratique de pouvoir accéder à une propriété ou une fonction exposée par le

composant `Card` lui-même. Imaginons qu'il ait une fonction permettant de masquer le composant, appelée quand on clique sur le bouton `Close` :

`Card.vue`

```
<template>
  <div v-if="!isClosed" class="card">
    <div class="card-body">
      <h4 class="card-title">
        <slot name="title">Default title</slot>
      </h4>
      <p class="card-text">
        <slot></slot>
      </p>
      <button class="btn btn-primary" @click="close()">Close</button>
    </div>
  </div>
</template>

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

const isClosed = ref(false);
const close = () => (isClosed.value = true);
</script>
```

Tout le template du composant est enveloppé dans un `v-if`, et si l'utilisateur clique sur le bouton `Close`, la condition du `v-if` devient fausse, enlevant donc la carte de la page.

Tout fonctionne comme prévu, mais, quelques semaines plus tard, le client demande à pouvoir fermer certaines cartes en cliquant sur un bouton contenu dans le titre de la carte. Le problème, c'est que le titre ne fait pas partie du composant `Card`. Il est passé au composant `Card` par le composant parent, dans un template. Et ce template doit donc pouvoir appeler la fonction `close` du composant `Card`.

C'est dans ce genre de situation qu'on utilise des *slot props*. Il s'agit de propriétés exposées au template d'un slot par le composant qui définit ce slot. Il peut exposer n'importe quelle propriété ou fonction en utilisant la syntaxe (classique ou raccourcie) de `v-bind` :

`Card.vue`

```
<h4 class="card-title">
  <slot name="title" :closeCard="close">Default title</slot>
</h4>
```

Dans l'exemple ci-dessus, le composant `Card` met à disposition du template passé par le parent pour le slot `title` sa fonction `close`. Cette fonction est exposée sous le nom `closeCard`, mais on pourrait bien sûr choisir d'utiliser le nom original.

On parle aussi de *scoped props* : ce sont des propriétés dont la portée est limitée au template d'un

slot précis.

Maintenant, quand on utilise le composant `Card`, on peut accéder aux propriétés du slot `title` (c'est-à-dire à la fonction `closeCard`) depuis le template passé pour remplir ce slot :

```
<Card>
  <template #title="titleProps">
    <div class="title" @click="titleProps.closeCard()">Card title</div>
  </template>
  Some quick <strong>example</strong> text
</Card>
```

Si le composant expose plusieurs `slot props`, mais que le parent ne veut en utiliser qu'une ou quelques-unes, la déstructuration s'avère utile pour rendre le code plus concis et lisible :

```
<Card>
  <template #title="{ closeCard }">
    <div class="title" @click="closeCard()">Card title</div>
  </template>
  Some quick <strong>example</strong> text
</Card>
```

Nous renconterons à nouveau cette syntaxe dans les chapitres suivants !

18.5. Slots typés avec `defineSlots`

Depuis Vue v3.3, il est possible d'utiliser une nouvelle macro appelée `defineSlots` (ou une option `slots` si tu utilises `defineComponent`). Cette macro a été ajoutée au framework pour aider à déclarer le typage des slots. En faisant cela, Volar sera capable de vérifier les slot props.

```
defineSlots<{
  // default slot has no props
  default: (props: Record<string, never>) => void;
  // title slot has a prop `closeCard`, which is a function
  title: (props: { closeCard: () => void }) => void;
}>();
```

Si le composant `Card` n'est pas utilisé correctement, Volar émet une erreur :

```
<Card>
  <template #title="{ close }">...</template>
</Card>
<!-- error TS2339: Property 'close' does not exist on type '{ closeCard: () =&gt; void; }'. --&gt;</pre>
```

La valeur renvoyée par `defineProps` peut être utilisée et est le même objet qui est renvoyé par `useSlots`.

Chapter 19. Suspense



Cette API est expérimentale et pourrait changer dans le futur.

Lorsqu'un composant doit obtenir des données de manière asynchrone, nous avons vu qu'il pouvait le faire en passant au *hook* `onMounted` une fonction `async`, qui charge et attend les données.

Par exemple, supposons que l'on veuille afficher un poney et son palmarès.

Le composant `Pony` pourrait ressembler à cela :

`Pony.vue`

```
<div>
  <h1>Pony {{ ponyModel.name }}</h1>
  <TrackRecord :ponyId="ponyModel.id" />
</div>
```

`TrackRecord` étant lui-même un composant avec le template suivant :

`TrackRecord.vue`

```
<div>
  <h2>Track record</h2>
  <ul>
    <li v-for="record in trackRecord" :key="record.id">{{ record.position }} - {{ record.race.name }}</li>
  </ul>
</div>
```

qui obtient le palmarès du poney depuis le *hook* `onMounted` :



On n'utilise pas ici `script setup` volontairement, afin de plus facilement expliquer ce qu'il se passe.

`TrackRecord.vue`

```
setup(props) {
  const trackRecord = ref<Array<TrackRecordModel>>([]);
  const ponyService = usePonyService();
  onMounted(async () => (trackRecord.value = await ponyService.getTrackRecord(props.ponyId)));
  return { trackRecord };
}
```

Que se passe-t-il si on enlève le *hook* `onMounted`, et qu'on charge le palmarès directement depuis la fonction `setup` ?

```
async setup(props) {  
  const ponyService = usePonyService();  
  const trackRecord = ref(await ponyService.getTrackRecord(props.ponyId));  
  return { trackRecord };  
}
```

C'est un peu moins verbeux, mais pour que ce code compile, il faut ajouter le mot-clé `async` à la fonction `setup`. Si tu essaies d'utiliser ce composant dans l'application, tu verras que cela ne fonctionne pas : Vue affiche une erreur dans la console et n'affiche pas le composant.

En effet, si la fonction `setup` d'un composant est asynchrone, Vue exige qu'on enveloppe son élément dans un composant `Suspense`. `Suspense` est fourni par Vue, disponible globalement, et donc directement utilisable dans tous les templates. Utilisons-le dans le composant `Pony` :

```
<div>  
  <h1>Pony {{ ponyModel.name }}</h1>  
  <Suspense>  
    <div>  
      <TrackRecord :ponyId="ponyModel.id" />  
    </div>  
  </Suspense>  
</div>
```

Ça marche ! `Suspense` attend la complétion de la fonction `setup` du composant `TrackRecord`, et l'affiche dès qu'il est prêt.

Tu reconnais (j'espère) ici que le composant `Suspense` a un `slot`, et que nous lui passons `TrackRecord` comme contenu par défaut. Tu pourrais envelopper l'élément `TrackRecord` dans un élément `template` avec le nom `#default`, et cela reviendrait donc au même.

Mais ça ne s'arrête pas là !

19.1. Afficher une alternative

Pendant tout le temps où la fonction `setup()` s'exécute, `Suspense` permet d'afficher un contenu alternatif. Il suffit pour cela de lui fournir un élément `template` nommé `fallback` :

```
<div>  
  <h1>Pony {{ ponyModel.name }}</h1>  
  <Suspense>  
    <div>  
      <TrackRecord :ponyId="ponyModel.id" />  
    </div>
```

```

<template #fallback>Loading...</template>
</Suspense>
</div>

```

`Suspense` affichera donc `Loading...` jusqu'à ce que le composant `TrackRecord` soit prêt et puisse remplacer ce message alternatif !

`Suspense` ne se limite pas à un seul composant. On peut y placer autant de composants que l'on veut. Le contenu alternatif sera affiché jusqu'à ce que tous les composants soient prêts.

Par exemple, si on veut aussi afficher l'extrait de naissance du poney, on peut ajouter le ce composant `BirthCertificate` à l'intérieur de `Suspense` :

Pony.vue

```

<div>
  <h1>Pony {{ ponyModel.name }}</h1>
  <Suspense>
    <div>
      <TrackRecord :ponyId="ponyModel.id" />
      <BirthCertificate :ponyId="ponyModel.id" />
    </div>
    <template #fallback>Loading...</template>
  </Suspense>
</div>

```

Le message de chargement sera affiché tant que *toutes* les données ne sont pas disponibles, c'est-à-dire jusqu'à ce que le palmarès *et* l'extrait de naissance soient prêts.

19.2. Gérer les erreurs avec `onErrorCaptured`

Que se passe-t-il si l'un des composants échoue à charger les données dont il a besoin ? L'indicateur de chargement sera-t-il affiché *ad vitam æternam* ?

Non, le contenu alternatif disparaît quand les fonctions `setup` se terminent, avec succès ou pas. Donc seuls les composants qui sont parvenus à charger leurs données seront affichés.

On peut cependant réagir en cas d'erreur, en utilisant le *hook* `onErrorCaptured`. `onErrorCaptured` peut être utilisé, dans un composant, pour capturer toute erreur levée par un composant descendant. Le *hook* peut bien sûr gérer l'erreur comme il l'entend, puis retourner `true` (ou ne rien retourner du tout) pour propager l'erreur vers les composants parents, ou retourner `false` pour stopper la propagation de l'erreur.

Pony.vue

```

const errorMessage = ref<string | null>(<b>null</b>);
onErrorCaptured((e: unknown) => {
  console.warn(e);
  errorMessage.value = (e as Error).message;
}

```

```
    return false;  
});
```

Une fois l'erreur capturée comme montré ci-dessus, on peut l'afficher avec un `v-if` :

Pony.vue

```
<div>  
  <h1>Pony {{ ponyModel.name }}</h1>  
  <div v-if="errorMessage">An error occurred while loading data: {{ errorMessage }}</div>  
  <Suspense>  
    <div>  
      <TrackRecord :ponyId="ponyModel.id" />  
      <BirthCertificate :ponyId="ponyModel.id" />  
    </div>  
    <template #fallback>Loading...</template>  
  </Suspense>  
</div>
```

Si l'un des composants échoue à charger ses données, le message d'erreur sera affiché, ainsi que les autres composants. On pourrait bien sûr placer un `v-else` sur l'élément `Suspense` pour n'afficher aucun autre composant en cas d'erreur.

19.3. Événements `resolve` et `recede`

Le composant `Suspense` émet deux événements que l'on peut écouter :

- `resolve`, émis lorsque tous les composants enfants à afficher sont résolus, c'est-à-dire prêts à être affichés ;
- `recede`, émis lorsqu'un nouveau composant enfant en attente de données apparaît.

Tu pourrais par exemple combiner ces deux événements pour afficher un indicateur chaque fois qu'un composant enfant charge des données, et le masquer lorsque plus rien n'est en cours de chargement.

19.4. `Suspense` contre `onMounted`

Alors quel est l'intérêt d'utiliser `Suspense` plutôt que `onMounted`? Les deux conviennent dans la plupart des cas pour être honnête. Un composant avec un setup `async` est un peu moins verbeux que son alternative utilisant `onMounted`, mais tu dois utiliser `Suspense` pour l'afficher. La grande force de `Suspense`, c'est qu'il fonctionne avec un ou plusieurs composants asynchrones, ou même avec composants asynchrones imbriqués les uns dans les autres. Cela nous donne une façon élégante de gérer l'état de chargement ou l'état en erreur d'une manière centralisée, même si l'on charge un grand nombre de composants asynchrones qui ne se résoudront pas au même moment.

`onMounted` est quand même toujours utile. Elle est toujours nécessaire si tu as besoin de manipuler le DOM quand le composant est créé. Elle fonctionne aussi un peu différemment :

- en utilisant `Suspense`, le composant asynchrone ne s'affichera pas tant que les données ne sont pas chargées.
- en utilisant `onMounted`, le composant s'affiche directement, sans données, puis les données apparaissent quand elles sont chargées.

`Suspense` est encore au stade expérimental, et devrait être stabilisé en version 3.1.

19.5. script setup et await

Tu as peut-être remarqué que ce chapitre utilise des composants avec `defineComponent`. C'est volontaire afin d'expliquer la subtilité du `async` à ajouter. Mais que se passe-t-il si on utilise `script setup`? On n'a alors pas besoin d'ajouter de mot-clé `async`: le compilateur Vue s'en charge pour nous !

Voici le même composant `TrackRecord` en mode `script setup`:

`TrackRecord.vue`

```
<script setup lang="ts">
import { ref } from 'vue';
import { usePonyService } from './PonyService';

const props = defineProps<{
  ponyId: number;
}>();

const ponyService = usePonyService();
const trackRecord = ref(await ponyService.getTrackRecord(props.ponyId));
</script>
```

Tu peux bien sûr utiliser cette version dans tes applications, et c'est ce que l'on fait dans l'exercice pour mettre ça en pratique.



Essaie notre exercice `Suspense` 🎯 pour charger des données asynchrones avec style !

Chapter 20. Le routeur

Être capable de naviguer entre différents composants de l'application et voir l'URL changer dans la barre d'adresse du navigateur est crucial, même dans une application *single page*.

Les utilisateurs doivent pouvoir ajouter une page de l'application à leurs favoris, ou utiliser l'historique de navigation du navigateur par exemple.

La partie du framework en charge de gérer cette navigation est appelée le *routeur*. Tous les frameworks tels que Vue en ont un, voire plusieurs. Vue n'échappe pas à la règle, et fournit un routeur officiel, appelé `vue-router`.

Le routeur a un objectif simple : permettre d'utiliser des URLs reflétant l'état de l'application. Et pour chaque URL, savoir quel composant doit être initialisé et inséré dans la page. Il permet de naviguer d'une route à l'autre, sans recharger la page depuis le serveur. C'est tout le principe d'une *Single Page Application (SPA)*.

20.1. En route

Le routeur est un plugin optionnel, qui n'est donc pas inclus dans le noyau du framework.

Pour pouvoir l'utiliser, il faut donc, comme pour les autres bibliothèques tierces, l'installer et l'inclure dans l'application. Mais installer le routeur n'est pas suffisant. Il faut aussi lui fournir une configuration, qui définit des associations entre URLs et composants : les routes. Faisons tout cela dans un fichier dédié, qu'on nommera `router.ts` :

`router.ts`

```
import { createWebHistory, createRouter } from 'vue-router';
import Home from '@/views/Home.vue';
import Races from '@/views/Races.vue';
const router = createRouter({
  history: createWebHistory(),
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/races',
      name: 'races',
      component: Races
    },
  ]
});

export default router;
```

L'objet exporté `router` doit ensuite être utilisé quand nous créons notre application, dans `main.ts` :

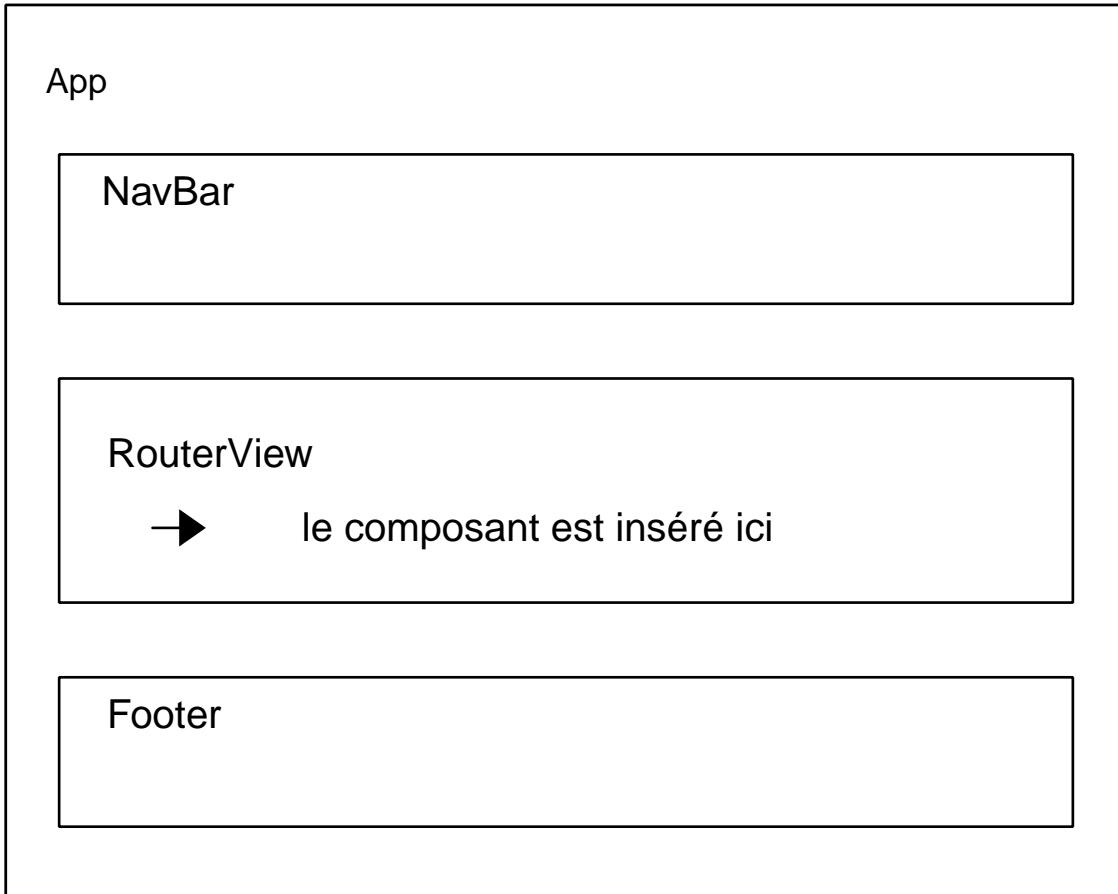
`main.ts`

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

Comme tu peux le voir, les `routes` sont un simple tableau d'objets, chaque objet étant ... une route. Une route est généralement un triplet de propriétés :

- `path` : le chemin de l'URL pour lequel il faut afficher le composant ;
- `name` : un nom permettant d'identifier la route, que nous utiliserons plus tard ;
- `component` : le composant initialisé et affiché lorsqu'on navigue vers le chemin donné.

Tu te demandes sans doute où le routeur insère le composant de la route dans la page. En général, toutes les routes de l'application partagent des parties communes : la barre de navigation, le pied de page, etc. Il serait dommage d'avoir à les inclure dans chacun des composants associés à une route. Pour ne pas avoir à faire ça, le routeur insère le composant de la route courante (par exemple, le composant `Races` dans l'exemple ci-dessus), à l'emplacement marqué par un tag spécifique, en général placé dans le template du composant racine : `RouterView`.



`RouterView` est bien sûr un composant View, fourni par `vue-router`. Son unique responsabilité est d'agir comme un conteneur pour la vue du composant de la route courante.

Le template du composant racine ressemble donc à ceci :

```
<div>
  <NavBar />
  <RouterView />
  <Footer />
</div>
```

Quand on naviguera d'une route à l'autre, tout restera donc affiché (c'est-à-dire la *Navbar* et le *Footer*) excepté le contenu de `RouterView`, qui sera remplacé par la vue du composant de la nouvelle route.

Tu n'as pas besoin de déclarer `RouterView` dans le composant `App` : l'instruction `createApp(App).use(router)` qu'on a ajoutée plus tôt permet non seulement d'enregistrer le plugin `router` dans l'application, mais aussi d'enregistrer globalement son composant `RouterView`.

20.2. Navigation

Comment naviguer d'une route à une autre ?

On peut bien sûr taper manuellement l'URL de la route dans la barre d'adresse et recharger la page, mais ce n'est pas franchement le plus simple ni le plus efficace. Et on ne veut pas non plus utiliser des liens classiques (``). Cliquer sur un tel lien aurait le même effet : demander au navigateur de charger une nouvelle page, et donc redémarrer l'application. Le but du routeur est précisément d'éviter de tels rechargements : on veut créer une *Single Page Application*.

La solution, c'est d'utiliser le composant `RouterLink`, et de lui fournir via sa *prop* nommée `to`, le chemin de la route vers laquelle on veut naviguer. On peut aussi lui passer un tableau de chaînes de caractères, représentant le chemin et ses paramètres. Par exemple, dans le template du composant `Races`, si on veut naviguer vers le composant `Home`, on peut inclure le lien suivant :

Races.vue

```
<RouterLink to="/">Home</RouterLink>
```

On peut aussi, plutôt que d'utiliser un chemin, passer le `name` qui identifie la route :

Races.vue

```
<RouterLink :to="{ name: 'home' }">Home</RouterLink>
```

Le composant `RouterLink` ajoute une classe CSS `.router-link-active` (ou une classe de ton choix) automatiquement au lien qu'il génère s'il pointe vers la route courante. Ca permet, par exemple, de styler un élément de menu lorsqu'il pointe vers la page courante.

Il est aussi possible de naviguer vers une route programmatiquement, depuis le code TypeScript du composant. On utilise pour cela la méthode `push()` de l'objet `router` retourné par `useRouter()`. C'est utile pour, par exemple, rediriger l'utilisateur après une soumission de formulaire :

Races.vue

```
const router = useRouter();
function saveAndMoveBackToHome() {
    // ...
    router.push('/');
    // or
    router.push({ name: 'home' });
}
```

La méthode accepte un chemin, sous forme de `string`, ou un objet contenant le chemin ou le nom de la route vers laquelle naviguer.

L'objet `router` expose aussi une méthode `replace()`, pour le cas où on désire remplacer l'URL courante dans l'historique par une nouvelle URL, plutôt que d'ajouter une nouvelle entrée.

20.3. Paramètres

Bien entendu, dans une application, nous aurons besoin de chemins dynamiques, contenant des paramètres. Par exemple, si on veut afficher une page d'informations concernant un poney dans une course, on devra spécifier dans son chemin l'identifiant du poney en question et l'identifiant de la course : `/races/:raceId/ponies/:ponyId`.

Définissons une route dans la configuration, avec ces deux paramètres.

router.ts

```
{
  path: '/races/:raceId/ponies/:ponyId',
  name: 'pony',
  component: Pony
},
```

Pour naviguer vers cette route, on devra fournir la valeur de ces deux paramètres à `RouterLink` :

Races.vue

```
<RouterLink :to="{ name: 'pony', params: { raceId: race.id, ponyId: pony.id } }>
Pony</RouterLink>
```

Et bien sûr, le composant cible aura besoin de connaître la valeur de ces deux paramètres, pour pouvoir obtenir les informations du serveur. Pour récupérer la valeur des paramètres, l'objet `route` retourné par `useRoute()` représente la route courante et peut être utilisé dans notre `setup`. C'est une référence réactive, avec une propriété `params` qui contient les paramètres de la route.

Pony.vue

```
const pony = ref<PonyModel | null>(null);
// ... fetch the pony details
const route = useRoute();
const ponyId = route.params.ponyId as string;
getPony(ponyId).then(fetchedPony => (pony.value = fetchedPony));
```



Le routeur réutilisera les composants quand il le peut ! Supposons par exemple que le composant contienne un lien "Suivant", permettant d'afficher les informations du poney suivant dans la course. Le chemin passera donc de `/ponies/1` à `/ponies/2` (par exemple) lorsque l'utilisateur cliquera sur ce lien. Dans ce cas, comme ce n'est pas la route qui change, mais uniquement ses paramètres, le routeur ne détruit pas le composant `Pony` pour immédiatement en reconstruire un autre. Il laisse simplement le composant `Pony` en place. Cela signifie que la fonction `setup` ne sera pas rappelée ! Donc, pour que le composant rafraîchisse ses informations, il faudra qu'il utilise un *watcher* ou une *computed property* basée sur `route` !

Pony.vue

```
const route = useRoute();
const pony = ref<PonyModel | null>(null);
watchEffect(
  // every time the ID changes
  // use a watcher to fetch the pony details
  async () => {
    const ponyId = route.params.ponyId as string;
    pony.value = await getPony(ponyId);
  }
);
```

Vous pouvez également récupérer les paramètres de requête grâce à la propriété `query`.

20.4. Routeur et Suspense

`RouterView` propose aussi une API `v-slot`, qui nous permet d'écrire :

```
<RouterView v-slot="{ Component }">
  <component :is="Component" />
</RouterView>
```

`<component is="Races">` est une syntaxe de Vue pour charger un component, ici `Races`. `is` peut bien sûr être dynamique, permettant donc de charger un composant dynamiquement : `<component :is="Component">` avec `Component` la variable contenant le composant à charger.

Cela peut paraître accessoire au premier abord, mais c'est très pratique pour quelques scénarios

avancés, comme lorsque l'on veut faire des animations ou pour utiliser le routeur avec **Suspense** :

```
<RouterView v-slot="{ Component }">
  <Alert v-if="error" variant="danger">An error occurred while loading.</Alert>
  <Suspense v-else timeout="0">
    <component :is="Component" />
    <template #fallback>Loading...</template>
  </Suspense>
</RouterView>
```



Essaie notre [quiz](#) et l'exercice [Router](#) pour apprendre comment configurer le routeur, naviguer entre composants, et tester tout ce que nous venons de découvrir.

20.5. Passer les paramètres comme *props*

Le routeur permet de passer les paramètres de la route au composant en tant que *props*. Cela découpe le composant du routeur, et le rend donc utilisable en tant que composant principal d'une route, mais aussi en tant que simple composant utilisé à l'intérieur d'un autre composant.

Pour ce faire, il suffit d'ajouter **props: true** dans la définition de la route :

```
{
  path: '/users/:userId',
  name: 'user',
  component: User,
  props: true
},
```

Notre composant **User** peut maintenant déclarer et utiliser la *prop* **userId** :

User.vue

```
const props = defineProps<{
  userId: string;
}>();

const user = ref<UserModel | null>(null);
// ... fetch the user details
getUser(props.userId).then(fetchedUser => (user.value = fetchedUser));
```

Comme tu peux le voir, il n'y a plus aucune référence au routeur dans le composant.

Tu peux aussi, dans la route, passer un objet au lieu d'un booléen à **props**. Cela permet de passer directement des valeurs statiques au composant.

20.6. Redirections

Il est assez courant de vouloir rediriger d'une URL vers une autre. Par exemple, le chemin racine `/` pourrait rediriger vers le chemin `/breaking` listant les toutes dernières nouvelles. Ou bien, après une réorganisation de l'application, on voudrait rediriger une ancienne URL, qui n'existe plus, vers l'URL qui la remplace. C'est possible en utilisant :

```
{  
  path: '/news',  
  redirect: '/breaking'  
},
```

20.7. Sélection de la route

Tu peux définir une route qui sera activée si aucune autre route ne l'est :

```
{  
  path: '/:catchAll(.*)',  
  redirect: '/404'  
}
```

L'astérisque peut aussi être utilisé pour définir des modèles de chemins tels que `path: '/pony-*`', qui sera sélectionnée si le chemin est `/pony-rainbow-dash` ou `/pony-pinkie-pie` par exemple.

Note que le routeur choisit toujours la route la plus spécifique.

20.8. Routes imbriquées

On l'a vu un peu plus tôt : lorsqu'une route est activée, le composant de la route est inséré dans la page à l'emplacement marqué par le composant `RouterView`.

Ce mécanisme peut en fait être réutilisé avec des composants imbriqués. Imagine une page complexe, affichant le profil d'un poney. Cette page afficherait le nom du poney et son portrait dans sa partie supérieure, et aurait plusieurs onglets dans sa partie inférieure : un pour afficher l'extrait de naissance du poney, un autre pour son palmarès, et un troisième pour les articles de presse concernant le poney. Chaque onglet a sa propre URL, afin de pouvoir avoir des liens qui mènent directement à un onglet spécifique. Mais on ne veut pas recharger le poney et répéter son nom et son portrait sur chacun des trois composants.

La solution est d'utiliser un `RouterView` dans le template du composant `Pony`, et de définir une route mère avec trois routes filles, de cette manière :

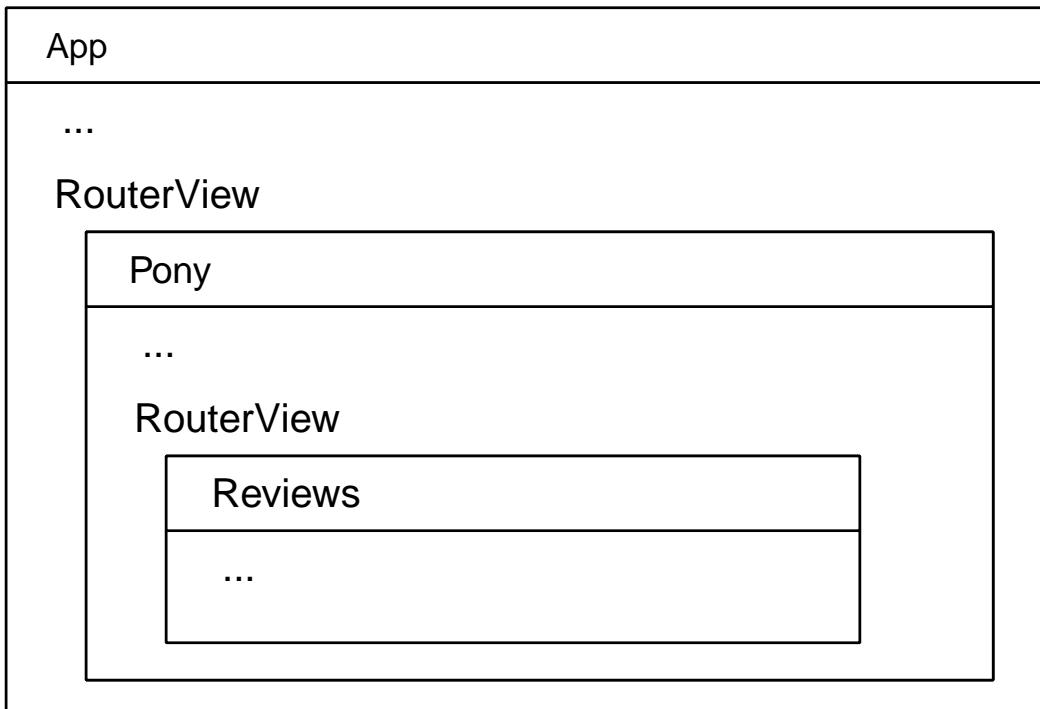
```
{  
  path: '/ponies/:ponyId',  
  component: Pony,  
  children: [
```

```

    { path: 'birth-certificate', component: BirthCertificate },
    { path: 'track-record', component: TrackRecord },
    { path: 'reviews', component: Reviews }
  ],
},

```

Lorsqu'on navigue vers le chemin `/ponies/42/reviews`, par exemple, le routeur insérera la vue du composant `Pony` à l'emplacement du `RouterView` contenu dans le composant `App`. Le template du composant `Pony`, en plus du nom et du portrait du poney contient un deuxième `RouterView`. C'est là que le composant `Reviews` sera inséré.



Lorsqu'on consulte le chemin `/ponies/42`, le composant `Pony` sera affiché, mais aucun des trois composants fils ne le sera. Ce serait préférable d'afficher l'extrait de naissance par défaut. Et c'est très simple en utilisant une route fille supplémentaire, avec un chemin vide, et redirigeant vers la route `birth-certificate`:

```

{
  path: '/ponies/:ponyId',
  component: Pony,
  children: [
    { path: '', redirect: 'birth-certificate' },
    { path: 'birth-certificate', component: BirthCertificate },
    { path: 'track-record', component: TrackRecord },
    { path: 'reviews', component: Reviews }
  ]
}

```

```
},
```

Note que, dans l'exemple ci-dessus, le chemin de la redirection est relatif au chemin `/ponies/:ponyId`, parce qu'il ne commence pas par un `/`.

Au lieu de rediriger, tu voudrais peut-être utiliser l'URL `/ponies/42` pour l'extrait de naissance. Cela peut aussi être accompli avec un chemin vide :

```
{
  path: '/ponies/:ponyId',
  component: Pony,
  children: [
    { path: '', component: BirthCertificate },
    { path: 'track-record', component: TrackRecord },
    { path: 'reviews', component: Reviews }
  ]
},
```

20.9. Gardes de navigation

Certaines des routes de l'application ne devraient être accessibles qu'aux personnes autorisées. Bien sûr, il est généralement préférable de désactiver ou masquer un lien si l'utilisateur n'est pas censé l'utiliser. Plus important encore, le backend doit vérifier les permissions de l'utilisateur avant de lui donner accès aux données protégées. Mais cela n'empêchera pas les utilisateurs d'accéder à des routes qui leur sont inaccessibles, simplement en rentrant leur adresse, ou en utilisant un marque-page.

C'est ici que les gardes de navigation entrent en jeu. Il en existe de trois sortes :

- les gardes globaux : ils s'appliquent à toutes les routes de l'application ;
- les gardes de routes : ils s'appliquent aux routes sur lesquelles ils sont définis ;
- les gardes *in-component* : ils s'appliquent aux composants sur lesquels ils sont définis.

Commençons par les gardes globaux !

20.9.1. Gardes globaux

Les gardes globaux sont enregistrés directement sur l'objet `router`. Ils sont appelés par le routeur lorsque la route change (c'est-à-dire pas lorsque les paramètres changent, uniquement quand le routeur s'apprête à activer une autre route).

`beforeEach`

Le plus commun des gardes est `beforeEach` :

```
router.beforeEach((to: RouteLocationNormalized, from: RouteLocationNormalized) => {
  return to.name === 'login' || isLoggedIn;
```

```
// or `return to.name === 'login' || isLoggedIn || '/login'` to redirect
});
```

Cet exemple montre que le garde est une fonction qui prend deux paramètres :

- **from** est la **Route** d'où on vient ;
- **to** est la **Route** où on va.

Le garde montré dans l'exemple ci-dessus est très simple : si l'utilisateur essaye de se rendre sur la page de "Login" ou est déjà authentifié, alors on retourne **true** (ou rien) pour approuver la navigation. Sinon, on retourne **false** pour l'empêcher. On peut retourner `'/login'` ou `{ name: 'login' }` pour empêcher la navigation et rediriger l'utilisateur vers une autre route. On peut également retourner une **Promise** si la décision est asynchrone.

Et c'est en fait tout l'intérêt d'un garde. Empêcher l'accès à une route sans expliquer aux utilisateurs pourquoi ils ne peuvent y accéder et ce qu'ils doivent faire pour y accéder (dans ce cas, s'authentifier) n'est pas très ergonomique. Un garde est un outil qui est destiné à améliorer l'expérience d'utilisation, et non un outil de sécurité. Empêcher l'accès à des ressources protégées est le rôle du serveur, pas celui du client.

Note que tu peux définir plusieurs gardes **beforeEach**. Le routeur les exécutera tous, et n'activera la route que si tous les gardes approuvent la navigation.

afterEach

On peut définir aussi un garde global **afterEach**. Mais il s'agit plutôt d'un *hook* que d'un garde : il ne peut pas empêcher la navigation, mais permet seulement d'exécuter du code après que la navigation a été confirmée.

20.9.2. Gardes de route

On peut définir un seul type de garde sur une route : **beforeEnter**.

beforeEnter

Ce garde sera invoqué après tous les gardes globaux (s'il y en a). Sa signature est la même que le garde global **beforeEach** :

```
{
  path: '/search',
  component: Search,
  beforeEnter: (
    to: RouteLocationNormalized,
    from: RouteLocationNormalized
  ): boolean | RouteLocationNormalized | string => {
    // check that the user has the permissions to see the races
  }
},
```

20.9.3. Gardes de composants

Enfin, à l'intérieur d'un composant, on peut définir deux types de gardes sous forme de fonctions :

- **onBeforeRouteUpdate** : c'est le seul garde appelé lorsque les paramètres de la route changent. Il constitue une alternative à un *watcher* sur les paramètres de la route ;
- **onBeforeRouteLeave** : ce garde est appelé lorsque le composant est sur le point d'être remplacé par un autre à cause d'une navigation. Il est utile pour demander confirmation à l'utilisateur avant de quitter un long formulaire par exemple, et de perdre les données patiemment saisies si l'utilisateur n'a pas vraiment l'intention de quitter la page.

```
const saved = ref(false);
// ...
onBeforeRouteLeave(() => {
  if (!saved.value) {
    return confirm('The form has not been saved. Are you sure you want to leave the current page?');
  }
});
// ...
```

20.10. Meta information

Il est également possible de définir des métadonnées sur une route.

```
{
  path: '/search',
  component: Search,
  meta: {
    requiresAuth: true
  }
},
```

C'est un mécanisme intéressant pour pouvoir définir un garde.

```
router.beforeEach((to: RouteLocationNormalized, from: RouteLocationNormalized) => {
  if (to.meta.requiresAuth && !isLoggedIn) {
    return '/login';
  }
  return true;
});
```

20.11. Tests avec vue-router-mock

Il est possible de tester des composants qui utilisent le routeur grâce à Vitest et en mockant

`useRoute`, `useRouter`, etc. Mais Eduardo "posva" San Martin, le créateur du routeur, a aussi fait une mini-bibliothèque qui permet de simplifier l'écriture des tests : `vue-router-mock`.

`vue-router-mock` permet notamment de créer un faux routeur dans les tests avec `createMockRouter`, et de l'injecter dans les composants testés avec `injectRouterMock`. On peut ensuite appeler `setParams()` dans les tests pour changer les paramètres de la route, et vérifier que les méthodes de navigation ont bien été appelées par exemple :

```
const router = createRouterMock({
  spy: {
    create: fn => vi.fn(fn),
    reset: spy => spy.mockRestore()
  }
});

beforeEach(() => {
  injectRouterMock(router);
  router.setParams({
    raceId: '1',
    ponyId: '2'
  });
});

test('should display the race and navigate back', async () => {
  const wrapper = mount(Races);

  const button = wrapper.get('button');
  expect(button.text()).toContain('Back to home');
  await button.trigger('click');

  expect(router.push).toHaveBeenCalledWith('/');
});
```

Très pratique, c'est ce qu'on utilise dans les exercices si vous voulez d'autres exemples !

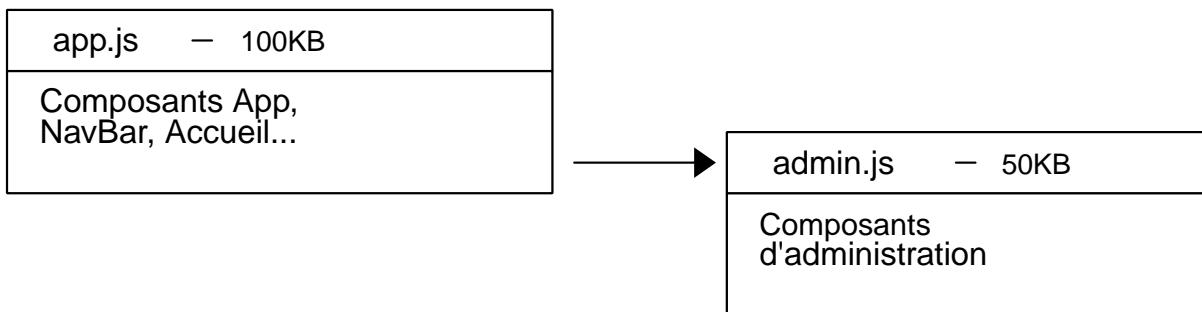


Essaie nos exercices [Gardes 🦄](#) et [Routes imbriquées et redirection 🦄](#) pour apprendre à utiliser ces fonctionnalités plus avancées du routeur.

Chapter 21. Chargement à la demande

À mesure que l'application grandit en taille et en fonctionnalités, charger toute l'application d'un coup peut devenir problématique : le *bundle* de l'application devient trop volumineux. Son chargement puis son analyse par le navigateur deviennent trop longs. Par ailleurs, certaines parties de l'application ne sont souvent utilisées que par certains utilisateurs, ou sont simplement peu utilisées. Les charger systématiquement au démarrage constitue donc une perte de temps et un gaspillage de bande passante. C'est là que le chargement à la demande (*lazy loading*) est utile, voire nécessaire.

Le chargement à la demande consiste à diviser l'application en plusieurs *bundles* JavaScript, et à ne charger ces *bundles* que lorsque c'est nécessaire.



Le chargement à la demande est configuré au niveau composant. Cela signifie que chaque composant peut être chargé à la demande, soit lorsqu'il doit être affiché à l'intérieur de la vue d'un autre composant, soit lorsqu'il est affiché par une route et que l'utilisateur navigue vers cette route. Dans les deux cas, la même technique est utilisée pour charger le composant paresseusement : un import JavaScript/TypeScript dynamique.

21.1. Composants asynchrones

Commençons par le premier cas : un composant est initialement absent de la vue parce qu'il est à l'intérieur d'un `v-if`, et on veut le charger et l'afficher dès que la condition du `v-if` devient vraie.

App.vue

```
<div>
  <h1>Ponyracer</h1>
  <button @click="showRaces = true">Show races</button>
  <div v-if="showRaces">
    <PendingRaces />
  </div>
</div>
```

Nous avons appris que, lorsqu'un composant parent utilise un composant enfant dans son template, nous devons mentionner le composant enfant dans la propriété `components` du parent (ou

juste les importer avec `script setup`).

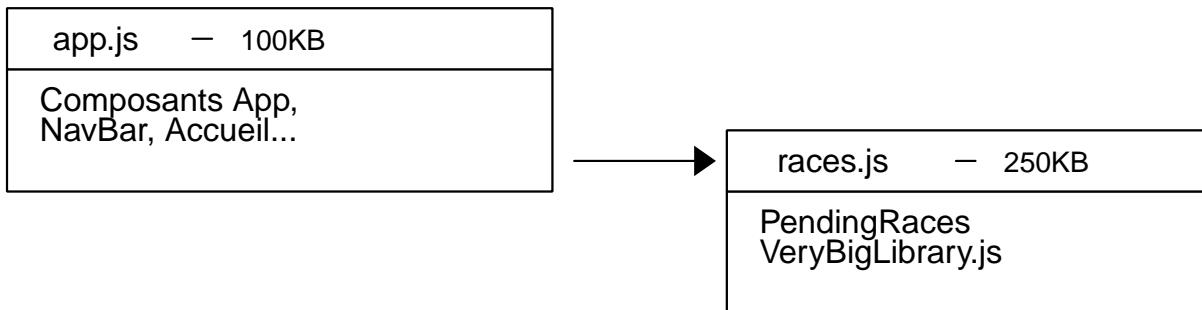
Il s'avère que l'on peut simplement remplacer cette déclaration par un appel à `defineAsyncComponent` avec un import dynamique, au lieu d'utiliser un import statique :

App.vue

```
const PendingRaces = defineAsyncComponent(() => import('./PendingRaces.vue'));
```

Si tu utilises Webpack (et c'est le cas lorsque tu utilises Vue CLI) ou Rollup et Vite (si tu as généré ton projet avec `create-vue`) alors le code du composant `PendingRaces` sera placé par le bundler dans un fichier JavaScript séparé, qui ne sera chargé que si nécessaire.

Dans ce cas précis, le gain est minime. Mais imagine que `PendingRaces` utilise une bibliothèque volumineuse. Le bundler sera assez malin pour placer le composant et cette bibliothèque dans ce même *bundle* distinct. Et tout ce code ne sera donc chargé que si le composant est affiché. Le *bundle* principal s'en trouve donc allégé d'autant, ce qui donne un coup de boost au démarrage de l'application.



21.2. Composants asynchrones et `Suspense`

Nous avons montré plus tôt comment on pouvait utiliser `Suspense` pour afficher une alternative pendant qu'un composant charge dynamiquement les données dont il a besoin. On peut aussi utiliser `Suspense` pour afficher une alternative pendant qu'un composant asynchrone lui-même est chargé paresseusement.

Illustrons cela avec le composant asynchrone `PendingRaces` :

App.vue

```
<div>
  <h1>Ponyracer</h1>
  <Suspense>
    <PendingRaces />
    <template #fallback>Loading...</template>
  </Suspense>
```

```
</div>
```

L'écran affichera automatiquement **Loading** pendant que le composant se charge.

On peut également avoir un comportement similaire sans utiliser **Suspense**. **defineAsyncComponent** accepte, outre un import dynamique, un objet permettant de définir :

- un **loadingComponent**, pour spécifier ce qu'il faut afficher pendant le chargement ;
- un **errorComponent**, pour spécifier ce qu'il faut afficher en cas d'erreur ;
- un **delay**, pour spécifier combien de temps attendre avant d'afficher le **loading** ;
- un **timeout**, pour spécifier combien de temps attendre avant de considérer que le chargement a échoué et afficher l'erreur. (le composant s'affichera quand même s'il finit par se charger).

Par défaut, **delay** vaut 100ms, et il n'y a ni **timeout**, ni **loadingComponent**, ni **errorComponent**.

App.vue

```
const PendingRaces = defineAsyncComponent({
  loader: () => import('./PendingRaces.vue'),
  delay: 100,
  timeout: 2000,
  loadingComponent: defineComponent({ template: 'Loading' }),
  // or just use another component or a simple function like the following
  errorComponent: () => 'An error occurred while loading'
});
```

21.3. Chargement à la demande par le routeur

Un cas d'utilisation plus courant est de charger à la demande un composant associé à une route : lorsque l'utilisateur clique sur un lien pour changer d'écran (et donc de route), le code du composant de la route est d'abord chargé paresseusement, puis la navigation est effectuée.

La mise en œuvre est là aussi très simple : on utilise la même syntaxe d'import dynamique, mais dans la déclaration de la route :

router.ts

```
{
  path: '/races',
  component: () => import('@/views/Races.vue')
},
```

Et voilà ! Le bundler créera un *bundle* séparé pour le code du composant **Races**, et ce *bundle* ne sera chargé que lorsque l'utilisateur naviguera vers la route **/races**.

21.4. Grouper plusieurs composants dans le même bundle

Passer d'un *bundle* unique à un *bundle* par composant n'est pas nécessairement la meilleure chose à faire. Cela peut avoir du sens de charger en une seule requête plusieurs composants d'un seul coup.

Avec Vite, tu dois déclarer quels imports tu veux grouper dans la configuration Vite, en utilisant la propriété `build.rollupOptions`:

`vite.config.ts`

```
export default defineConfig({
  plugins: [
    vue({
      script: {
        propsDestructure: true,
        defineModel: true
      }
    }),
  ],
  build: {
    rollupOptions: {
      // https://rollupjs.org/guide/en/#outputmanualchunks
      output: {
        manualChunks: {
          races: [
            './src/views/Races.vue',
            './src/views/Pony.vue'
            // all other imports to group...
          ]
        }
      }
    }
  }
});
```



Essaie notre exercice [Lazy-loading 🐾](#) pour mettre en pratique le chargement à la demande et mesurer les gains de bande passante qu'il permet. Il y a aussi un [quiz 🎯](#) sur le lazy-loading et le routeur !

Chapter 22. Formulaires

Les formulaires sont compliqués : il faut valider les valeurs saisies par l'utilisateur et afficher les messages d'erreurs adéquats. Certains champs sont obligatoires, d'autres non. Certains dépendent de la valeur d'autres champs, et il faut donc réagir aux changements, etc.

Vue offre une directive que nous n'avons pas encore rencontrée : `v-model`. Elle nous aide à construire des formulaires, mais pas bien plus. Pour ce qui est de la validation, des messages d'erreur, etc., tu te débrouilles ! Heureusement, il existe des librairies complémentaires pour nous aider.

Nous allons commencer par un exemple qui utilise `v-model`, puis nous verrons ce qu'une librairie additionnelle peut nous apporter.

22.1. La directive `v-model`

La directive `v-model` peut être appliquée à un `input`, une `textarea` ou un `select`. Tu lui donnes une propriété à modifier, et elle se charge automatiquement de faire la synchronisation entre la valeur de la propriété dans le code, et la valeur du champ de formulaire.

C'est ce qu'on appelle le *binding bidirectionnel (two-way binding)* :

- si le code change la valeur de la propriété, la valeur affichée ou sélectionnée par le champ du formulaire change ;
- si l'utilisateur saisit une nouvelle valeur dans le champ de formulaire, la valeur de la propriété est mise à jour.

Register.vue

```
<!-- 'user.name' is updated every time the user enters a value -->
<!-- every time the code changes 'user.name', v-model changes the input value -->
<!-- the binding is thus two-way -->
<input v-model="user.name" />
```

Register.vue

```
const user = reactive({
  name: '',
  age: 18,
  profile: null,
  isAdmin: false
});
```

`v-model` est en fait du sucre syntaxique, équivalent (pour un élément `input`) à :

Register.vue

```
<input :value="user.name" @input="user.name = ($event.target as
```

```
HTMLInputElement).value" />
<!-- the `as HTMLInputElement` part is just to help template type checking tools -->
<!-- this is the same as `$event.target.value` -->
```

Grâce à cet exemple, on peut comprendre le fonctionnement du binding bidirectionnel :

- si `user.name` change, la propriété `value` de l'input est modifiée
- si l'utilisateur saisit une valeur dans le champ, un événement `input` est émis, et son gestionnaire modifie la valeur de `user.name`.

Cela fonctionne aussi avec des cases à cocher :

Register.vue

```
<input v-model="user.isAdmin" type="checkbox" />
<!-- displays "Is admin: true" or "Is admin: false" -->
<p>Is admin: {{ user.isAdmin }}</p>
```

Parfois, la valeur de la propriété doit être différente de `true` ou `false`. Dans ce cas, `v-model` permet de spécifier `true-value` et `false-value` :

Register.vue

```
<input v-model="user.isAdmin" type="checkbox" true-value="yes" false-value="no" />
<!-- displays "Is admin: yes" or "Is admin: no" -->
<p>Is admin: {{ user.isAdmin }}</p>
```

Les boutons radio peuvent également utiliser `v-model`, avec des options statiques ou dynamiques :

Register.vue

```
const profiles = ['developer', 'accountant', 'manager'];
const user = reactive({
  name: '',
  age: 18,
  profile: null,
  isAdmin: false
});
```

Register.vue

```
<!-- displays one radio button per profile -->
<input
  v-for="profile in profiles"
  :key="profile"
  v-model="user.profile"
  type="radio"
  name="profile"
  :value="profile"
```

```
/>  
<!-- displays "Profile: developer" or "Profile: accountant", etc. -->  
<p>Profile: {{ user.profile }}</p>
```

Et bien sûr, on peut faire de même avec un `select`, et des options statiques ou dynamiques :

Register.vue

```
<select v-model="user.profile">  
  <!-- displays a default static option -->  
  <option :value="undefined">None</option>  
  <!-- displays one option profile -->  
  <option v-for="profile in profiles" :key="profile" :value="profile">{{ profile }}</option>  
</select>  
<!-- displays "Profile: developer" or "Profile: accountant", etc. -->  
<p>Profile: {{ user.profile }}</p>
```

Note que ça fonctionne également avec un objet comme modèle.

Pour soumettre le formulaire, tu peux écouter l'événement `submit` et empêcher la soumission par défaut avec `prevent` :

Register.vue

```
<form @submit.prevent="register()">
```

Dans la fonction `register` tu auras accès à la propriété `user`, remplie avec les valeurs saisies.

Vue permet en outre d'ajouter quelques *modifiers* à `v-model`.

22.1.1. Le modifier `.trim`

Tu peux ajouter `.trim` à `v-model` pour enlever les espaces indésirables :

Register.vue

```
<input v-model.trim="user.name" />
```

Si l'utilisateur saisit une valeur avec des espaces au début ou à la fin, alors `user.name` recevra la valeur sans ces espaces.

22.1.2. Le modifier `.number`

Ajouter le *modifier* `.number` à `v-model` permet de convertir la valeur en nombre (`type="number"` sur l'input n'est pas suffisant : la valeur sera quand même une chaîne de caractères) :

Register.vue

```
<input v-model.number="user.age" type="number" />
```

Si l'utilisateur saisit `22`, alors `user.age` aura la valeur `22` et non `'22'`. Mais si la valeur saisie n'est pas convertible en nombre par Vue, alors elle sera stockée telle quelle. Par exemple, si l'utilisateur saisit `Not a number`, la valeur liée sera une chaîne vide si le champ est de type `number`, ou `'Not a number'` sinon.

22.1.3. Le modifier `.lazy`

Lorsque j'ai montré plus tôt le principe du binding bidirectionnel sans le sucre syntaxique de `v-model`, j'ai utilisé :

Register.vue

```
<input :value="user.name" @input="user.name = ($event.target as  
HTMLInputElement).value" />  
<!-- the 'as HTMLInputElement' part is just to help template type checking tools --&gt;<br/><!-- this is the same as '$event.target.value' --&gt;</pre>
```

On peut changer le comportement de `v-model` en écoutant l'événement `change` au lieu de l'événement `input`, simplement en ajoutant le *modifier* `.lazy`.

Cela signifie que `v-model` ne mettra à jour la valeur de la variable qu'au moment où le champ perdra le focus.

Register.vue

```
<input v-model.lazy="user.name" />
```

`v-model` fait le job pour les cas d'utilisation simples. Mais les limites sont vite atteintes. Qu'en est-il de la validation des valeurs saisies, ou du formulaire complet ? Des messages d'erreur ?

On peut gérer ces aspects soi-même, comme nous le ferons dans l'exercice sur `v-model`. On peut aussi utiliser une bibliothèque tierce pour nous y aider.

Il en existe deux populaires dans l'écosystème de Vue :

- [Vuelidate](#)
- [VeeValidate](#)

Elles ont des fonctionnalités similaires, et toutes deux sont de bons choix. Mais VeeValidate est ma préférée, et nous allons la découvrir plus en détails ensemble !



Essaie notre exercice [Register](#) ! Il fait partie de notre Pro Pack, et tu y apprendras comment mettre en oeuvre un formulaire complet avec `v-model`. Dans cet exercice, tu seras responsable de la validation et des messages d'erreur !

22.2. De meilleures formulaires avec VeeValidate

Écrire notre application en JavaScript nous permet de faire de la validation côté client. Ce n'est pas un remplacement pour la validation côté serveur, mais c'est un plus pour les utilisateurs de savoir instantanément que le formulaire est invalide, sans avoir à attendre de soumettre le formulaire et obtenir une réponse du serveur.

VeeValidate est dédiée à cette validation côté client. Elle fonctionne avec la directive `v-model` que nous venons de découvrir, et y ajoute quelques fonctionnalités de validation bien pratiques.

VeeValidate vient avec de nombreuses règles de validation, mais aucune d'elles n'est incluse dans l'application par défaut. Elles sont fournies dans un package séparé appelé `@vee-validate/rules`. Cela permet de garder l'application aussi légère que possible, mais nous oblige à déclarer explicitement quelles règles de validation doivent être incluses. Tu peux les importer toutes d'un coup, mais ce n'est pas une très bonne idée, parce que tu n'en utiliseras sans doute que quelques-unes.

Voici les règles de validation disponibles qui sont le plus souvent utilisées :

- `required` pour rendre un champ obligatoire ;
- `min:N` et `max:N` pour s'assurer qu'une valeur saisie a au moins/au plus N caractères ;
- `min_value:N`, `max_value:N`, `between` pour s'assurer qu'un nombre est au moins/au plus égal à N, ou est compris entre deux bornes ;
- `email` pour vérifier que l'utilisateur a saisi une adresse email valide ;
- `url` pour vérifier que l'utilisateur a saisi URL valide ;
- `alpha`, `alpha_num`, `alpha_dash`, `alpha_space` pour s'assurer qu'une valeur est alphabétique seulement/avec des nombres/avec des tirets/avec des espaces ;
- `numeric` pour vérifier que la valeur saisie est un nombre ;
- `regex` pour s'assurer qu'une valeur correspond à une expression régulière donnée.

Pour les `input type="file"`, il en existe d'autres spécifiques :

- `mimes` permet de définir quels types MIME sont acceptés pour le fichier ;
- `ext` permet de lister les extensions acceptées ;
- `size` permet de fixer une taille de fichier maximale ;
- `image` n'autorise que des formats d'image ;
- `dimensions` permet de définir les dimensions précises requises pour une image.

Certaines règles permettent de faire de la validation sur plusieurs champs :

- `required_if` rend un champ obligatoire si un autre est rempli avec une des valeurs données ;
- `confirmed` permet de s'assurer qu'un champ a la même valeur qu'un autre (double vérification classique pour un mot de passe).

Pour pouvoir utiliser une quelconque de ces règles de validation dans ton application, tu dois la

charger explicitement :

main.ts

```
import { defineRule } from 'vee-validate';
import { confirmed, min, required } from '@vee-validate/rules';

defineRule('min', min);
defineRule('required', required);
defineRule('confirmed', confirmed);
```

Comme tu peux le deviner grâce à cet exemple, nous pourrions renommer les règles de validation comme bon nous semble. Mais en règle générale, j'utilise les noms prédéfinis. Maintenant que nous avons chargé les règles désirées, nous pouvons les utiliser dans nos composants pour valider des champs de formulaire.

On peut aussi configurer VeeValidate, par exemple pour indiquer *quand* valider les champs. Par défaut, la librairie vérifie les champs sur les évènements `blur` et `change`. Mais j'aime bien valider sur l'évènement `input` également :

main.ts

```
import { configure } from 'vee-validate';
configure({
  validateOnInput: true,
});
```

VeeValidate propose deux modes d'utilisation :

- un mode basé sur l'API de Composition,
- un mode basé sur des composants à utiliser dans les templates (un mode de fonctionnement que l'on appelle des *Higher Order Components*, ou *HOC*, pour *composants de plus haut ordre*).

22.2.1. VeeValidate avec l'API de Composition

VeeValidate offre une fonction pour déclarer un champ de formulaire : `useField()`.

Register.vue

```
import { useField } from 'vee-validate';

const { value: name, errorMessage: nameErrorMessage, meta: nameMeta } = useField('
  name', { required: true });
```

L'objet retourné par `useField()` a plusieurs propriétés intéressantes, comme `value`, la valeur du champ, que l'on peut utiliser dans un `v-model`. VeeValidate gère également la validation du champ, et offre une propriété `errorMessage`, qui contient le potentiel message d'erreur du champ.

Register.vue

```
<template>
  <label for="name">Name</label>
  <input id="name" v-model="name" name="name" />
  <div v-if="nameMeta.dirty && !nameMeta.valid" class="error">{{ nameErrorMessage }}</div>
</template>

<script setup lang="ts">
import { useField } from 'vee-validate';

const { value: name, errorMessage: nameErrorMessage, meta: nameMeta } = useField('name', { required: true });
</script>
```

`useField` offre en fait plus de propriétés que ça grâce à `meta` :

- `valid` est un booléen qui indique si le champ est valide ou invalide ;
- `dirty` est un booléen qui indique si le champ est "vierge" (l'utilisateur n'a rien saisi) ou "sale" (l'utilisateur a saisi une valeur différente de la valeur initiale) ;
- `touched` est un booléen qui indique si le champ a perdu le focus ou pas ;

VeeValidate aide également à valider l'état du formulaire dans son ensemble, et fournit pour cela une autre fonction `useForm()`.

Register.vue

```
import { useField, useForm } from 'vee-validate';

const { meta: formMeta, handleSubmit, errors } = useForm();
// add a 'name' field to the form, initialized with 'JB'
const { value: name } = useField('name', { required: true }, { initialValue: 'JB' });
// add a 'password' field to the form
const { value: password } = useField('password', { required: true });
// the register function has access to the form values directly
const register = handleSubmit(values => {
  // this will only be called if the form is valid
  console.log(values);
});
```

Tu peux enregistrer autant de champs que tu le désires. `useForm` renvoie un objet avec une fonction `handleSubmit` qui nous sert à définir comment gérer la soumission, un object `errors` contenant les erreurs de validation, et les mêmes propriétés que `useField` dans une propriété `meta`. La logique est assez simple à comprendre : `valid` est `true` si tous les champs sont valides, et `false` si au moins un champ est invalide ; `dirty` est `false` si tous les champs sont "vierges", et `true` si au moins un champ est "sale", etc.

On peut par exemple désactiver le bouton de soumission du formulaire s'il est invalide :

```

<template>
  <form @submit="register()">
    <label for="name">Name</label>
    <input id="name" v-model="name" />
    <div class="error">{{ errors.name }}</div>

    <label for="password">Password</label>
    <input id="password" v-model="password" type="password" />
    <div class="error">{{ errors.password }}</div>

    <!-- button is disabled while the form is invalid -->
    <button :disabled="!formMeta.valid">Register</button>
  </form>
</template>

<script setup lang="ts">
  import { useField, useForm } from 'vee-validate';

  const { meta: formMeta, handleSubmit, errors } = useForm();
  // add a 'name' field to the form, initialized with 'JB'
  const { value: name } = useField('name', { required: true }, { initialValue: 'JB' });
  // add a 'password' field to the form
  const { value: password } = useField('password', { required: true });
  // the register function has access to the form values directly
  const register = handleSubmit(values => {
    // this will only be called if the form is valid
    console.log(values);
  });
</script>

```

L'API de composition proposée par VeeValidate fait le gros du travail en validant les champs et en calculant l'état du formulaire. C'est puissant et simple à utiliser. Et, comme nous allons le voir dans la partie suivante, il y a une autre façon, encore plus simple, d'utiliser VeeValidate.

22.2.2. VeeValidate avec des composants de plus haut ordre

Au lieu d'utiliser l'API de Composition, VeeValidate offre un composant pour définir un champ dans le template, et ajouter de la validation sur celui-ci. Ce composant s'appelle... **Field** ! **Field** ne fait pas grand-chose en lui-même. Mais on peut l'utiliser avec un autre composant offert par VeeValidate : **Form**. **Form** émet un événement **submit** avec les valeurs du formulaire :

```

<Form @submit="register($event)">
  <label for="name">Name</label>
  <Field id="name" name="name" rules="required" value="JB" />
  <label for="password">Password</label>
  <Field id="password" name="password" type="password" rules="required" />

```

```
<button type="submit">Register</button>
</Form>
```

Tu as juste à coder la fonction qui gère la soumission :

Register.vue

```
import { Field, Form } from 'vee-validate';

function register(values: Record<string, unknown>) {
    // { name: string; password: string }
    console.log(values);
}
```

Comme tu peux le voir, **Field** attend simplement un nom, et (par défaut) affiche un **input**. Si tu veux donner une valeur initiale au champ, tu peux utiliser la prop **value**. Ou tu peux aussi utiliser **v-model** : cela peut être très pratique si tu veux réagir à un changement de valeur ! Tu as alors simplement besoin d'une valeur réactive et d'un watcher.

Register.vue

```
const user = reactive({ name: 'Cédric', password: '' });

const passwordStrength = computed(() => computeStrength(user.password));
```

Et d'ajouter le **v-model** sur ton **Field** :

Register.vue

```
<Field id="password" v-model="user.password" name="password" type="password"
rules="required" />
<div id="strength">{{ passwordStrength }}</div>
```

On peut aussi spécifier une prop **type** (pour avoir un input de type différent, comme **type="number"**), ou carrément un élément différent avec la prop **as** (par exemple, **:as="textarea"**). Si tu veux afficher du HTML custom, peut-être afin d'ajouter une classe CSS quelque part si le champ est invalide, tu peux le faire facilement :

Register.vue

```
<Field v-slot="{ field, meta }" v-model="user.name" name="name" rules="required">
    <label for="name-input" :class="{ 'text-danger': meta.dirty && !meta.valid }">
        Name</label>
        <input id="name-input" :class="{ 'is-invalid': meta.dirty && !meta.valid }" v-
bind="field" />
    </Field>
```

Ce type de composant qui projette du contenu avec un *slot* est celui que nous avons déjà rencontré

dans le chapitre [Slots](#). Il est parfois appelé *Higher Order Component*, un composant de plus haut ordre.

`Field` expose une *slot prop*, contenant toutes les propriétés que `useField()` renvoie.

On peut accéder à la propriété `meta` comme tu peux le voir dans l'exemple précédent, mais aussi à d'autres propriétés comme `errorMessage`, qui contient le potentiel message d'erreur du champ. On peut même faire un peu mieux pour afficher ce message, car VeeValidate offre un troisième composant : `ErrorMessage`.

Register.vue

```
<Field v-slot="{ field, meta }" v-model="user.name" name="name" rules="required">
  <label for="name-input" :class="{ 'text-danger': meta.dirty && !meta.valid }">
    Name</label>
  <input id="name-input" :class="{ 'is-invalid': meta.dirty && !meta.valid }" v-
  bind="field" />
  <ErrorMessage name="name" class="error" />
</Field>
```

VeeValidate est suffisamment malin pour ne générer ce message que lorsque l'utilisateur a effectivement interagi avec le champ, donc il ne crie pas sur le pauvre utilisateur qui viendrait d'atterrir sur la page.

Le message affiché ici est `name is not valid.`, mais il peut être personnalisé lorsqu'on charge la règle de validation, en utilisant le package `@vee-validate/i18n`:

forms.ts

```
import { configure } from 'vee-validate';
import { localize } from '@vee-validate/i18n';

configure({
  validateOnInput: true,
  generateMessage: localize('en', {
    messages: {
      // use a function
      required: context => `The ${context.field} is required.`,
      confirmed: context => `The ${context.field} does not match.`,
      // or use the special syntax offered by the library
      min: `The ${field} must be at least ${min} characters.`,
    }
  })
});
```

Le message est maintenant `The name is required..`

Tu peux aussi renommer un champ pour un joli message d'erreur, avec la prop `label` :

Register.vue

```
<Field
  v-slot="{ field, meta }"
  v-model="user.confirmPassword"
  name="confirmPassword"
  rules="required|confirmed:@password"
  label="password confirmation">
  <label for="confirm-password-input" :class="{'text-danger': meta.dirty && !meta.valid }">Confirm password</label>
  <input
    id="confirm-password-input"
    type="password"
    :class="{'is-invalid': meta.dirty && !meta.valid }"
    v-bind="field"
  />
  <ErrorMessage name="confirmPassword" class="error" />
</Field>
```

Note qu'on peut bien sûr appliquer plusieurs règles d'un coup à un champ :

Register.vue

```
<Field v-slot="{ field, meta }" v-model="user.password" name="password"
  rules="required|min:3">
  <label for="password-input" :class="{'text-danger': meta.dirty && !meta.valid }">Password</label>
  <input id="password-input" type="password" :class="{'is-invalid': meta.dirty && !meta.valid }" v-bind="field" />
  <ErrorMessage name="password" class="error" />
</Field>
```

Form a aussi une slot prop avec différentes propriétés, comme `meta` qui contient l'état du formulaire (`valid`, `dirty`, etc.).

Register.vue

```
<Form v-slot="{ meta: formMeta }" @submit="register($event)">
  <!-- Several fields... -->
  <button :disabled="!formMeta.valid">Register</button>
</Form>
```

Un troisième composant, `FieldArray` est également disponible pour gérer des tableaux de valeurs. Pour terminer, sache que VeeValidate permet aussi de traduire les messages d'erreur dans la langue voulue, de définir ses propres règles de validation, d'ajouter des attributs d'accessibilité aux champs, etc.



VeeValidate vient avec un plugin pour les devtools, ce qui permet de directement

inspecter l'état de ton formulaire dans les devtools de ton navigateur pendant que tu développes 🎨.



Essaie notre exercice [Login](#) 🎯 ! Il fait partie du Pro Pack, et tu y apprendras à construire un formulaire complet validé par VeeValidate.

22.2.3. Validateurs custom

VeeValidate nous met à disposition un ensemble de règles, mais il est également possible de construire les nôtres.

Une règle de validation est assez simple : c'est une fonction qui renvoie `true` si la valeur est valide, ou `false` sinon. Il est aussi possible de retourner directement le message d'erreur, mais j'utilise généralement seulement un booléen comme type de retour. La fonction peut également renvoyer une `Promise` si tu as besoin de validation asynchrone (par exemple pour vérifier une valeur auprès du serveur).

Essayons donc de construire un validateur qui vérifie si une valeur est entre 18 et 130.

validators.ts

```
export function between18And130(value: string) {
  return +value >= 18 && +value <= 130;
}
```

Tu dois ensuite ajouter la règle, comme on le fait avec les règles fournies.

forms.ts

```
defineRule('between18And130', between18And130);
```

Et définir un message pour cette erreur :

forms.ts

```
between18And130: context => `The ${context.field} must be between 18 and 130.`,
```

On peut alors l'utiliser dans un composant :

Register.vue

```
<Field v-slot="{ field, meta }" name="age" rules="required|between18And130">
```

Un validateur peut aussi avoir des paramètres. Au lieu de coder en dur les valeurs minimum et maximum, nous pouvons les passer au validateur.

validators.ts

```
export function betweenParams(value: string, params: Array<string>) {  
  const min = +params[0];  
  const max = +params[1];  
  return +value >= min && +value <= max;  
}
```

On passe des paramètres en utilisant `:`, et `,` pour les séparer.

Register.vue

```
<Field v-slot="{ field, meta }" name="age" rules="required|betweenParams:17,120">
```

Le message d'erreur peut également refléter ces paramètres :

forms.ts

```
betweenParams: context => {  
  const params = context.rule!.params as Array<string>;  
  return `The ${context.field} must be between ${params[0]} and ${params[1]}.`;  
}
```

Une autre fonctionnalité assez cool est la possibilité de faire de la validation entre champs. Par exemple, notre validateur peut utiliser les valeurs définies dans deux autres champs pour les valeurs minimum et maximum.

Si le formulaire a des champs `min` et `max`, on peut écrire :

Register.vue

```
<Field v-slot="{ field, meta }" name="age" rules="required|betweenParams:@min,@max">
```

La syntaxe `@something` indique à VeeValidate qu'il faut utiliser la valeur du champ nommé `something`.

Pour résumer, il est assez facile de créer nos propres validateurs. On écrit une fonction qui renvoie vrai ou faux, on l'ajoute, on définit un message d'erreur, et on l'utilise, comme un validateur fourni par défaut.



Essaie notre exercice [Validateurs custom](#) ! Il fait partie du Pro Pack, et tu y apprendras à créer tes propres validateurs.

22.3. Composants de formulaire personnalisés

HTML définit un nombre important de contrôles standards : texte, mot de passe, case à cocher, etc. Mais parfois ces contrôles standards ne conviennent pas.

Vue permet de définir des composants bien sûr, et il est en fait possible d'en faire des éléments de formulaires, c'est-à-dire de les lier en leur appliquant la directive `v-model`.

Remplir le contrat de cette directive est assez simple. Il faut :

- accepter une prop appelée `modelValue`;
- notifier Vue que l'utilisateur a changé la valeur, en émettant un événement nommé `@update:modelValue`;

Nous allons illustrer tout cela en utilisant un composant *rating*. Ce composant permet de donner une note de 0 à 5 à un film, par exemple. Mais au lieu d'utiliser un champ de type `number` ou `range`, on voudrait que l'utilisateur clique simplement sur un bouton parmi 6 boutons affichés (qui, typiquement, seraient présentés sous forme d'étoiles, mais nous laisserons ça de côté dans l'exemple qui suit).

Voici le code d'un tel composant :

```
<script setup lang="ts">
defineProps<{
  modelValue: number;
}>();

const emit = defineEmits<{
  'update:modelValue': [value: number];
}>();

const pickableValues = [0, 1, 2, 3, 4, 5];

function setValue(pickedValue: number) {
  emit('update:modelValue', pickedValue);
}
</script>
```

Et voici son template :

```
<template>
  <div>
    <button
      v-for="pickableValue of pickableValues"
      :key="pickableValue"
      :class="{ selected: modelValue != null && pickableValue <= modelValue }"
      type="button"
      @click="setValue(pickableValue)"
    >
      {{ pickableValue }}
    </button>
  </div>
</template>
```

Et voilà. Nous disposons à présent d'un joli composant qui peut être utilisé dans n'importe quel formulaire, en lui appliquant simplement la directive `v-model` :

```
<form>
  <div>
    <label for="title">Title</label>
    <input id="title" v-model="movieTitle" />
  </div>
  <div>
    <label for="rating">Rating</label>
    <Rating id="rating" v-model="movieRating" />
  </div>
```

22.4. Macro `defineModel`

Quand tu as un composant de formulaire personnalisé qui doit seulement lier une valeur de `v-model` à un input classique, la mécanique de prop/event est parfois un peu pénible :

```
<template>
  <input :value="modelValue" @input="setValue(($event.target as
HTMLInputElement).value)" />
</template>

<script setup lang="ts">
defineProps<{ modelValue: string }>();
const emit = defineEmits<{ 'update:modelValue': [value: string] }>();
function setValue(newValue: string) {
  emit('update:modelValue', newValue);
}
</script>
```

Depuis Vue v3.3, il est possible de simplifier ce composant, en utilisant la macro (expérimentale) `defineModel` :

```
<template>
  <input v-model="modelValue" />
</template>

<script setup lang="ts">
const modelValue = defineModel<string>();
</script>
```

`defineModel` accepte également quelques options :

- `required: true` indique que la prop est obligatoire
- `default: value` permet de spécifier la valeur par défaut

- **local: true** indique que la prop est disponible et mutable même si le composant parent n'a pas passé le **v-model** correspondant

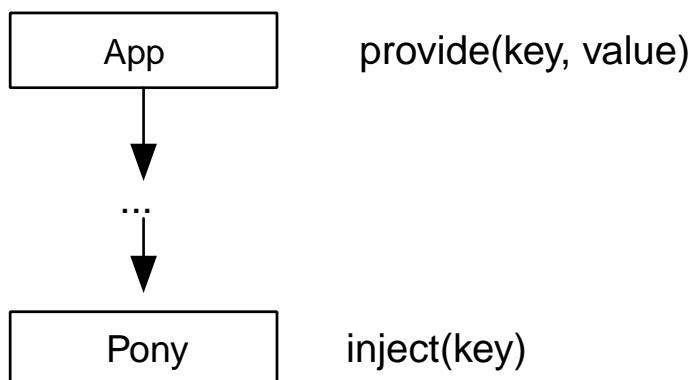
Chapter 23. Provide/inject

Vue dispose d'un système basique d'injection de dépendances. J'ai hésité à écrire ce chapitre, parce que ce n'est pas quelque chose que tu utiliseras tous les jours. C'est principalement utile pour les auteurs de bibliothèques, mais il y a néanmoins un cas d'utilisation où cela peut s'avérer utile dans les applications.

23.1. Une manière d'éviter les `props` passe-plat

Quand une application se complexifie, on se retrouve parfois à devoir passer des `props` depuis un composant vers ses petits-enfants ou ses arrière-petits-enfants. Les composants intermédiaires font "passe-plat" : ils reçoivent une `prop` de leur parent dans le seul but de la fournir à leurs enfants.

Une manière d'éviter cela est d'utiliser `provide/inject`. `provide` permet d'enregistrer une valeur associée à une clé dans un composant, et `inject` permet de récupérer cette valeur dans un descendant.



La clé peut être une `string` ou un `number` (un troisième type, `InjectionKey`, est autorisé, mais nous en parlerons plus tard). La valeur peut être n'importe quoi : une valeur statique, une valeur réactive, une fonction... Si la valeur a besoin de changer pendant la vie du composant, alors utiliser une valeur réactive fait sens :

App.vue

```
const color = ref('#123456');
provide('color', color);
```

Un composant, plus bas dans l'arbre, peut alors s'injecter la valeur.

Pony.vue

```
const color = inject<Ref<string>>('color');
```

Cette valeur peut être utilisée comme n'importe quelle autre propriété réactive. Si le composant `App` modifie la valeur de `color`, alors le composant `Pony` sera mis à jour !

Comme tu peux le voir, j'ai dû dire à TypeScript que la valeur retournée par `inject` pour cette clé est de type `Ref<string>`. Il est possible de faire mieux en utilisant une clé de type `InjectionKey<T>`. `InjectionKey` est en fait un `Symbol`, un type primitif qui a la particularité de créer des valeurs uniques. Cela en fait un type parfait pour une clé, puisqu'on a la garantie qu'une clé de ce type n'en écrasera jamais aucune autre :

```
const symbol1 = Symbol();
const symbol2 = Symbol('hello');
const symbol3 = Symbol('hello'); // symbol2 !== symbol3
```

Créons donc une `InjectionKey` :

injection-keys.ts

```
import { InjectionKey, Ref } from 'vue';

export const colorKey: InjectionKey<Ref<string>> = Symbol('color');
```

Et utilisons-là pour fournir la couleur :

App.vue

```
const color = ref('#123456');
provide(colorKey, color);
```

Il s'agit ensuite d'utiliser cette même clé pour s'injecter la valeur :

Pony.vue

```
const color = inject(colorKey);
// color is automatically typed as 'Ref<string> | undefined'
```

Note que tu peux passer un deuxième paramètre à `inject` pour obtenir une valeur par défaut :

Pony.vue

```
const color = inject(colorKey, ref('#999999'));
// color is automatically typed as 'Ref<string>'
```

On peut aussi fournir une valeur au niveau applicatif :

main.ts

```
createApp(App)
  .provide(colorKey, ref('#987654'))
```

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

23.2. Tester des composants qui utilisent inject

Pour pouvoir tester les composants qui utilisent `inject`, il va falloir leur fournir la valeur qu'ils attendent. Vue Test Utils permet de le faire avec l'option `provide` :

Pony.spec.ts

```
const wrapper = mount(Pony, {
  global: {
    provide: {
      color: ref('#999999')
    }
  }
});
```

ou, si tu as utilisé une `InjectionKey` :

Pony.spec.ts

```
const wrapper = mount(Pony, {
  global: {
    provide: {
      [colorKey as symbol]: ref('#999999')
    }
  }
});
```

23.3. Providers hiérarchiques

Pour être honnête, utiliser `provide/inject` pour éviter le passe-plat n'est pas vraiment nécessaire : on peut simplement utiliser une fonction `useColor()` qui retourne une couleur réactive et on obtient le même résultat.

Mais `inject` offre un autre avantage :

- il commence par regarder dans le composant parent pour savoir s'il a fourni une valeur pour la clé en question;
- sinon, il la cherche dans le grand-parent;
- etc.
- sinon, il la cherche dans le composant racine;
- et enfin, il la cherche dans les valeurs fournies au niveau applicatif.

Bien sûr, dès que la valeur est trouvée, la recherche s'arrête et la valeur trouvée est retournée.

À la fin, si aucun *provider* n'a fourni de valeur pour cette clé, `inject` retourne `undefined` et affiche un avertissement dans la console (à moins qu'on ait passé une valeur par défaut en deuxième argument à `inject`) :

```
'[Vue warn]: injection "Symbol(color)" not found'
```

Cette hiérarchie de *providers* peut s'avérer pratique si une valeur fournie à un niveau de l'arbre doit être remplacée par une autre pour un sous-ensemble de composants.

23.4. Plugins

Vue offre un mécanisme de *plugins*. Les *plugins* sont installés grâce à `app.use(plugin)`. Le routeur par exemple utilise ce mécanisme :

main.ts

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

Lorsqu'on veut obtenir le routeur dans un composant, on utilise `useRouter()` :

Races.vue

```
const router = useRouter();
function saveAndMoveBackToHome() {
  // ...
  router.push('/');
  // or
  router.push({ name: 'home' });
}
```

Sous le capot, le routeur, comme de nombreux *plugins*, utilise provide/inject !

Quand on crée le routeur avec `createRouter()`, la fonction fournit le routeur créé au niveau applicatif. Et le code de `useRouter()` est en fait très simple : `return inject(routerKey)`. En fait tu pourrais directement t'injecter le routeur de cette manière dans tes composants pour l'obtenir ☺.

C'est le cas d'utilisation le plus répandu pour provide/inject, et c'est comme cela que la plupart des *plugins* fonctionnent.

Chapter 24. Gestion d'état

Dans une application web bien conçue, l'état de l'application est préservé sur le serveur, et dans l'URL. On doit être capable de rafraîchir n'importe quelle page (et donc redémarrer l'application), et se retrouver là où on se trouvait juste avant : l'URL permet de retenir la page consultée, et les données affichées par cette page sont stockées sur le serveur et téléchargés grâce aux paramètres de l'URL.

Les différents composants de cette page cependant, une fois les données chargées depuis le serveur, doivent souvent partager un état. Une partie de cet état peut même être global à toute l'application, comme par exemple l'utilisateur ou l'utilisatrice connecté(e) : la barre de navigation doit afficher son nom, et lui permettre de se déconnecter ; la page d'accueil doit afficher un message d'accueil personnalisé ; n'importe quel composant peut en avoir besoin pour décider s'il ou elle a le droit d'exécuter une action donnée.

Alors, comment les composants partagent-ils leur état ?

On peut bien sûr utiliser les `props` : un composant parent passe son état ou une partie de son état à un composant enfant via des `props`, et l'enfant émet des événements à destination de son parent afin qu'il modifie son état. C'est approprié pour des associations parent-enfant, mais ce n'est pas très adapté quand l'état doit être partagé entre de nombreux composants frères, ou quand l'arbre des composants est profond. Et pour un état global, qui doit être partagé entre des composants qui n'ont rien à voir l'un avec l'autre et sont affichés par diverses routes, ce n'est pas le bon outil.

La solution dans ce genre de cas consiste à mettre en œuvre le pattern *store*.

24.1. Le pattern *store*

Supposons donc que deux composants aient besoin d'accéder aux données de l'utilisateur ou utilisatrice connecté(e). On peut utiliser l'API composition de Vue et déclarer une propriété réactive que les deux composants vont utiliser. Je place généralement cette propriété dans un "service", mais tu peux choisir de l'appeler *store*.

UserService.ts

```
import { ref } from 'vue';
import { UserModel } from '@/models/UserModel';

const userModel = ref<UserModel | null>(null);

async function authenticate(login: string, password: string): Promise<UserModel> {
    // call the HTTP API
    // in case of success, store the logged-in user
    userModel.value = response.data;
}

function logout(): void {
    // ...
    userModel.value = null;
```

```

}

// ...

export function useUserService() {
  return {
    userModel,
    authenticate,
    logout
  };
}

```

Une fois ce service et cette propriété définis, les composants peuvent y accéder très facilement :

Home.vue

```

<template>
  <div v-if="userModel">Hello {{ userModel.name }}!</div>
  <div v-else>Welcome, anonymous comrade!</div>
</template>

<script setup lang="ts">
  import { useUserService } from './UserService';

  const { userModel } = useUserService();
</script>

```

Et comme la propriété est réactive, c'est formidable : si la valeur de `userModel` change, tous les composants verront la modification automatiquement ✨.

Par exemple, si le composant `Navbar` a un bouton *Sign out* qui permet de se déconnecter :

Navbar.vue

```

const { userModel } = useUserService();
function signout() {
  userModel.value = null;
}

```

Le composant `Home` affichera automatiquement `Welcome, anonymous comrade!` quand l'utilisateur clique sur le bouton "Sign out" de la `Navbar`.

Cela dit, pour simplifier le debugging et pour encapsuler les responsabilités on évite de modifier les propriétés réactives du store directement depuis les composants. La façon propre de modifier cette propriété est de déléguer sa gestion au service :

```
const { logout } = useUserService();
```

Le simple principe d'architecture que nous venons de décrire est appelé le *Store pattern*, et il fonctionne très bien, même pour les grosses applications !



Essaie nos exercices [State management](#) 🦄 et [Remember me](#) 🦄 ! Tu apprendras comment utiliser l'API Composition pour gérer l'utilisateur courant dans Ponyracer.

24.2. Les librairies *flux-like*

Le pattern *store* est notre façon préférée de gérer l'état dans une application. Mais ce n'est pas la seule.

Certains développeurs lui préfèrent le pattern *flux*, qui est fortement inspiré du [framework Elm](#), et qui a été popularisé par l'écosystème React. La librairie la plus connue dans la communauté React est [Redux](#). Tu en as sans doute déjà entendu parler.

Le pattern flux et les bibliothèques comme Redux encouragent les développeurs à ne pas mettre à jour un *store* directement, mais plutôt à le faire en *dispatchant* des actions. Ces actions sont ensuite prises en charge par des *reducers*, qui ont la responsabilité de modifier le *store*. Ça te paraît confus ? Ça l'est au premier regard, et ces couches et indirections peuvent en effet mener à des applications sur-architecturées.

En fait, l'auteur de Redux lui-même, Dan Abramov, a écrit un article de blog devenu célèbre [You might not need Redux](#), parce qu'il s'est rendu compte que tout le monde se mettait à utiliser Redux dans les applications React, quelle que soit la pertinence et le coût de ce choix. L'écosystème Angular souffre parfois du même syndrome : [NgRx](#) est assez populaire, mais est bien souvent inutile, parce qu'il est parfaitement possible d'utiliser un pattern similaire au *store*.

Tu vois sans doute où je veux en venir : je ne suis pas sûr que tu aies besoin d'autre chose que de Vue pour gérer l'état de ton application. C'est sans doute plus sage de démarrer sans utiliser de librairie tierce et de voir comment ça se passe. Normalement, ça devrait rouler.

Mais je peux aussi comprendre que tu sois curieux et que tu aies envie d'en savoir un peu plus sur les bibliothèques de gestion d'état de l'écosystème Vue. Bon, allons-y !

24.3. Vuex



La bibliothèque de gestion d'état la plus populaire en Vue 2 est [Vuex](#). Vuex v4 fonctionne avec Vue 3, mais n'est pas idéale, car elle n'est pas type-safe, et n'est pas super bien intégrée avec l'API de Composition. En Vue 3, la recommandation est maintenant d'utiliser Pinia, qui est pratiquement Vuex v5, mais avec un nom différent et un logo mignon. Sens-toi libre de sauter cette section et de lire directement le passage consacré à Pinia ci-dessous !

Avec Vuex, on commence par définir un *store* avec `createStore`. En général, on utilise un seul *store* pour toute l'application.

Le *store* a un état initial, auquel on peut apporter des mutations successives. Les mutations sont des modifications, synchrones, de l'état du *store*.

store.ts

```
export interface State {
  userModel: UserModel | null;
}

const store = createStore<State>({
  state: () =>
    reactive({
      userModel: null
    }),
  mutations: {
    logout: state => (state.userModel = null)
  }
);
export default store;
```

Le *store* est un plugin Vue, qui doit être installé dans l'application (comme nous l'avons fait pour le routeur). Ensuite, on peut accéder au *store* avec `useStore()`. Mais si on procède ainsi, TypeScript ne connaît pas le type du *store*. Pour avoir un *store* typé correctement, la manière recommandée de procéder est d'utiliser une clé d'injection (nous venons d'en parler d'en le chapitre précédent si tu te rappelles bien) :

store.ts

```
export const storeKey: InjectionKey<Store<State>> = Symbol();
```

On peut maintenant l'enregistrer :

main.ts

```
createApp(App)
  .use(store, storeKey)
  .use(router)
  .mount('#app');
```

Dans un composant, on accède au *store* avec `useStore(storeKey)`, et il sera ainsi correctement typé !

Pour éviter de répéter ce code dans tous les composants, on peut aussi définir une fonction qui retourne le *store* :

store.ts

```
export function useAppStore() {
  return useStore(storeKey);
}
```

Et utiliser ensuite cette fonction pour accéder au *store* et à son état :

Home.vue

```
const store = useAppStore();
const { userModel } = toRefs(store.state);
```

Quand on doit modifier l'état du *store*, on utilise une *mutation*, pour déclencher une modification synchrone, avec `store.commit()` :

Navbar.vue

```
const store = useAppStore();
function logout() {
  store.commit('logout');
}
```

Le nom de la mutation n'est malheureusement pas *type-safe* dans Vuex v4. Certaines équipes pallient ce problème en déclarant les noms des mutations dans une `enum`, et en utilisant cette `enum` partout.

Les modifications asynchrones sont quant à elles gérées via des *actions* dans le *store* :

store.ts

```
export interface State {
  userModel: UserModel | null;
}

const store = createStore<State>({
  state: () =>
    reactive({
      userModel: null
    }),
  actions: {
    login: async ({ commit }, credentials) => {
      // call the backend with the credentials
      // ...
      commit('login', response.data);
    }
  },
  mutations: {
    login: (state, user) => (state.userModel = user),
  }
},
```

```
    logout: state => (state.userModel = null)
  }
});

export default store;
```

En règle générale, une action fait un appel asynchrone et finit par exécuter une mutation (avec `commit()`) pour mettre à jour l'état du *store*.

Au fur et à mesure que l'application s'étoffe, l'état devient de plus en plus volumineux. Vuex a heureusement une notion de "module", qui permet de découper le *store* en plus petites parties, qui ont chacune leur état, leurs mutations et leurs actions.

Même avec les modules, l'un des inconvénients évidents de Vuex est le *boilerplate* et les indirections nécessaires pour faire quoi que ce soit. Un autre inconvénient est l'intégration perfectible avec TypeScript et l'API de composition.

L'avantage que son utilisation peut apporter est la structuration du code et la centralisation de l'état, ainsi que les aides au débogage permises par la bibliothèque. [Vue devtools](#) peut par exemple montrer l'état courant du *store* et permet même de "voyager dans le temps" en montrant ou rétablissant les états antérieurs. Vuex a aussi un support pour des plugins. Un plugin [createLogger](#) est par exemple disponible, qui permet de tracer dans la console toutes les modifications d'état. Ça peut se révéler pratique pendant le développement. Bien sûr, tu peux aussi construire tes propres plugins.

24.4. Pinia

L'auteur du routeur de Vue, [@posva](#), a créé une alternative à Vuex, avec une élégante API de composition. La bibliothèque s'appelle [Pinia](#), et elle a probablement le logo le plus mignon de l'écosystème.



Le projet a démarré comme une expérience pour Vuex v5, mais s'est avéré tellement bien (et avec un nom et logo memorable) qu'il est devenu la recommandation officielle en matière de gestion d'état pour Vue.

Comme tu peux l'imaginer, la philosophie est très similaire à Vuex. Tu définis un *store* (ou plusieurs) avec `defineStore`. Un store a un état et des actions. Il peut également avoir des getters. Mais il n'y a pas de mutations comme dans Vuex : il y a simplement des actions. C'est une chose que j'apprécie avec Pinia : les mutations tendent à demander du code verbeux et pas toujours nécessaire. L'autre grand point fort est que Pinia est complètement type-safe, et a une super API de composition. En effet, tu peux écrire ton store avec une fonction, pratiquement comme tu le fais pour tes composants (l'alternative étant d'utiliser un objet avec des propriétés `state`, `getters` et `actions`, mais je préfère la version fonctionnelle).

Si l'on prend le même exemple, voici à quoi ressemble le store :

store.ts

```
export const useAppStore = defineStore('user', () => {
  const userModel = ref<UserModel | null>(null);

  async function login(credentials: { name: string; password: string }) {
    // call the backend with the credentials
    // ...
    userModel.value = response.data;
  }

  function logout() {
    userModel.value = null;
  }

  return { userModel, login, logout };
});
```

On peut ensuite dire à notre application d'utiliser Pinia :

main.ts

```
createApp(App)
  .use(createPinia())
  .use(router)
  .mount('#app');
```

Et nous pouvons alors utiliser le store dans nos composants (note que pour garder la réactivité, tu dois utiliser `storeToRefs()` quand tu déstructures le store) :

Home.vue

```
import { storeToRefs } from 'pinia';

const store = useAppStore();
const { userModel } = storeToRefs(store);
```

`storeToRefs()` est très similaire à `toRefs()` (que tu peux d'ailleurs aussi utiliser), mais ne renvoie

que les propriétés réactives (et pas les méthodes du store).

Quand on veut modifier l'état, on n'a pas besoin d'une mutation, on peut directement modifier l'état (sans avoir à s'embêter avec `.value` car le store lui-même est enveloppé avec `reactive`) :

Navbar.vue

```
const store = useAppStore();
function logoutDirectModification() {
  store.usermodel = null;
}
```

ou on peut appeler une fonction exposée par le store :

Navbar.vue

```
const store = useAppStore();
function logout() {
  store.logout();
}
```

Même s'il n'y a pas de mutation, Pinia capture quand même la modification, et l'affiche dans les Devtools. Pinia vient avec un plugin automatiquement installé pour les Devtools, qui permet de voir quel est l'état, d'inspecter sur une frise comment il a été modifié, et bien sûr de modifier l'état manuellement. On peut également s'abonner aux changements d'état avec `$subscribe()`. Pinia annule automatiquement l'abonnement dès que le composant est détruit.

24.5. Tester Pinia

Pinia vient avec un tout petit package appelé `@pinia/testing` qui nous simplifie la vie en matière de tests. Ce package nous donne une fonction `createPiniaTesting()` qui fait le gros du travail. Par défaut, elle *stub* toutes les actions, ce qui rend les composants qui les utilisent très simples à tester :

Navbar.spec.ts

```
import { createTestingPinia } from '@pinia/testing';

describe('NavbarWithPinia.vue', () => {
  test('should logout the user', async () => {
    // mount the component
    const wrapper = mount(NavbarWithPinia, {
      global: {
        // with a "fake" pinia
        plugins: [
          createTestingPinia({
            createSpy: vi.fn
          })
        ]
      }
    })
  })
})
```

```

    });
    // you can get the store, and change its state
    const store = useAppStore();
    const logout = wrapper.get('#logout');
    await logout.trigger('click');
    // actions are already spied,
    // so you just have to check if they are properly called
    expect(store.logout).toHaveBeenCalled();
  });
});

```

Dans le cas où tu ne veux *pas* remplacer les actions par des fausses, tu peux utiliser l'option `stubActions: false`:

App.spec.ts

```

describe('AppWithPinia', () => {
  test('should display the user', async () => {
    const wrapper = mount(AppWithPinia, {
      global: {
        plugins: [
          createTestingPinia({
            createSpy: vi.fn,
            stubActions: false
          })
        ]
      }
    });
    const store = useAppStore();
    store.userModel = { id: 1, name: 'Cedric' };
    await nextTick();

    // the user is logged in
    expect(wrapper.text()).toContain('Hello Cedric');

    await wrapper.get('#logout').trigger('click');
    // the action really logged out the user
    expect(wrapper.text()).toContain('Welcome, anonymous comrade!');
  });
});

```

24.6. Pourquoi utiliser un store ?

Comme expliqué ci-dessus, l'API de Composition est généralement suffisante.

Un store comme Pinia peut cependant être nécessaire dans un scénario : le *server-side rendering* (SSR). Si l'application est rendue sur le serveur, (par exemple si tu utilises [Nuxt](#)), alors tu devras faire attention à la manière dont tu gères l'état global afin de ne pas faire fuiter des informations d'un utilisateur à un autre par inadvertance. Comme les stores conçus manuellement sont

généralement des singltons, cela peut entraîner un problème bien connu appelé "*cross-request state pollution*" (pollution de l'état inter-requête). Pinia peut t'aider à gérer ça : il y a même [une intégration avec Nuxt](#).

En dehors de ce cas-là, un store comme Pinia peut être intéressant pour :

- le debugging, en utilisant les Devtools
- les tests, avec le package [@pinia/testing](#)
- le support du *Hot Module Replacement*, permettant de modifier les valeurs dans les stores quand on développe, et de voir le résultat sans recharger la page.
- les [plugins existants](#) (ou construire les tiens)

Pour résumer ce chapitre, plusieurs options existent pour gérer l'état de l'application :

- utiliser simplement l'API de composition et le pattern *store* : c'est suffisant dans la majorité des cas;
- ou utiliser Pinia (on évitera Vuex pour les applications Vue 3).



Essaie notre exercice [State management avec Pinia](#) 🦄 où tu refactoreras notre gestion de l'état en utilisant Pinia.

Chapter 25. Animations et effets de transition

Les animations donnent toujours une touche sympathique à une application. Ce n'est pas la première chose que l'on fait, bien sûr, mais cela peut réellement améliorer l'expérience utilisateur.

Vue répond présent pour nous aider à ajouter des animations et des effets de transitions à notre application. Voyons ça !

25.1. Animations en pur CSS

Avant de se plonger dans les effets de transitions, commençons par les "simples" animations CSS. On se retrouve souvent dans une situation où l'on veut attirer l'attention de notre utilisateur sur une partie de l'écran. Les animations sont une très bonne façon de faire cela !

Nous allons notifier notre utilisateur/utilisatrice que le formulaire qu'il/elle remplit est prêt à être envoyé, par exemple avec un effet de secousse (*shake*) sur le bouton de soumission.

Le procédé pour ajouter une animation est assez simple. On s'appuie sur la propriété CSS `animation`, qui déclenche généralement une animation que l'on peut définir avec `@keyframes`.

Une animation de secousse pourrait ressembler à :

```
.form-valid {
    animation: shake 300ms ease;
}

@keyframes shake {
    0%,
    10%,
    20%,
    30%,
    40% {
        transform: translateX(0.5rem);
    }
    50%,
    60% {
        transform: translateX(-0.5rem);
    }
}
```

`@keyframes` peut paraître un peu compliqué au premier abord. Ici, on batit une animation `shake`, en définissant différentes étapes :

1. Au début de l'animation (0%), l'élément est à sa position initiale.
2. Ensuite (10%), on veut déplacer l'élément légèrement sur la droite en utilisant `translateX`.
3. Ensuite (30%), on veut le déplacer un peu sur la gauche. On notera qu'il n'est pas nécessaire de spécifier où est l'élément à 20%, le navigateur le comprendra tout seul.

4. Ensuite (50%), on le déplace à droite.
5. Ensuite (70%), on le déplace à gauche.
6. Ensuite (90%), on le déplace à droite.
7. Enfin (100%), l'élément retourne dans sa position initiale.

Cette animation `shake` sera appliquée sur un élément qui a la classe CSS `form-valid`, pendant 300ms, avec un effet de *easing* (lentement, puis rapidement, puis lentement). Tu peux bien sûr choisir d'autres effets (`linear`, `ease-in`, `ease-out`, etc.).

Nous allons maintenant ajouter la class CSS `form-valid` au bouton de soumission si le formulaire est valide, et l'enlever s'il ne l'est pas :

```
<button :class="{ 'form-valid': valid }" :disabled="!valid">Save</button>
```

Quand le formulaire devient valide, le navigateur secoue le bouton pendant 300ms.

Voyons maintenant comment faire des effets de transitions.

25.2. Transitions d'entrée/sortie

Les transitions CSS sont une autre manière d'animer des éléments. Elles sont même plus simples que les animations. Avec les transitions CSS, on peut définir qu'un changement dans une ou plusieurs propriétés CSS d'un élément doit se faire de façon progressive.

Par exemple, si l'on définit les règles CSS suivantes :

```
.pill {
  transition: transform 300ms ease-out;
}

.pill.pill-selected {
  transform: scale(1.1);
}
```

On indique que pour les éléments avec la classe `pill`, chaque changement dans la valeur de la propriété `transform` doit être fait de façon progressive, pendant 300ms.

Si on ajoute alors dynamiquement la classe `pill-selected` à un élément avec la classe `.pill`, celui-ci va devenir légèrement plus gros en 300ms. Si l'on enlève la classe `pill-selected`, il reviendra alors à sa taille normale en 300ms.

Cependant, le problème est que l'on veut souvent appliquer un effet de transition quand un élément apparaît dans le DOM, ou lorsqu'il disparaît. Mais les directives `v-if` et `v-for` ne changent pas le style d'un élément : elles ajoutent ou suppriment simplement les éléments du DOM. Nous avons donc besoin de plus pour pouvoir, par exemple, avoir un effet de fondu quand un élément apparaît et disparaît.

Vue nous permet d'animer `v-if` avec le composant `<Transition>`. `Transition` attend un seul élément enfant (avec le `v-if`), et applique automatiquement des classes CSS dessus quand l'élément entre ou sort.

Les classes CSS suivantes sont appliquées quand l'élément entre dans le DOM :

- `v-enter-from` juste avant que l'élément apparaisse, au tout début de la transition.
- `v-enter-active` pendant qu'il apparaît. C'est celle que l'on utilise pour définir l'animation de la transition.
- `v-enter-to` quand la transition est terminée, et que l'élément apparaît (elle remplace alors `v-enter-from`).

Ensuite, quand l'élément est supprimé, les classes CSS suivantes sont appliquées :

- `v-leave-from` quand la transition commence
- `v-leave-active` pendant la transition
- and `v-leave-to` quand la transition est terminée.

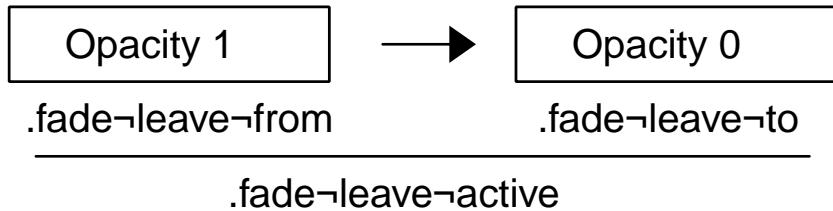
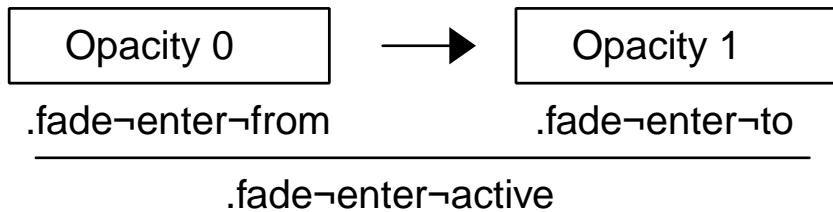
Il est aussi possible de nommer la transition (`fade` dans l'exemple suivant), et les classes sont préfixées avec le nom donnée au lieu de `v-`.

```
<Transition name="fade">
  <div v-if="display">Content</div>
</Transition>
```

Les classes vont donc être ici `fade-enter-from`, `fade-enter-active`, etc. Définissons donc un effet de fondu en CSS en utilisant ces classes :

```
.fade-enter-active,
.fade-leave-active {
  transition: opacity 1s;
}
.fade-enter-from,
.fade-leave-to {
  opacity: 0;
}
```

La transition va durer une seconde, et va progressivement changer l'opacité de 0 à 1 quand l'élément sera inséré dans le DOM, et de 1 à 0 quand il va être supprimé.



À chaque fois que la condition du `v-if` changera, l'élément va avoir un effet de fondu pendant une seconde, au lieu d'apparaître ou disparaître brutalement.

La seule limite est ton imagination (et tes compétences CSS) !

Si tu fais un essai, tu verras que l'animation ne se déclenche pas quand l'élément est affiché pour la première fois, mais seulement quand il disparaît/ré-apparaît. Tu peux alors utiliser la prop `appear` pour avoir l'aninamtion la première fois que l'élément apparaît :

```
<Transition name="fade" appear>
  <div v-if="display">Content</div>
</Transition>
```

`Transition` fonctionne aussi avec `v-else`:

```
<Transition name="fade" :mode="mode">
  <div v-if="loggedIn">
    <button @click="logout()">Log out</button>
  </div>
  <div v-else>
    <button @click="login()">Log in</button>
  </div>
</Transition>
```

Nous avons ici un bouton de connexion et un bouton de déconnexion qui s'affichent selon la valeur de `loggedIn`, une propriété réactive qui est vraie si l'utilisatrice est connectée. L'animation `fade` est appliquée aux deux boutons à chaque fois que `loggedIn` change. Ce qui est... bizarre, car on va alors voir les deux boutons en même temps, l'un disparaissant progressivement, pendant que l'autre

apparaît doucement.

C'est ici que la prop `mode` de `Transition` peut aider. On peut lui donner trois valeurs possibles :

- `default`, qui est évidemment la valeur par défaut, et a le comportement évoqué.
- `in-out`, qui déclenche d'abord l'animation de l'élément entrant, puis celle de l'élément sortant. Dans notre cas, cela veut dire que le nouveau bouton à afficher apparaît, puis l'autre bouton disparaît. Toujours pas la solution idéale ici.
- `out-in`, qui déclenche l'animation de l'élément à supprimer, puis celle de l'élément à afficher.

`out-in` est donc ce qu'il nous faut ici : le bouton actuel va progressivement disparaître, puis quand la transition sera terminée, le nouveau bouton apparaîtra progressivement ! C'est le mode que vous voulez dans la plupart des cas quand vous travaillez avec `v-else`.

25.3. Transitions dans les listes

Il est aussi possible d'animer `v-for` ! Cette fois, nous allons utiliser `TransitionGroup` pour envelopper notre `v-for`. Tu auras besoin d'ajouter une clé avec `:key` sur l'élément portant le `v-for` pour que cela fonctionne (c'est de toute façon une bonne pratique). À la différence de `Transition`, `TransitionGroup` affiche un `span`, mais on peut modifier ce comportement si nécessaire avec la prop `tag` :

```
<TransitionGroup name="list" tag="ul" appear>
  <li v-for="pony of ponies" :key="pony.id">{{ pony.name }}</li>
</TransitionGroup>
```

C'est ensuite très similaire à `Transition`, puisqu'il applique les mêmes classes. On peut définir une animation `list`, où les éléments vont glisser pour entrer ou sortir :

```
.list-leave-active,
.list-enter-active {
  transition: 1s;
}

.list-enter-from {
  transform: translateX(100%);
}

.list-leave-to {
  transform: translateX(-100%);
}
```

Quand un poney est ajouté à la collection, le nouvel élément `li` va glisser depuis la droite de l'écran dans la liste !

`TransitionGroup` ajoute aussi une classe CSS `v-move`, quand l'élément change de position :

```
.list-move {  
  transition: transform 1s ease;  
}
```

Si la liste est mélangée par exemple, tu verras les éléments glisser vers leurs nouvelles positions !

25.4. Et plus encore !

Il est aussi possible d'écrire des animations en pur JavaScript si tu as un cas d'utilisation plus avancé, en utilisant par exemple [GSAP](#). Vue peut aider là aussi, car [Transition](#) émet aussi des événements :

- `@beforeEnter/@enter/@afterEnter`
- `@beforeLeave/@leave/@afterLeave`
- `@enterCancelled/@leaveCancelled`

[Transition](#) fonctionne aussi bien avec le routeur, si l'on veut animer une transition entre deux pages.

Tu pourrais penser que tester un composant avec des animations va rendre les choses compliquées, comme il va falloir attendre la fin des animations pour tester le rendu. Mais en fait non, pas d'inquiétude ! Vue Test Utils remplace les composants [Transition](#) et [TransitionGroup](#), et tout fonctionne donc comme s'il n'y avait pas de transition 🤓.



Essaie notre exercice [Animations and transition effects](#) 🤓 ! Tu ajouteras des petites animations et transitions sympas à l'application.

Chapter 26. Patterns pour composants avancés

Parfois un simple composant ne suffit pas pour la tâche donnée. Voyons que faire dans ces cas-là.

26.1. Références de template avec `ref`

Avec les frameworks JavaScript modernes, nous ne manipulons pas la structure du DOM nous-même : c'est le travail du framework.

Mais parfois, on a besoin de récupérer une référence sur un élément DOM. Imaginons que l'on veuille donner le focus sur un input par exemple. Pour cela, on veut appeler la méthode native `focus()` sur l'input. Il nous faut donc récupérer une référence sur celui-ci.

Dans ce cas, on peut utiliser `ref()`. Oui, c'est bien la même fonction que celle que l'on utilise dans le système de réactivité, mais pour un usage un peu différent.

Commençons par créer la référence dans la section script :

Register.vue

```
const nameInput = ref<HTMLInputElement | null>(null);
```

Puis utilisons le nom de cette variable (`nameInput`) comme valeur de l'attribut `ref` de l'input, dans le template HTML :

Register.vue

```
<input id="name" ref="nameInput" v-model="name" />
```

Le type de `nameInput` est `Ref<HTMLInputElement | null>`, car l'input ne peut être utilisé que lorsque Vue a généré le DOM du composant et peuplé la référence.

Quand pouvons-nous alors utiliser `nameInput` ? Dans la fonction du cycle de vie `onMounted()` !

```
onMounted(() => {
  nameInput.value?.focus();
});
```

Ce pattern est très utile dans certains cas. Par exemple, quand on veut afficher un graphique ou une carte, on utilise souvent des bibliothèques tierces, qui ne connaissent pas Vue. Dans ce cas, on récupère simplement la référence DOM de l'élément et on la passe à la bibliothèque tierce.



C'est exactement ce que nous allons voir dans le Pack Pro 🧑. Essaye l'exercice [Charts in your app 🐾](#) qui apprend à utiliser les références de template pour intégrer une bibliothèque tierce.

26.2. Références de composant

Ce pattern fonctionne aussi avec les composants ! Imaginons que tu as un composant de menu déroulant, une *dropdown* (à toi, ou venant d'une bibliothèque, peu importe).

Ce composant de menu déroulant a une fonction pour ouvrir/fermer le menu.

Dropdown.vue

```
const opened = ref(true);
function toggle() {
  opened.value = !opened.value;
}

defineExpose({
  toggle
});
```

Comme le composant est défini avec la syntaxe `script setup`, la fonction doit être exposée avec `defineExpose` pour être visible en dehors du composant.

Le composant de menu déroulant peut ensuite être utilisé comme ceci :

Navbar.vue

```
<Dropdown ref="dropdown">
  <button>First choice</button>
  <button>Second choice</button>
</Dropdown>

<button @click="toggleDropdown()">Open the dropdown</button>
```

Comme tu peux le voir ci-dessous, le composant `Navbar` a une `ref` appelé `dropdown`. Cette `ref` est déclarée dans la section `script`, et peut être utilisée pour accéder à l'instance du composant, et à ses méthodes.

Navbar.vue

```
const dropdown = ref<typeof Dropdown | null>(null);
function toggleDropdown() {
  dropdown.value?.toggle();
}
```

Dans ce cas, la référence `dropdown` ne référence pas un élément HTML, elle référence un composant Vue.

C'est un pattern très utile dans certains cas.

Chapter 27. Directives personnalisées

Vue offre un tas de directives que tu peux utiliser dans vos templates (`v-if`, `v-for`, `v-model`, etc.), mais tu peux aussi créer les vôtres !

Cela peut être très pratique pour encapsuler un comportement que tu veux réutiliser plusieurs fois, mais qui n'est pas vraiment un composant (parce qu'il n'a pas de template) ni un composable (parce qu'on veut quand même manipuler le DOM).

Par exemple, dans le chapitre précédent, nous avons vu comment mettre le focus sur un élément dans le template grâce aux références de template. Ce code fonctionne parfaitement, mais nous devrions le dupliquer à chaque fois que nous voulons mettre le focus sur un élément.

Comment faire mieux ? Avec une directive !

Pour définir une directive, tu crées un objet qui peut avoir plusieurs propriétés, correspondant aux fonctions de cycle de vie de la directive que tu veux utiliser.

27.1. Cycle de vie d'une directive

Une directive a un cycle de vie similaire à celui d'un composant. Mais ici, au lieu d'utiliser des fonctions comme `onMount`, `onUnmount`, etc. dans un `setup`, nous allons utiliser des propriétés comme `mounted`, `unmounted`, etc. Ces fonctions sont appelées avec l'élément sur lequel ils sont appliqués en premier paramètre.

Donc notre directive de focus est aussi simple que cela :

Focus.ts

```
import { Directive } from 'vue';

export const vFocus: Directive<HTMLInputElement> = {
  // el is of type 'HTMLInputElement'
  // as vFocus is declared as 'Directive<HTMLInputElement>'
  mounted(el) {
    el.focus();
  }
};
```

On peut l'utiliser comme ceci :

Login.vue

```
<input v-focus name="login" />
```

Comme pour les composants, tu dois déclarer la directive dans la partie script. Si tu utilises la syntaxe `script setup`, tu dois simplement l'importer. C'est pourquoi le nom est important : c'est comme ça que Vue comprend ce que `v-focus` veut dire dans le template.

```
import { vFocus } from '@/directives/Focus';
```

Si tu n'utilises pas `script setup`, alors tu dois déclarer la directive dans l'option `directives` du composant.

27.2. Valeur des directives

Tu as peut-être remarqué que certaines directives de Vue peuvent avoir des valeurs, par exemple `v-model="user.name"`. D'autres peuvent avoir un argument comme `v-bind:src` ou `v-on:click`. Et enfin, certaines peuvent avoir des modificateurs, comme `v-model.number`.

Nos directives personnalisées peuvent avoir aussi une valeur, un argument et des modificateurs !

On peut imaginer une directive `v-focus` vraiment géniale qui aurait tout cela ! Par exemple, elle peut avoir une valeur pour indiquer si l'élément doit être mis en focus ou non. On peut l'utiliser comme ceci :

```
<input v-focus="shouldFocus" name="login" @blur="shouldFocus = false" />
<button @click="shouldFocus = true">Focus</button>
```

Chaque fonction de cycle de vie peut recevoir un second paramètre, généralement appelé `binding`. `binding` est un objet qui contient la valeur, les paramètres et les modificateurs. La directive serait définie comme ceci :

```
import { Directive } from 'vue';

export const vFocus: Directive<HTMLInputElement, boolean> = {
    // called when the bound element is mounted to the DOM
    mounted(el, binding) {
        // as vFocus is declared as Directive<HTMLInputElement, boolean>
        // 'binding' is inferred as 'DirectiveBinding<boolean>',
        // so 'binding.value' is of type boolean
        if (binding.value) {
            el.focus();
        }
    },
    // called after the containing component has re-rendered
    updated(el, binding) {
        if (binding.value) {
            el.focus();
        }
    }
}
```

```
};
```

La directive utilise deux fonctions de cycle de vie : `mounted` et `updated`. La première est appelée quand l'élément est monté dans le DOM. La deuxième est appelée quand le composant qui utilise la directive est mis à jour.

Comme c'est un cas d'utilisation assez courant, Vue permet de définir la directive comme une fonction, qui sera appelée au "mount" et à la mise à jour :

Focus.ts

```
import { Directive } from 'vue';

// called when the bound element is mounted to the DOM
// and when the containing component has re-rendered
export const vFocus: Directive<HTMLInputElement, boolean> = (el, binding) => {
    // as vFocus is declared as Directive<HTMLInputElement, boolean>
    // 'binding' is inferred as 'DirectiveBinding<boolean>',
    // so 'binding.value' is of type boolean
    if (binding.value) {
        el.focus();
    }
};
```

C'est la forme que nous utiliserons dans le reste du chapitre.

27.3. Argument des directives

Une directive peut aussi avoir un argument.

Par exemple, la directive `v-bind` peut avoir un argument pour indiquer l'attribut à lier avec `v-bind:src`. La directive `v-on` peut avoir un argument pour indiquer l'événement à écouter avec `v-on:click`. Nos directives personnalisées peuvent aussi avoir des arguments !

On peut imaginer une directive `v-focus` qui aurait un argument pour indiquer un délai avant de mettre le focus sur l'élément. Par exemple, elle serait utilisée comme ceci :

Login.vue

```
<input v-focus:300="shouldFocus" name="login" @blur="shouldFocus = false" />
```

L'argument est disponible dans l'objet `binding`, dans la propriété `arg`. C'est une chaîne de caractères, il nous faut donc le convertir en nombre dans notre exemple. La directive peut être définie comme ceci :

Focus.ts

```
import { Directive } from 'vue';
```

```
// called when the bound element is mounted to the DOM
// and when the containing component has updated
export const vFocus: Directive<HTMLInputElement, boolean> = (el, binding) => {
  const delay = parseInt(binding.arg ?? '0');
  if (binding.value) {
    setTimeout(() => el.focus(), delay);
  }
};
```

27.4. Modificateurs des directives

Enfin, une directive peut avoir des modificateurs, comme `v-model.number` qui indique que la valeur doit être convertie en nombre.

On peut imaginer une directive `v-focus` qui aurait un modificateur `seconds` pour indiquer que le délai est en secondes (et pas en millisecondes), par exemple. On peut alors l'utiliser comme ceci :

Login.vue

```
<input v-focus.seconds:2="shouldFocus" name="login" @blur="shouldFocus = false" />
```

Les modificateurs sont disponibles dans l'objet `binding`, dans la propriété `modifiers`. Si un modificateur est présent, alors la valeur de la propriété est `true`.

La directive peut être définie comme ceci :

Focus.ts

```
import { Directive } from 'vue';

// called when the bound element is mounted to the DOM
// and when the containing component has updated
export const vFocus: Directive<HTMLInputElement, boolean> = (el, binding) => {
  // if the modifiers object contains a 'seconds' property, then we need to multiply
  // the delay by 1000
  const isDelayInSeconds = binding.modifiers['seconds'];
  const delay = parseInt(binding.arg ?? '0') * (isDelayInSeconds ? 1000 : 1);
  if (binding.value) {
    setTimeout(() => el.focus(), delay);
  }
};
```



Nous avons un cas d'utilisation intéressant dans le Pro Pack, où tu construis une directive personnalisée pour ajouter des classes sur les champs de formulaire en fonction de leur validation. Essaye notre exercice [Custom directive](#) pour apprendre à la construire.

Chapter 28. Internationalisation

So you want to internationalize your application, huh?

Bon, ne te tracasse pas si ton niveau d'anglais est si bas que tu n'as pas pu comprendre cette petite introduction. Ton rôle de développeur n'est pas de traduire ton application en anglais, espagnol ou quelque autre dialecte exotique. Ce que tu peux faire par contre, c'est rendre cette traduction possible. Ce chapitre explique comment y arriver.

Vue en lui-même ne fait pas grand-chose pour nous aider. Mais un plugin de l'écosystème est là pour nous sauver : [vue-i18n](#).

vue-i18n nous aide sur deux sujets :

- la traduction de textes
- le formatage de nombres et de dates

28.1. Mise en place de vue-i18n

vue-i18n est un plugin que tu peux instancier avec `createI18n()`. Le plugin permet de définir la locale de l'application et les traductions que tu veux utiliser.

Ces traductions sont généralement définies dans des fichiers spécifiques à chaque langue. vue-i18n supportent plusieurs formats pour ces fichiers, par exemple YAML ou JSON. Les messages peuvent également être déclarés ou surchargés dans un composant, soit en TypeScript, soit dans une section dédiée `<i18n></i18n>` d'un SFC.

Même si ce n'est pas le format idéal, JSON a un avantage de taille : il peut être compris par TypeScript et nous donne du typage strict.

`src/en.json`

```
{  
  "chart": {  
    "title": "Score history",  
    "legend": "Legend",  
  }  
}
```

C'est extrêmement pratique, car cela permet l'auto-complétion dans ton IDE quand tu travailles avec les clés des messages (`chart.t` est autocomplété en `chart.title`). TypeScript va aussi échouer le build s'il manque une ou plusieurs clés à l'un des fichiers de traduction !

`src/i18n.ts`

```
import { createI18n } from 'vue-i18n';  
// translation files  
import en from './locales/en.json';  
import fr from './locales/fr.json';
```

```
// we can leverage TypeScript to type-check the translations
export type Message = typeof en;
export default createI18n<[Message], 'en' | 'fr'>({
  legacy: false, // as we only want to use the "modern" composition API
  locale: 'en',
  messages: {
    en,
    fr
  }
});
```

Tu peux ensuite utiliser ce plugin dans ton application :

main.ts

```
createApp(App)
  .use(i18n)
  .mount('#app');
```

On est maintenant prêt à traduire nos textes !

28.2. Traduction de textes

vue-i18n suit le modèle que nous avons vu plusieurs fois : il offre un *composable*, appelé `useI18n()` que l'on peut utiliser dans nos composants. Ce *composable* nous permet de récupérer la fonction `t()` et de l'utiliser dans notre code et dans nos templates pour traduire du texte.

Par exemple, on peut utiliser la clé `chart.title` que nous avons défini pour afficher le titre :

ScoreHistory.vue

```
const { t } = useI18n<{ message: Message }, 'en' | 'fr'>();
const title = ref(t('chart.title'));
```

Tu verras plus souvent `t` utilisée directement dans les templates :

ScoreHistory.vue

```
<div id="legend">{{ t('chart.legend') }}</div>
```

La fonction va récupérer le texte associé à la clé depuis les messages correspondant à la locale courante et l'afficher.

28.3. Messages paramétrisés

On a parfois besoin d'avoir des paramètres dans les messages. Par exemple, on aimerait afficher le login d'un utilisateur dans un message :

src/en.json

```
{  
  "chart": {  
    "title": "Score history",  
    "legend": "Legend",  
    "user": "Score for user { login }",  
  }  
}
```

`t` accepte alors un objet comme second paramètre :

ScoreHistory.vue

```
<div id="score">{{ t('chart.user', { login: 'cedric' }) }}</div>
```

28.4. Pluralisation

Il est aussi possible de gérer la pluralisation avec `|` :

```
{  
  "chart": {  
    "title": "Score history",  
    "legend": "Legend",  
    "ponies": "no ponies | one pony | {n} ponies"  
  }  
}
```

Tu peux alors passer un nombre d'éléments à afficher à la fonction `t` :

```
<div id="ponies">{{ t('chart.ponies', count) }}</div>  
<!-- Displays 'no ponies' if count is 0, '3 ponies' if count is 3 --&gt;</pre>
```

28.5. Changer la locale

Quand tu supportes plusieurs langues dans ton application, tu veux probablement fournir à tes utilisateurs une façon de changer la locale (ou lire la préférence de tes utilisateurs au démarrage).

Tu peux récupérer la locale courante et celles disponibles grâce à `useI18n()` :

Navbar.vue

```
const { locale, availableLocales } = useI18n();
```

`locale` est une `ref`, tu peux mettre à jour sa valeur et Vue rafraîchira les templates en conséquence.

Tu peux par exemple ajouter un simple `select` à l'application pour laisser l'utilisateur choisir :

Navbar.vue

```
<select v-model="locale">
  <option v-for="availableLocale in availableLocales" :key="availableLocale"
  :value="availableLocale">
    {{ availableLocale }}
  </option>
</select>
```

Note que `t` rafraîchit automatiquement les traductions dans les templates (car la fonction est réévaluée). Mais pas dans le code : la fonction n'est exécutée qu'une seule fois pendant le setup. Si nécessaire, tu peux cependant surveiller la locale pour déclencher le rafraîchissement de la traduction :

ScoreHistory.vue

```
watchEffect(() => (title.value = t('chart.title')));
```

28.6. Formatage

vue-i18n aide également au formatage des dates avec `d(Date|string|number)`, et des nombres avec `n(number)`. Ces deux fonctions s'appuient sur le support `Intl` des navigateurs et permettent toutes deux de préciser un *pattern* en deuxième paramètre si besoin.

Pour récupérer ces fonctions, tu peux utiliser `useI18n` comme précédemment :

```
const { d, n, availableLocales, locale } = useI18n();
```

et les utiliser dans les templates :

```
<div id="default-date">{{ d('2020-09-18') }}</div>
<!-- 9/18/2020 in English, 18/09/2020 in French --&gt;
&lt;div id="default-number"&gt;{{ n(2010.983) }}&lt;/div&gt;
<!-- 2,010.983 in English, 2 010,983 in French --&gt;</pre>
```

28.7. Autres fonctionnalités (lazy-loading, support de Vite et plus)

vue-i18n vient aussi avec d'autres fonctionnalités sympathiques :

- des options de formatage avancées pour les nombres, dates et monnaies
- une directive custom `v-t` qui peut être utilisée à la place de la fonction `t()`. La directive custom offre quelques optimisations, car elle permet de pré-compiler les traductions. Mais elle est

moins flexible et nécessite un build custom.

- le chargement à la demande des traductions est possible
- Vite est supporté grâce à un plugin

Tu devrais te sentir prêt à internationaliser ton application maintenant !



Pour commencer avec l'internationalisation et apprendre quelques astuces de plus, essaye notre exercice [Internationalisation 🦄](#)! Il fait partie de notre Pack Pro, et tu apprendras comment internationaliser une application entière.

Chapter 29. Sous le capot

J'ai une confession à te faire : J'aime particulièrement apprendre comment les choses marchent *vraiment*. Il y a de nombreux frameworks JavaScript, mais peu de gens connaissent vraiment les différences entre eux, sous le capot.

Si toi aussi tu es curieux de savoir comment Vue fonctionne *vraiment*, suis moi ! ☺

29.1. Les changements de rendu

Tous les frameworks JavaScripts sont confrontés au même problème : lorsque l'état de l'application change, le framework doit appliquer ces changements au rendu de la page. Il ne peut pas simplement regénérer toute la page cependant : ce serait un gaspillage de ressources énorme parce que, la plupart du temps, seuls quelques nœuds du DOM doivent être mis à jour.

Il y a actuellement deux solutions classiques à ce problème. Les frameworks populaires que tu connais utilisent l'une ou l'autre.

La première solution est de compiler les templates HTML en code JavaScript qui crée et modifie les éléments du DOM quand l'état du composant change.

En pseudo-code, cela signifie que le template d'un composant `Greetings` tel que celui-ci :

`Greetings.vue`

```
<div>Hello {{ user.name }}</div>
<input type="checkbox" :value="user.isAdmin" @change="updateAdminRights()">
```

serait transformé par le framework en code tel que celui-ci pour afficher le composant :

```
const root = document.createElement('div');
const elements = [];
function createGreetings(component: Greetings) {
  const div = document.createElement('div');
  root.appendChild(div);
  elements.push(div);
  const input = document.createElement('input');
  input.type = 'checkbox';
  input.addEventListener('change', () => component.updateAdminRights());
  elements.push(input);
  root.appendChild(input);
}
```

Il faut aussi une fonction qui modifie le DOM lorsque l'état est modifié :

```
function updateGreetings(component: Greetings, previousState: Greetings) {
  if (component.user.name !== previousState.user.name) {
```

```

const div = elements[0];
previousState.user.name = component.user.name;
div.innerText = 'Hello ' + component.user.name;
}
if (component.user.isAdmin !== previousState.user.isAdmin) {
  const input = elements[1];
  previousState.user.isAdmin = component.user.isAdmin;
  input.value = component.user.isAdmin;
}
}

```

C'est en gros ce que tu devrais écrire toi-même si tu étais forcé d'écrire une application en utilisant *uniquement* du JavaScript, et pas de HTML.

Et c'est ce que des frameworks comme Angular ou Svelte font : ils génèrent du code JavaScript contenant des instructions de génération et de modification du DOM, et c'est cette version compilée des composants qui est livrée aux utilisateurs.

Vue a une approche différente, popularisée par React : le DOM virtuel (*Virtual DOM*).

L'idée est assez similaire, mais avec une différence : le framework convertit également le template en code JavaScript, mais ce code JavaScript agit comme s'il devait regénérer toute la page à chaque fois. En réalité, il ne le fait pas bien sûr : les changements génèrent un DOM virtuel, une représentation en mémoire du DOM. Le framework compare ensuite ce nouvel arbre DOM virtuel avec la version précédente, et applique uniquement les différences au DOM réel.

Commençons par voir comment Vue compile nos templates.

29.2. La compilation des templates

Quand Vue compile le template d'un composant, il génère une fonction `render` pour le composant. Cette fonction `render` est appelée quand Vue doit regénérer le DOM après un changement d'état.

Donc, un composant `Greetings` comme celui-ci :

`Greetings.vue`

```

<div>
  <span>Hello {{ user.name }}</span>
  <small v-if="user.isAdmin">admin</small>
</div>
<input type="checkbox" :value="user.isAdmin" @change="updateAdminRights()" />

```

résulterait en une fonction `render` ajoutée au composant.

Le compilateur Vue est constitué de 3 parties pour arriver au résultat :

- un *parser*
- plusieurs *transformers*

- un générateur de code qui produit la fonction `render`.

La responsabilité du *parser* est de lire le template, caractère par caractère, et d'en déduire la structure du template.

En gros, ça fonctionne comme ça :

- `<` : Oh ! Le début d'un élément ! Analysons-le !
- `d` : OK le nom de l'élément commence par `d`
- ...
- `>` : OK l'élément est un `div` et n'a aucun attribut. Examinons maintenant ses enfants.
- ...
- `{` : Serait-ce le début d'une interpolation ?
- `{` : Oui ! C'est une interpolation ! Analysons-la !
- ...
- `v` : Voilà le début d'un attribut
- `-` : Oh c'est en fait une directive Vue !
- ...
- `:` : Ah, c'est un binding Vue
- etc.

(Oui, dans mon imaginaire, un *parser* est extrêmement joyeux et toujours surpris)

Si le *parser* rencontre quelque chose d'inattendu, il lève une exception. Lorsqu'il est arrivé au bout du template, il génère un *AST*, un *Abstract Syntax Tree*, représentant la structure du template :

Greetings AST

```
const greetingsAST = {
  type: 'ELEMENT',
  tag: 'div',
  props: [],
  children: [
    {
      type: 'ELEMENT',
      tag: 'span',
      props: [],
      children: [
        { type: 'TEXT', content: 'Hello ' },
        { type: 'INTERPOLATION', content: 'user.name' }
      ]
    },
    {
      type: 'ELEMENT',
      tag: 'small',
      props: [{ type: 'DIRECTIVE', name: 'if', exp: 'user.isAdmin' }]
    }
  ]
};
```

```
    children: [{ type: 'TEXT', content: 'admin' }]
  }
  // etc.
]
};
```

Le compilateur Vue applique ensuite des transformations sur cet *AST*. Les *transformers* modifie l'*AST* avant que la génération de code ne soit lancée, et le font davantage ressembler à la structure d'un programme, avec des appels de fonctions et des arguments. C'est aussi la phase pendant laquelle chaque directive est branchée : la transformation de `v-if` change l'*AST* afin qu'il contienne un `if`.

Lorsque les transformations ont été appliquées, le compilateur n'a plus qu'à générer le code. C'est la phase "*codegen*", qui transforme l'arbre en une fonction JavaScript `render` contenant les instructions.

Et cette fonction `render` est appelée à chaque fois que Vue veut regénérer le DOM virtuel pour notre composant.

29.3. Le DOM virtuel

Implémentons un DOM virtuel nous-mêmes pour avoir une idée de ce en quoi ça consiste.

Il nous faut une structure de données représentant le DOM à générer, très similaire à la structure de l'*AST* produite par le *parser*.

Dans notre version simplifiée, un nœud textuel sera un simple objet tel que `{ type: 'text', text: 'Hello' }`, et un `HTMLElement` sera représenté par un objet à peine plus complexe :

```
const input = {
  type: 'input',
  properties: { type: 'checkbox' },
  children: []
}
```

`children` est un tableau qui peut contenir d'autres éléments. Les deux types d'éléments seront appelés `VNode` dans les exemples qui suivent. Mais nous faisons une distinction entre ces deux types de nœuds parce que, pour générer un élément HTML dans le DOM, il faudra appeler `document.createElement` alors que pour générer un nœud textuel il faudra appeler `document.createTextNode`.

Commençons par écrire deux fonctions utilitaires pour créer ces `VNodes` :

```
function createVirtualElement(type: string, properties: { [key: string]: any },
  children: Array<VNode | null>): VNode {
  return { type, properties, children };
}
```

```

function createVirtualText(text: string): VNode {
  return { type: 'text', text };
}

```

La fonction `render`, de notre composant, générée par le compilateur Vue, utilise ces fonctions et ressemble à ceci :

```

/**
 * <div>
 *   <div>
 *     <span>Hello {{ user.name }}</span>
 *     <small v-if="user.isAdmin">admin</small>
 *   </div>
 *   <input type="checkbox" :value="user.isAdmin" @change="updateAdminRights()" />
 * </div>
 */

function render(component: Greetings): VNode {
  // div
  return createVirtualElement('div', {}, [
    // div
    createVirtualElement('div', {}, [
      // span
      createVirtualElement('span', {}, [
        // text: Hello {{ user.name }}
        createVirtualText('Hello ' + component.user.name)
      ]),
      // small only if v-if is true
      component.user.isAdmin
        ? createVirtualElement('small', {}, [
            // text: admin
            createVirtualText('admin')
          ])
        : null
    ]),
    // input
    createVirtualElement(
      'input',
      {
        type: 'checkbox',
        value: component.user.isAdmin,
        onChange: () => component.updateAdminRights()
      },
      []
    )
  ]);
}

```

Elle retourne donc une structure de données représentant le DOM tel qu'on le désire : le DOM virtuel.

À l'exécution, Vue examine cette structure et la compare avec la version précédente, gardée en mémoire, qui correspond au DOM réellement affiché. Cette comparaison permet de détecter les changements dans cette structure, qui doivent ensuite être appliqués au DOM réel.

Voici ce que notre implémentation simple réalise : elle reçoit l'élément racine de l'application (celui où l'application est "montée"), le `VNode` précédent (celui qui est donc réellement affiché), et le nouveau `VNode`, qui vient d'être généré par la fonction `render` ci-dessus.

```
function applyChanges(parent: ChildNode, index: number, previousNode: VNode | null,
newNode: VNode | null) {
}
```

Premier cas de figure : le nouveau `VNode` est null, indiquant que l'élément correspondant du DOM doit être supprimé. Nous pourrions l'enlever du DOM mais, afin de simplifier la suite, gardons le même nombre d'éléments, (et donc des indices cohérents) et remplaçons-le par un commentaire :

```
function applyChanges(parent: ChildNode, index: number, previousNode: VNode | null,
newNode: VNode | null) {
    // no new element
    const oldChild = parent.childNodes[index];
    if (!newNode) {
        // we replace the old node with a comment
        const comment = document.createComment('removed element');
        parent.replaceChild(comment, oldChild);
        return;
    }
}
```

Cas de figure inverse : le `VNode` précédent est null, indiquant qu'un nouvel élément doit être créé et ajouté au DOM.

```
function applyChanges(parent: ChildNode, index: number, previousNode: VNode | null,
newNode: VNode | null) {
    // no new element
    const oldChild = parent.childNodes[index];
    if (!newNode) {
        // we replace the old node with a comment
        const comment = document.createComment('removed element');
        parent.replaceChild(comment, oldChild);
        return;
    }
    // no element previously
    else if (!previousNode) {
        const child = createElement(newNode);
        // if there was a comment we replace it
        if (oldChild) {
            parent.replaceChild(child, oldChild);
        } else {
```

```

        // otherwise we create a new child
        parent.appendChild(child);
    }
}
}

```

avec `createChildElement`:

```

function createChildElement(vNode: VNode): Text | HTMLElement {
    // if it is a text VNode
    if (isTextNode(vNode)) {
        return document.createTextNode(vNode.text);
    }
    // otherwise, we create a DOM element fo the correct type
    const child = document.createElement(vNode.type);
    // and we set its properties
    for (const prop in vNode.properties) {
        (child as any)[prop] = vNode.properties[prop];
    }
    return child;
}

```

Autre cas de figure que nous devons gérer : le type de l'élément a changé. Dans ce cas, nous créons un nouvel élément et remplaçons l'ancien par le nouveau :

```

function applyChanges(parent: ChildNode, index: number, previousNode: VNode | null,
newNode: VNode | null) {
    // no new element
    const oldChild = parent.childNodes[index];
    if (!newNode) {
        // we replace the old node with a comment
        const comment = document.createComment('removed element');
        parent.replaceChild(comment, oldChild);
        return;
    }
    // no element previously
    else if (!previousNode) {
        const child = createChildElement(newNode);
        // if there was a comment we replace it
        if (oldChild) {
            parent.replaceChild(child, oldChild);
        } else {
            // otherwise we create a new child
            parent.appendChild(child);
        }
    }
    // type changed?
    else if (previousNode.type !== newNode.type) {
        const child = createChildElement(newNode);

```

```

    parent.replaceChild(child, oldChild);
}
}

```

Nous pouvons aussi rencontrer un nœud textuel dont la valeur a changé. Dans ce cas, on doit remplacer le `textContent` par la nouvelle valeur :

```

function applyChanges(parent: ChildNode, index: number, previousNode: VNode | null,
newNode: VNode | null) {
    // no new element
    const oldChild = parent.childNodes[index];
    if (!newNode) {
        // we replace the old node with a comment
        const comment = document.createComment('removed element');
        parent.replaceChild(comment, oldChild);
        return;
    }
    // no element previously
    else if (!previousNode) {
        const child = createElement(newNode);
        // if there was a comment we replace it
        if (oldChild) {
            parent.replaceChild(child, oldChild);
        } else {
            // otherwise we create a new child
            parent.appendChild(child);
        }
    }
    // type changed?
    else if (previousNode.type !== newNode.type) {
        const child = createElement(newNode);
        parent.replaceChild(child, oldChild);
    }
    // text changes
    else if (isTextNode(previousNode) && isTextNode(newNode)) {
        // the previous node exists
        if (oldChild) {
            // update the text only if necessary
            if (previousNode.textContent !== newNode.textContent) {
                oldChild.textContent = newNode.textContent;
            }
        } else {
            // create a new text node
            const newTextNode = createElement(newNode);
            parent.appendChild(newTextNode);
        }
        return;
    }
}

```

Il nous faut évidemment gérer les changements de propriétés :

```
function applyChanges(parent: ChildNode, index: number, previousNode: VNode | null, newNode: VNode | null) {
    // no new element
    const oldChild = parent.childNodes[index];
    if (!newNode) {
        // we replace the old node with a comment
        const comment = document.createComment('removed element');
        parent.replaceChild(comment, oldChild);
        return;
    }
    // no element previously
    else if (!previousNode) {
        const child = createElement(newNode);
        // if there was a comment we replace it
        if (oldChild) {
            parent.replaceChild(child, oldChild);
        } else {
            // otherwise we create a new child
            parent.appendChild(child);
        }
    }
    // type changed?
    else if (previousNode.type !== newNode.type) {
        const child = createElement(newNode);
        parent.replaceChild(child, oldChild);
    }
    // text changes
    else if (isTextNode(previousNode) && isTextNode(newNode)) {
        // the previous node exists
        if (oldChild) {
            // update the text only if necessary
            if (previousNode.textContent !== newNode.textContent) {
                oldChild.textContent = newNode.textContent;
            }
        } else {
            // create a new text node
            const newTextNode = createElement(newNode);
            parent.appendChild(newTextNode);
        }
        return;
    }
    // properties changed?
    else if (!isTextNode(previousNode) && !isTextNode(newNode)) {
        for (const prop in newNode.properties) {
            (oldChild as any)[prop] = newNode.properties[prop];
        }
    }
}
```

Dernier cas de figure de notre implémentation : que faire si les enfants ont changé ? Et bien c'est assez simple : on itère dessus et on appelle notre fonction `applyChanges` récursivement :

```
function applyChanges(parent: ChildNode, index: number, previousNode: VNode | null, newNode: VNode | null) {
    // no new element
    const oldChild = parent.childNodes[index];
    if (!newNode) {
        // we replace the old node with a comment
        const comment = document.createComment('removed element');
        parent.replaceChild(comment, oldChild);
        return;
    }
    // no element previously
    else if (!previousNode) {
        const child = createElement(newNode);
        // if there was a comment we replace it
        if (oldChild) {
            parent.replaceChild(child, oldChild);
        } else {
            // otherwise we create a new child
            parent.appendChild(child);
        }
    }
    // type changed?
    else if (previousNode.type !== newNode.type) {
        const child = createElement(newNode);
        parent.replaceChild(child, oldChild);
    }
    // text changes
    else if (isTextNode(previousNode) && isTextNode(newNode)) {
        // the previous node exists
        if (oldChild) {
            // update the text only if necessary
            if (previousNode.textContent !== newNode.textContent) {
                oldChild.textContent = newNode.textContent;
            }
        } else {
            // create a new text node
            const newTextNode = createElement(newNode);
            parent.appendChild(newTextNode);
        }
        return;
    }
    // properties changed?
    else if (!isTextNode(previousNode) && !isTextNode(newNode)) {
        for (const prop in newNode.properties) {
            (oldChild as any)[prop] = newNode.properties[prop];
        }
    }
}
```

```

// children changed?
if (!isTextNode(newNode)) {
  const maxLength = Math.max(
    newNode.children.length,
    previousNode && !isTextNode(previousNode) ? previousNode.children.length : 0
  );
  for (let i = 0; i < maxLength; i++) {
    applyChanges(
      parent.childNodes[index],
      i,
      previousNode && !isTextNode(previousNode) ? previousNode.children[i] : null,
      newNode.children[i]
    );
  }
}
}

```

Cette implémentation est loin d'être parfaite et optimale, mais en à peu près 50 lignes de code, elle permet de comprendre le principe du DOM virtuel !

Lorsque Vue détecte que l'état de l'application change, il appelle la fonction `render` de notre composant, puis appelle `applyChanges` avec le DOM virtuel précédent et le nouveau DOM virtuel. `applyChanges` modifie le DOM réel et Vue stocke ensuite le résultat de `render` pour la prochaine fois :

```

// first render
let previousRender = greetings.render();
// state update
greetings.user.isAdmin = true;

// Vue calls render again to get the new Virtual DOM
const newRender = greetings.render();
// and diff the changes to apply them on the real DOM
applyChanges(root, 0, previousRender, newRender);
// and store the new Virtual DOM as the reference
previousRender = newRender;

```

Pour récapituler, Vue a donc, au cœur de son fonctionnement, deux éléments essentiels :

- un compilateur de templates (pour générer la fonction `render`)
- une implémentation de DOM virtuel qui permet de détecter et d'appliquer les changements

L'implémentation du DOM virtuel était à l'origine un fork de `snabbdom` mais elle a depuis été complètement réécrite pour satisfaire au mieux les besoins de Vue.

29.3.1. Les fonctions `render` et `h`

Pour expliquer comment le DOM virtuel fonctionne, on a écrit une version simplifiée de la fonction `render`. Tu peux en fait jouer avec de compilateur Vue en ligne sur le [Vue Template Explorer](#) pour

voir à quoi ressemble une vraie fonction `render`.

La vraie fonction `render` est assez proche de la nôtre. Et Vue te permet d'ailleurs, si vraiment tu le veux, de fournir ta propre fonction `render`. Pour l'écrire, il te faudra utiliser une fonction utilitaire, nommée `h` (c'est le même nom que celui qu'avait choisi `snabbdom`), qui génère des nœuds du DOM virtuel.

Dans tes composants, au lieu d'écrire un template, tu peux directement ajouter une fonction `render` qui utilise `h`. Ou, si tu utilises l'API Composition, tu peux directement retourner cette fonction comme résultat de la fonction `setup`.

Notre composant `Greetings` peut donc être écrit comme ceci :

`Greetings.ts`

```
import { defineComponent, h, reactive } from 'vue';

export default defineComponent({
  name: 'Greetings',

  setup() {
    const user = reactive({
      name: 'Cyril',
      isAdmin: true
    });

    function updateAdminRights() {
      user.isAdmin = !user.isAdmin;
    }

    return () =>
      h(
        'div',
        /* children */ [
          h(
            'div',
            /* children */ [
              h('span', 'Hello ' + user.name),
              /* v-if becomes a simple condition */
              user.isAdmin ? h('small', 'admin') : null
            ]
          ),
          h(
            'input',
            /* props */ {
              type: 'checkbox',
              value: user.isAdmin,
              onChange: () => updateAdminRights()
            }
          )
        ]
      )
  }
})
```

```
    );
}
});
```

Comme tu peux le voir, c'est assez compréhensible, mais pas aussi agréable à écrire ni à lire qu'un template HTML. Néanmoins, c'est une alternative valide et puissante puisqu'elle permet d'écrire le code que tu veux.

Il y a encore une *autre* alternative : JSX !

29.4. JSX

Bientôt...

29.5. Réactivité

Nous avons vu comment Vue *applique* les changements d'état. Voyons maintenant comment Vue *déetecte* les changements d'état.

À nouveau, pour vraiment comprendre comment ça se passe, j'aimerais réécrire les différentes fonctions réactives que Vue propose : `reactive`, `ref`, `computed` et `watchEffect`.

Commençons par un exemple simple pour illustrer comment `watchEffect` fonctionne : nous avons un prix et une quantité, et nous voulons ajouter une ligne dans les logs chaque fois que l'un ou l'autre change.

```
const state = reactive({
  price: 10,
  quantity: 1
});

watchEffect(() => console.log(state.price * state.quantity));
// logs '10' right away
// update the price and quantity
state.price = 9;
state.quantity = 3;
// wait for Vue to have run the watcher again
await nextTick();
// logs '27'
```

Comment réécrire `reactive` et `watchEffect` pour faire fonctionner cet exemple ? D'abord, il nous faut savoir quand le prix et la quantité sont modifiés. A-t-on un moyen d'*intercepter* une mise à jour de ces valeurs en JavaScript ? Oui, nous en avons un : avec des getters et des setters !

29.5.1. getter/setter

JavaScript a toujours eu quelques fonctionnalités de "meta-programmation". L'usage de `get` et `set`

en est un exemple.

```
const pony = { name: 'Rainbow Dash' };
console.log(pony.name);
// logs 'Rainbow Dash'
```

Même si cela ressemble à une simple lecture de propriété, tu peux définir un getter pour accéder à cette propriété, et faire faire quelque chose à ce getter chaque fois qu'il est invoqué :

```
const pony = {
  get name() {
    console.log('get name');
    return 'Rainbow Dash';
  }
};
console.log(pony.name);
// logs 'get name'
// logs 'Rainbow Dash'
```

Mais comment faire pour permettre le même genre d'interception sur un objet existant, que tu n'as pas écrit toi-même, et qui n'utilise pas de getter ? Et bien depuis 2011 environ, JavaScript dispose d'une méthode `Object.defineProperty` :

```
const pony = { name: 'Rainbow Dash' };
const value = pony.name;
Object.defineProperty(pony, 'name', {
  get() {
    console.log('get name');
    return value;
  }
});
console.log(pony.name);
// logs 'get name'
// logs 'Rainbow Dash'
```

Cet exemple est simpliste, mais c'est sur ce genre de technique que de nombreux frameworks et librairies s'appuient. Vue 2.x, par exemple, utilisait ce mécanisme pour savoir quand l'état des composants est modifié, et donc déclencher une mise à jour du DOM : il réécrivait chaque propriété en utilisant un setter, qui change la valeur de la propriété bien sûr, mais qui, en plus, notifie le framework que la propriété a changé.

```
Object.defineProperty(component, 'user', {
  set() {
    this.user = user;
    heyVue2APropertyChanged(); // 🔥
  }
})
```

```
});
```

Et il faisait cela pour chaque propriété du composant, au moment de son initialisation. C'était efficace, mais ne couvrait pas certains cas assez communs. Par exemple, ajouter une propriété à un objet après son initialisation ne déclenche pas de changement :

```
component.newProperty = 'hello';
// won't call heyVue2APropertyChanged() 🤔
```

Pour supporter ce cas d'utilisation, Vue 2.x imposait l'usage de `Vue.set(component.newProperty, 'hello')`. Ça fait le job, mais ce n'est pas très intuitif.

`Object.defineProperty`, comme son nom l'indique, fonctionne sur des objets. Mais sur d'autres types, comme les tableaux, ça ne fonctionne pas. Donc là encore, dans Vue 2.x, on ne pouvait pas utiliser `myArray[3] = 'hello'`, parce que Vue ne détectait pas le changement 🤔 (voir la [documentation officielle](#)).

29.5.2. Les *proxies* à la rescousse

C'est là que les *proxies*, une nouvelle fonctionnalité de JavaScript introduite dans la spécification ES2015, peuvent être très utiles. 🎉

Le concept de *proxy* est assez connu en programmation en général. Il décrit un intermédiaire entre un appelant et un appelé. En ES2015, un proxy peut cibler des objets, mais aussi des tableaux, fonctions, ou... d'autres proxies (mais pas les types prédéfinis comme `Date`).

```
const user: UserModel = { name: 'Cédric' };
const handler: ProxyHandler<UserModel> = {};
const uselessProxy: UserModel = new Proxy(user, handler);
```

Le *handler* peut faire plein de choses, par exemple, du *trapping* de propriétés :

```
const handler = {
  get(obj: any, prop: any) {
    console.log(`${prop} was accessed`);
    return obj[prop];
  }
};
const user = { name: 'Cédric' };
const proxy = new Proxy(user, handler);
console.log(proxy.name);
// logs 'name was accessed'
// logs 'Cédric'
```

Il y a de nombreux *traps* disponibles : `get` et `set` bien sûr, mais aussi `has`, `apply`, `construct`, `defineProperty`, `deleteProperty`, etc.

Pour Vue 3.x, cela représente une opportunité très intéressante, puisque les proxies résolvent le problème des propriétés ajoutées dynamiquement :

```
const handler: ProxyHandler<any> = {
  set(obj: any, prop: string | number | symbol, value: any): boolean {
    obj[prop] = value;
    console.log(`\${String(prop)} was updated with \${value}`);
    return true;
  }
};
const user = {};
const proxy = new Proxy(user, handler);
proxy.name = 'Cédric';
// logs 'name was updated with Cédric'
```

Vue 3 utilise donc les proxies au lieu de `Object.defineProperty`, et n'a plus besoin de `Vue.set`. Et comme les proxies permettent aussi d'intercepter les modifications sur les tableaux, `myArray[3] = 'hello'` est bien détecté par Vue 3.

Note que les proxies impliquent un coût additionnel en termes de performance. Ce n'est pas facile à mesurer, mais c'est plus lent que d'utiliser `defineProperty` pour détecter une modification de propriété par exemple. Vue fait tout ce qu'il peut pour être aussi performant que possible, et n'utilise donc les proxies que lorsqu'ils sont nécessaires.

Revenons à notre préoccupation du moment, et réimplémentons les fonctions réactives à l'aide des proxies.

29.5.3. Réimplémenter les fonctions `reactive` and `watchEffect`

Pour rappel : nous voulons invoquer l'effet suivant chaque fois que le prix ou la quantité est modifié.

```
const state = reactive({
  price: 10,
  quantity: 1
});

watchEffect(() => console.log(state.price * state.quantity));
// logs '10' right away
// update the price and quantity
state.price = 9;
state.quantity = 3;
// wait for Vue to have run the watcher again
await nextTick();
// logs '27'
```

On peut commencer par mettre en place un mécanisme basique, qui stocke tous les effets, et les appelle chaque fois qu'une valeur réactive quelconque est modifiée.

Implémentons donc la fonction `watchEffect`. Elle ne fait qu'invoquer l'effet et le stocker dans un tableau :

```
type Effect = () => void;
const effects: Array<Effect> = [];
export function watchEffect(effect: Effect): void {
  effects.push(effect);
  effect();
}
```

Maintenant, réinvoquons tous les effets lorsqu'une valeur réactive change. Il faut pour cela que la fonction `reactive` retourne un proxy qui enveloppe l'objet original. Ce proxy intercepte les modifications, et déclenche l'appel des *watchers* :

```
export function reactive<T extends object>(object: T): T {
  return new Proxy(object, {
    set(obj: T, key: string, value: unknown): boolean {
      // set the value
      (obj as any)[key] = value;
      // recompute all effects
      effects.forEach(effect => effect());
      return true;
    },
    get(obj: T, key: string | number | symbol) {
      return (obj as any)[key];
    }
  });
}
```

Et notre exemple fonctionne !

```
const state = reactive({
  price: 10,
  quantity: 1
});

watchEffect(() => console.log(state.price * state.quantity));
// logs '10' right away

// update the price and quantity
state.price = 9;
// logs '9'
state.quantity = 3;
// logs '27'
```

Pour être plus proche de la réalité, on aimerait aussi détecter les mises à jour d'objets imbriqués, comme Vue le fait :

```

const state = reactive({
  order: {
    price: 10,
    quantity: 1
  }
});

watchEffect(() => console.log(state.order.price * state.order.quantity));
// logs '10' right away

// update the price and quantity
state.order.price = 9;
// logs '9'
state.order.quantity = 3;
// logs '27'

```

Pour cela, on peut transformer récursivement les objets imbriqués en objets réactifs, s'ils ne le sont pas encore, au moment où on y accède :

```

function isObject(value: unknown): boolean {
  return !!value && typeof value === 'object';
}

function isReactive(value: any): boolean {
  return value['_reactive'];
}

export function reactive<T extends object>(object: T): T {
  const proxy = new Proxy(object, {
    set(obj: T, key: string, value: unknown): boolean {
      // set the value
      (obj as any)[key] = value;
      // recompute all effects
      if (key !== '_reactive') {
        effects.forEach(effect => effect());
      }
      return true;
    },
    get(obj: T, key: string | number | symbol) {
      const value = (obj as any)[key];
      // convert the value to its reactive version if it is an object and not already
      // reactive
      const newValue = isObject(value) && !isReactive(value) ? reactive(value) :
      value;
      (obj as any)[key] = newValue;
      return newValue;
    }
  });
  (proxy as { _reactive: boolean })['_reactive'] = true;
  return proxy;
}

```

```
}
```

Note qu'on exécute l'effet immédiatement, dès qu'on détecte un changement. Vue ne fait pas cela : il retient qu'il doit appeler l'effet en le poussant dans une queue, et il n'exécute les effets qu'au prochain "cycle" pour éviter d'appeler le même effet plusieurs fois. C'est la raison pour laquelle, dans l'exemple initial, je faisais appel à `await nextTick()` : c'est ce qui déclenche le viderage de la queue et l'exécution unique de chaque *effet*.

On peut réimplémenter cela relativement simplement, en ajoutant un tableau d'effets à notre solution, et en n'y ajoutant un effet que s'il ne s'y trouve pas déjà.

```
const jobQueue: Array<Effect> = [];
export function reactive<T extends object>(object: T): T {
  const proxy = new Proxy(object, {
    set(obj: T, key: string | number | symbol, value: unknown): boolean {
      // set the value
      (obj as any)[key] = value;
      if (key !== '_reactive') {
        // recompute all effects by adding them to the job queue
        effects.forEach(effect => {
          // but only if not already in there
          if (!jobQueue.includes(effect)) {
            jobQueue.push(effect);
          }
        });
      }
      return true;
    },
    get(obj: T, key: string | number | symbol) {
      const value = (obj as any)[key];
      // convert the value to its reactive version if it is an object and not already
      // reactive
      const newValue = isObject(value) && !isReactive(value) ? reactive(value) :
      value;
      (obj as any)[key] = newValue;
      return newValue;
    }
  });
  (proxy as { _reactive: boolean })['_reactive'] = true;
  return proxy;
}
```

On peut ensuite vider la queue, depuis la fonction `nextTick` :

```
export async function nextTick() {
  // execute all the jobs in the queue
  jobQueue.forEach(effect => effect());
  // empty the job queue
```

```

    jobQueue.length = 0;
    return Promise.resolve();
}

```

Et voilà, ça marche !

```

const state = reactive({
  price: 10,
  quantity: 1
});

watchEffect(() => console.log(state.price * state.quantity));
// logs '10' right away

// update the price and quantity
state.price = 9;
// does not log '9'
// as we have a job queue
state.quantity = 3;
// does not log '27' right away
// we need to call 'nextTick' to drain the job queue
await nextTick();
// log '27'!

```

Notre implémentation a un problème majeur cependant : elle déclenche l'appel de *tous* les effets, chaque fois qu'on modifie une *quelconque* valeur, même si cette nouvelle valeur n'a aucun intérêt pour un effet :

```

// update unrelated reactive value
(state as any).discount = 0.2;
await nextTick();
// logs '27' again!

```

Si Vue faisait cela, les performances de l'application s'écrouleraient rapidement ! On doit donc faire mieux, et ne déclencher l'exécution d'un effet que si son résultat dépend des valeurs modifiées. Et pour accomplir cela, on va tracer les dépendances de l'effet.

29.5.4. Traçage des dépendances

Le plan est donc de n'ajouter à la queue que les effets qui ont besoin d'être exécutés parce qu'une de leur dépendance a été modifiée. Notre effet trace les valeurs de `price` et `quantity`. Nous devons donc ne l'exécuter que si l'une de ces deux valeurs a été modifiée.

Lorsque `watchEffect` est appelé, nous allons donc :

- stocker l'effet en tant qu'*effet courant*
- l'exécuter, pour tracer ses dépendances

```

let currentEffect: Effect | null = null;
export function watchEffect(effect: Effect): void {
  currentEffect = effect;
  effect();
  currentEffect = null;
}

```

On veut à présent, pour chaque propriété dont dépend l'effet courant, ajouter cet effet au tableau d'effets qui dépendent de cette propriété. Lorsque l'effet est exécuté pour la première fois, il accède aux propriétés des objets réactifs qu'il utilise. On peut donc modifier la fonction `reactive`, afin que chaque appel à `get` sur le proxy trace la dépendance :

```

export function reactive<T extends object>(object: T): T {
  const effectsDependingOnKey = new Map<string | number | symbol, Array<Effect>>();
  const proxy = new Proxy(object, {
    set(obj: T, key: string | number | symbol, value: unknown): boolean {
      // set the value
    },
    get(obj: T, key: string | number | symbol) {
      // if we are in the process of tracking dependency for an effect
      if (currentEffect) {
        // get the effect currently depending on this property
        const effects = effectsDependingOnKey.get(key) ?? [];
        // and add the current one
        effectsDependingOnKey.set(key, [...effects, currentEffect]);
      }
      const value = (obj as any)[key];
      // convert the value to its reactive version if it is an object and not already
      // reactive
      const newValue = isObject(value) && !isReactive(value) ? reactive(value) :
      value;
      (obj as any)[key] = newValue;
      return newValue;
    }
  });
  (proxy as { _reactive: boolean })['_reactive'] = true;
  return proxy;
}

```

L'implémentation de `set` est similaire à la version précédente, mais peut cette fois n'exécuter que les effets nécessaires :

```

// for all effects depending on this property
const effects = effectsDependingOnKey.get(key) ?? [];
// re-evaluate them by adding them to the job queue
effects.forEach(effect => {
  // but only if not already in there
  if (!jobQueue.includes(effect)) {

```

```
    jobQueue.push(effect);
}
});
```

Et tout fonctionne !

29.5.5. Réimplémenter les fonctions `ref` et `computed`

Notre petit exemple fonctionne, mais tout s'écroule si on utilise une valeur primitive dans l'effet : si cette valeur primitive est modifiée, le *watcher* ne réexécutera pas l'effet, parce qu'il n'est capable que de tracer les changements effectués sur des objets réactifs.

C'est la raison pour laquelle `ref` existe : elle enveloppe une valeur primitive dans un objet réactif, dont Vue peut tracer les modifications.

Réimplémenter `ref` est donc très simple :

```
export interface Ref<T> {
  value: T;
}

export function ref<T>(value: T): Ref<T> {
  return reactive({ value });
}
```

Comme tu peux le voir, une `ref` n'est rien d'autre qu'un objet `reactive`, avec une propriété unique : `value`.

Qu'en est-il de `computed` ? `computed` est une espèce de `ref` "intelligente" (et en lecture seule). On doit réexécuter sa fonction `getter` chaque fois qu'on lit sa valeur :

```
export function computed<T>(getter: () => T): Ref<T> {
  return {
    get value() {
      return getter();
    }
  };
}
```

Et on peut donc à présent accomplir, bon an mal an, tout ce que Vue accomplit ! 🎉

```
const state = reactive({
  price: 10,
  quantity: 1
});

const total = computed(() => state.price * state.quantity);
```

```
const discount = ref(0.1);
const totalWithDiscount = computed(() => total.value * (1 - discount.value));
console.log(totalWithDiscount.value);
// logs '9'
// update the quantity and discount
state.quantity = 2;
discount.value = 0.2;
await nextTick();
console.log(totalWithDiscount.value);
// logs '16'
```

Bien sûr, Vue va bien plus loin (pour gérer les cas à la marge, être plus performant, etc.) mais tu devrais à présent avoir une idée plus précise de ce qui se passe sous son capot !

Chapter 30. Performances



Attention à ne pas optimiser prématurément. Il faut toujours mesurer avant et après. Fais attention également aux benchmarks que l'on trouve sur les internets : il est assez facile de leur faire dire ce que l'on veut.

Le terme performance peut vouloir dire beaucoup de choses : vitesse, utilisation du CPU (et donc consommation de la batterie), pression sur la mémoire...

Tout n'est pas important pour tout le monde : on peut avoir des besoins différents selon que l'on développe un site mobile, une plate-forme e-commerce, ou une application de gestion classique.

Les performances peuvent être découpées en plusieurs catégories, qui, une fois de plus, peuvent ne pas être aussi importantes les unes que les autres pour toi : premier chargement, rechargement, et performances durant la vie de l'application, ce que l'on appelle le *runtime*.

Le premier chargement se produit lorsqu'un utilisateur charge l'application pour la première fois. Le rechargement se produit quand il ou elle revient consulter l'application. Les performances au *runtime* concernent quant à elles ce qui se passe quand l'application est utilisée. Certains des conseils suivants sont très génériques, et pourraient être appliqués à n'importe quel framework. Nous en parlons quand même parce que l'on pense que c'est intéressant et important à savoir. Et aussi parce que, lorsque l'on parle de performances, le framework est parfois le goulot d'étranglement, mais très (très) souvent, ce n'est pas le cas.

30.1. Premier chargement

Quand tu charges une application web moderne dans ton navigateur, un certain nombre de choses se passent. Premièrement, le fichier `index.html` est chargé et interprété par le navigateur. Puis les scripts JS et les autres ressources référencées sont téléchargés. Quand une des ressources demandées est reçue, le navigateur l'analyse, et l'exécute si c'est un fichier JS.

30.2. Taille des ressources

La première astuce est relativement évidente : fais attention à la taille des ressources chargées !

La phase de chargement des ressources dépend du nombre de ressources que tu veux charger. Beaucoup de ressources veut dire que ce sera long. Des ressources volumineuses également. Particulièrement si le réseau n'est pas très bon, ce qui arrive plus souvent que tu ne le penses : tu testes peut-être l'application sur une bonne connection fibre, mais certains utilisateurs sont peut-être au milieu de nulle part, contraints d'utiliser une 3G lente. Voici ce que tu peux faire pour les aider.

30.3. Faire un beau paquet : le *bundle*

Quand on écrit une application Vue, on a des imports un peu partout, et notre code est réparti dans des dizaines, centaines, voire milliers de fichiers. Mais on ne veut pas que nos utilisateurs chargent des milliers de fichiers ! Donc avant de livrer notre application, nous allons faire ce que l'on appelle

un "bundle" : un seul fichier contenant tous les fichiers JavaScript.

C'est le travail de [Rollup](#) (pour Vite) ou [Webpack](#) (pour la CLI) de prendre tous nos fichiers JavaScript (et CSS, et les fichiers de template HTML) et de construire ces *bundles*.

Ce ne sont pas des outils simples à maîtriser pour être honnête, mais Vite et la CLI font un assez bon travail pour cacher cette complexité. Si tu n'utilises pas Vite ou la CLI, tu peux quand même construire ton application avec Rollup ou Webpack, ou tu peux choisir un autre outil qui produira des résultats peut-être même encore meilleurs (comme [Rollup](#) par exemple). Mais attention, cela demande pas mal d'expertise (et de travail) pour ne pas tout casser, et souvent juste pour gagner quelques kilobytes supplémentaires. J'aurais tendance à te recommander de rester avec Vite (idéalement) ou la CLI. Les équipes travaillant dessus font un très bon travail pour être à jour avec les dernières versions de Vue, de TypeScript et des bundlers.

30.4. *Tree-shaking*

Rollup ou Webpack (ou l'outil que tu utilises) commence par le point d'entrée de ton application (le fichier `main.ts` généré pour toi), et résout ensuite l'arbre des imports, puis produit le *bundle*. C'est assez cool parce que le bundle ne contient ainsi que les fichiers de ta base de code et des bibliothèques tierces que tu as importés. Le reste n'est pas embarqué. Donc même si tu as des dépendances dans ton `package.json` que tu n'utilises plus (et donc que tu n'importe plus), elles ne seront pas dans le *bundle* final.

Les bundlers sont même un petit peu plus malin que ça. Si tu as un fichier `models` qui exporte deux classes, disons `PonyModel` et `RaceModel`, et que tu n'importe que `PonyModel` dans le reste de l'application, mais jamais `RaceModel`, alors le bundler mettra seulement `PonyModel` dans le *bundle* final, et pas `RaceModel`. Ce procédé est appelé **tree-shaking**. Et tous les frameworks et bibliothèques dans l'éco-système font de leur mieux pour être "tree-shakable". En théorie, cela veut dire que ton bundle final ne contient que ce qui est réellement nécessaire. En pratique, les bundlers sont sensiblement plus conservateurs, et ne sont pas capables de comprendre deux ou trois choses à l'heure actuelle. Par exemple, si tu as une classe `Pony` avec deux méthodes `eat` et `run`, mais que tu n'utilise que `run`, le code de la méthode `eat` sera quand même dans le *bundle* final. Donc ce n'est pas parfait, mais cela fait déjà un bon boulot.

30.5. Minification et élimination de code mort

Quand le *bundle* est construit, le code est généralement minifié et le code mort est éliminé. Cela veut dire que toutes les variables, noms de méthodes, noms de classes... sont renommés pour n'utiliser qu'un ou deux caractères à travers toute la base de code. C'est un peu effrayant au premier abord et laisse penser que cela pourrait casser des choses, mais ESBUILD ou Terser fait un bon travail. Ils vont aussi éliminer le code mort qu'ils vont trouver.

30.6. Autres types de ressources

Alors que les sections précédentes concernent spécifiquement le JavaScript, une application contient aussi souvent d'autres types de ressources, comme des styles, des images, des polices de caractères... Tu devrais avoir le même souci d'efficacité à leurs propos, et faire de ton mieux pour

les garder à une taille raisonnable. Appliquer toutes sortes de techniques dingues pour optimiser la taille des *bundles* JS, mais charger des images de plusieurs Mo serait contre-productif sur le temps de chargement et la bande passante ! Comme ce n'est pas réellement le sujet de cet ebook, je ne vais pas creuser ce sujet, mais cette ressource en ligne de Addy Osmani à propos de l'optimisation des images est vraiment excellente : [Essential Image Optimization](#).

30.7. Compression

Tous les navigateurs modernes supportent la compression des ressources, et indiquent au serveur, via un header de la requête HTTP, que le serveur peut compresser la ressource demandée avant de l'envoyer. Cela veut dire que tu peux servir une version compressée à tes utilisateurs, et le navigateur se chargera de la décompresser avant de l'interpréter. C'est vraiment obligatoire car cela sauve une tonne de bande passante et de temps de chargement !

Tous les serveurs du marché ont une option qui permet d'activer la compression des ressources. Généralement le premier utilisateur qui demandera la ressource payera le coût de la compression à la volée, et ensuite les suivants recevront la ressource compressée directement.

L'algorithme de compression le plus courant est GZIP, mais d'autres comme [Brotli](#) sont également populaires.

30.8. Chargement fainéant : le *lazy-loading*

Parfois, en dépit de nos efforts pour garder le *bundle* JS le plus petit possible, on finit avec un gros fichier parce que notre application contient plusieurs dizaines de composants, et utilise plusieurs bibliothèques tierces. Et non seulement ce gros *bundle* va augmenter le temps nécessaire pour télécharger ce JavaScript, mais il va aussi augmenter le temps nécessaire à l'analyse du code JavaScript qu'il contient.

Une solution courante à ce problème est d'utiliser un chargement différé ou *lazy-loading*. Cela veut dire que plutôt que d'avoir un seul gros *bundle*, tu découpes ton application en plusieurs parties et demande au bundler de construire plusieurs *bundles*.

La bonne nouvelle est que Vue (et son routeur) rend cette tâche assez facile à accomplir. Tu peux lire le [chapitre](#) sur le lazy-loading pour en apprendre plus à ce sujet.

Le *lazy-loading* peut améliorer radicalement le temps de chargement : tu peux rendre le premier bundle chargé très petit, avec seulement le code nécessaire pour afficher la page d'accueil, et laisser Vue charger le reste à la demande quand ton utilisateur navigue vers une autre partie de l'application.

30.9. Rendu côté serveur

J'aimerais commencer par préciser que cette technique n'est sans doute destinée qu'à 0.0001% d'entre vous. Le rendu côté serveur (*server side rendering* ou *universal rendering*) est une technique qui consiste à pré-rendre l'application sur le serveur avant de la servir à tes utilisateurs. Avec cette technique, quand une utilisatrice demande [/dashboard](#), elle recevra une version pré-rendue du tableau de bord, plutôt que de recevoir [index.html](#) et ensuite laisser le routeur faire son travail une

fois que Vue aura fini de démarrer.

Vue offre un support natif qui permet de lancer une application dans un environnement différent du navigateur comme par exemple un serveur. Tu peux pré-rendre les pages et les servir à tes utilisateurs. La page va alors s'afficher très vite, puis Vue prendra le relais et fera son travail habituel.

C'est aussi un gros avantage si tu veux que ton site soit découvrable par les moteurs de recherche qui n'exécutent pas le JavaScript, puisque tu peux leur servir une page pré-rendue, plutôt qu'une page blanche.

C'est également une façon de permettre aux réseaux sociaux comme Twitter et Facebook d'afficher un aperçu de ton site. Ces réseaux sociaux vont essayer de faire une capture d'écran d'une URL partagée, mais comme ils n'exécutent pas le JavaScript, ils ne verront rien de ta page générée dynamiquement, à moins que tu ne leur serves une page pré-rendue par le serveur. Donc si tu veux être sûr que l'aperçu soit parfait, par exemple si ton application est un site médiatique, ou un site e-commerce, tu auras peut-être besoin de rendu côté serveur.

La mauvaise nouvelle est que cela augmente la complexité de ton application. Il faudra mettre en place le serveur et réfléchir à la stratégie à adopter. Veux-tu pré-rendre toutes les pages ou seulement quelques-unes ? Veux-tu pré-rendre toute la page, avec les données déjà récupérées et les vérifications de sécurité nécessaires, ou juste des parties critiques de la page ? Veux-tu pré-rendre les pages au build, ou les pré-rendre à la demande et les mettre en cache ? Veux-tu faire cela pour tous les profils d'utilisateur et toutes les langues supportées ou juste pour certaines combinaisons ? Toutes ces questions dépendent du type d'application que tu construis, et l'effort peut beaucoup varier pour atteindre ces objectifs.

Donc, encore une fois, je te conseille de te pencher sur cette technique uniquement si c'est critique pour ton application, et non pas seulement pour la hype...

30.10. Cache pour recharge

Une fois que les utilisateurs ont ouvert l'application une première fois, il est possible d'accélérer les chargements suivants.

Tu devrais toujours mettre en cache les ressources statiques de ton application (images, styles, *bundles JS*...). Tu peux facilement le faire en configurant ton serveur et en utilisant les headers HTTP `Cache-Control` et `ETag`. Tous les serveurs du marché permettent de faire cela, ou tu peux aussi utiliser un CDN. Si tu actives le cache, la prochaine fois que tes utilisateurs chargeront l'application, leur navigateur n'aura même pas à faire de requête HTTP pour récupérer ces ressources car il les aura déjà en cache !

Mais un cache est toujours un peu traître : tu peux avoir besoin de dire au navigateur "Hé ! J'ai déployé une nouvelle version en production, merci de récupérer les nouvelles ressources !".

La façon la plus simple de faire ceci est d'avoir un nom différent pour la ressource mise à jour. Cela veut dire qu'au lieu de déployer une ressource `main.js`, tu déploies `main.xxxx.js` où `xxxx` est un identifiant unique. Cette technique est appelée du *cache busting*, ce qui pourrait se traduire approximativement par "faire péter le cache". Et, encore une fois, Vite et la CLI sont là pour toi : en

mode production, ils vont automatiquement nommer les ressources avec un *hash* unique, calculé grâce au contenu du fichier. Ils mettent également automatiquement à jour les sources des scripts dans `index.html` pour refléter ces noms uniques, les sources des polices de caractères, les sources des fichiers de style, etc.

Si tu utilises Vite ou la CLI, tu peux donc déployer en toute sécurité une nouvelle version et tout mettre en cache, à l'exception du `index.html` (puisque il contiendra les liens vers les nouvelles ressources déployées) !

30.11. Performances à l'exécution

La magie de Vue réside dans son mécanisme de réactivité et son rendu VDOM : le framework détecte les changements de l'état de l'application et modifie le DOM en conséquence. C'est le principal travail de Vue, et donc, en règle générale, le meilleur moyen de rendre l'application performante est de limiter le nombre et la taille des détections de changements et la quantité de modifications à apporter au DOM (créations, modifications, suppressions).

Pour être honnête, la plupart des applications Vue n'auront aucun problème de performance, sans avoir à appliquer une quelconque optimisation. Lis le [chapitre](#) sur la magie de Vue pour savoir pourquoi Vue est performant.

Malheureusement, certains d'entre nous vont devoir recréer Excel dans le navigateur pour leur entreprise, ou ne vont pas échapper à un arbre devant absolument afficher 10 000 clients, ou vont être confrontés à une quelconque autre demande déraisonnable pour une application web. C'est là que, quel que soit le framework utilisé, les choses deviennent plus compliquées. Ces fonctionnalités nécessitent un nombre énorme d'éléments dans le DOM, et de composants Vue dans l'arbre de composants. Les trucs et astuces qui suivent peuvent être efficaces dans ce genre de situation.

30.12. key dans v-for

Cette astuce peut réellement améliorer les performances de `v-for` : ajouter une `key`. Pour que tu comprennes le principe, laisse-moi d'abord t'expliquer comment les frameworks JavaScript modernes (du moins, les plus populaires d'entre eux) gèrent les collections. Suppose qu'on veuille afficher une liste de 3 courses. Le code devrait ressembler à ça :

```
<ul>
  <li v-for="race in races">{{ race.name }}</li>
</ul>
```

Si tu ajoutes une course, Vue va insérer un nouvel élément DOM à la bonne position. Si tu modifies le nom de l'une des courses, Vue modifiera le texte de l'élément `li` correspondant.

Mais comment fait-il cela ? En créant un lien entre les éléments du DOM et les objets contenus dans le tableau. Vue utilise une table de correspondances qui ressemble à ça :

```
node li 1 -> race #e435 // { id: 3, name: London }
```

```
node li 2 -> race #8fa4 // { id: 4, name: Paris }
```

Ça fonctionne très bien. Si tu remplaces un objet par un autre dans le tableau, Vue détruira l'élément du DOM correspondant et en créera un nouveau.

```
node li 1 (recreated) -> race #c1ea // { id: 1, name: Berlin }
node li 2 -> race #8fa4 // { id: 4, name: Paris }
```

Si tous les éléments du tableau sont remplacés par de nouveaux objets, tous les éléments `li` du DOM seront détruits et recréés. Tout cela ne pose pas de problème en général, excepté lorsque le but est simplement de rafraîchir la liste avec un contenu qui sera pratiquement identique. Dans ce cas, il serait bien plus efficace de garder les éléments du DOM existants, et de faire les quelques modifications nécessaires. Le problème, c'est que si tu vas rechercher cette même liste de courses sur le serveur pour la rafraîchir, même si l'état des objets sera sensiblement identique à l'état précédent, les objets, eux (i.e. leur référence) seront tous différents.

La solution est d'aider Vue à maintenir cette table de correspondances sans tout détruire à chaque rafraîchissement, en lui disant d'identifier les objets non plus par leur référence, mais plutôt par une propriété fonctionnelle qui permet de les identifier, typiquement un ID de base de données.

C'est ce que permet de faire `key` :

```
<ul>
  <li v-for="race in races" :key="race.id">{{ race.name }}</li>
</ul>
```

Dans l'exemple ci-dessus, Vue ne créera un nouvel élément dans le DOM que si l'identifiant de la course change. Sur des longues listes dont le contenu change peu, cela permet d'économiser des tas de suppressions et de créations d'éléments. Cette optimisation est plutôt simple à mettre en oeuvre et n'a pas d'inconvénient, donc il ne faut pas hésiter à l'utiliser. Il ne s'agit d'ailleurs pas que d'une optimisation. `key` est aussi nécessaire en cas d'utilisation d'animations. Si le style d'un élément est supposé s'animer (en transitionnant progressivement de la valeur précédente à la nouvelle valeur), et que la liste de courses est remplacée par une nouvelle liste à chaque rafraîchissement, alors `key` est indispensable. Sans elle, l'animation ne va jamais se déclencher, puisque le style des éléments ne change jamais : l'élément lui-même est remplacé par un autre.

30.13. `v-memo`

Vue 3.2 offre une astuce pour les performances : la directive `v-memo`, qui permet d'optimiser agressivement les templates dans certains cas limites. Tu peux penser à `v-memo` comme un équivalent de `shouldComponentUpdate` en React, mais disponible pour les éléments ou composants dans Vue.

Supposons que nous avons un template comme celui-ci :

```
<ul>
```

```
<ul>
  <li v-for="race in races" :key="race.id" v-memo="[race.name]">{{ race.name }}</li>
</ul>
```

Sans `v-memo`, un changement dans l'une des propriétés d'une course, même si elle n'est pas affichée à l'écran comme son pays, entraîne la création d'un nouvel élément `li` virtuel qui sera comparé par l'algorithme de DOM virtuel à la version précédente (relis le [chapitre](#) sur le fonctionnement de Vue sous le capot pour en apprendre plus à ce sujet).

Avec `v-memo`, l'élément virtuel n'est pas recréé et le précédent est réutilisé, sauf quand les conditions de `v-memo` (ici, le nom de la course) changent. Cela peut avoir l'air d'une toute petite amélioration, mais c'est en fait un gros gain de performances si tu affiches de grandes listes d'éléments. Le populaire (bien que discutable) [benchmark des performances des frameworks JS](#) a un benchmark testant exactement ce cas-là, et Vue v3.2 a parmi les meilleurs résultats avec cette fonctionnalité, sans avoir besoin de créer un composant dédié pour chaque ligne et d'astuces pour améliorer les performances (comme `shouldComponentUpdate` en React).

Tu peux voir que `v-memo` accepte un tableau de conditions, donc tu peux écrire quelque chose comme ça :

```
<ul>
  <li v-for="race in races" :key="race.id" v-memo="[race.name, selectedRaceId ===
    race.id]">
    <span :class="{ selected: selectedRaceId === race.id }">{{ race.name }}</span>
  </li>
</ul>
```

Alors le `li` ne sera mis à jour que si le nom de la course change, ou le test de la course sélectionnée a un résultat différent.

Note que si la liste affiche une propriété non listée dans les conditions de `v-memo`, alors la liste ne sera pas mise à jour quand la propriété change :

```
<ul>
  <li v-for="race in races" :key="race.id" v-memo="[race.name]">{{ race.name }} - {{
    race.country }}</li>
</ul>
```

Si le pays de la course change, la liste n'est pas mise à jour. Si le nom de la course change, alors la liste sera mise à jour, et affichera le nouveau nom et le nouveau pays.

Cette astuce n'est réellement utile que pour afficher des très grandes listes, et ne sera pas nécessaire dans la plupart des cas.

30.14. Conclusion

Ce chapitre, j'espère, t'as appris quelques techniques permettant de résoudre d'éventuels problèmes de performance. Mais avant tout, retiens les règles d'or de l'optimisation des

performances (en anglais dans le texte) :

- don't
- don't... yet
- profile before optimizing.

Comme l'a dit un célèbre informaticien :

l'optimisation prématuée est la source de tous les maux.

— Donald Knuth

Alors pense d'abord à écrire du code aussi simple, correct et lisible que possible, et ne pense à profiler l'application, puis à l'optimiser, que si tu as réellement un problème de performance.

Chapter 31. Ce n'est qu'un au revoir

Merci d'avoir lu ce livre !

Quelques chapitres seront ajoutés dans des versions ultérieures sur des sujets avancés, et d'autres surprises. Ils ont encore besoin d'être fignolés, mais je pense que tu les apprécieras. Bien sûr, nous tiendrons à jour ce contenu avec les nouvelles versions du framework et des différentes bibliothèques, pour que tu ne rates pas les nouvelles fonctionnalités à venir. Et toutes ces révisions du livre seront gratuites, bien évidemment !

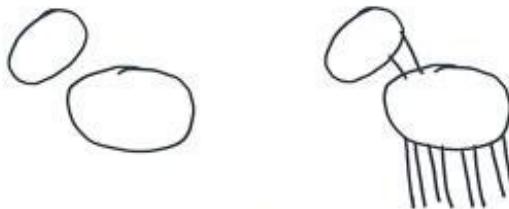
Si tu as aimé cette lecture, parles-en à tes amis !

Et si tu ne l'as pas déjà, sache qu'on propose [un cours en ligne, le Pack Pro](#), avec cet ebook. Ce cours te donne accès à toute une série d'exercices qui permettent de construire un projet complet pas à pas, en partant de zéro. Pour chaque étape on fournit les tests unitaires et de bout-en-bout nécessaires à une couverture de code de 100%, des instructions détaillées (qui ne sont pas un simple copier/coller, elles te feront réfléchir et comprendre ce que tu utilises) et une solution (qui sans trop se vanter est probablement la plus élégante, ou en tout cas celle conforme à l'état de l'art). Un outil maison calcule ton score pour l'exercice, et ta progression est visible sur un tableau de bord. Si tu cherches des exemples de code concrets, toujours à jour, qui peuvent te faire gagner des heures de développement, cette version pro est là pour toi ! Tu peux même [essayer gratuitement les premiers exercices](#), pour te faire une idée concrète. Et vu que tu es déjà le possesseur de cet ebook, on te remercie de ton soutien avec un généreux code de réduction pour le pack pro que tu peux aller chercher [à cette adresse](#).

On a essayé de te donner toutes les clés, mais l'apprentissage du web se passe souvent comme ça :

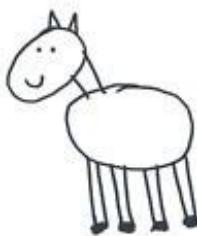
HOW TO: DRAW A HORSE

BY VAN OKTOP

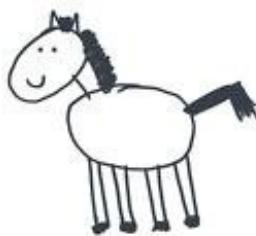


① DRAW 2 CIRCLES

② DRAW THE LEGS



③ DRAW THE FACE



④ DRAW THE HAIR



⑤
ADD
SMALL
DETAILS.

How to draw a horse. Credit to Van Oktop.

Alors on propose aussi une formation, en France et en Europe essentiellement, mais pourquoi pas dans le monde entier (et c'est aussi possible à distance). On peut aussi faire du conseil pour aider ton équipe, ou même travailler avec toi pour t'aider à construire ton produit. Envoie un email à hello@ninja-squad.com et on en discutera !

Mais surtout, j'adorerais avoir ton retour sur ce que tu as aimé, adoré, ou détesté dans ce livre. Cela peut être une petite faute de frappe, une grosse erreur, ou juste pour nous raconter comment tu as trouvé le job de tes rêves grâce à ce livre (on ne sait jamais...).

Je ne peux pas conclure sans remercier quelques personnes. Ma compagne tout d'abord, qui a été d'un soutien formidable, même quand je passais mon dimanche sur la dixième réécriture d'un chapitre dans une humeur détestable. Mes collègues ninjas, pour leur travail et retours sans

relâche, leur gentillesse et encouragements, et pour m'avoir laissé le temps nécessaire à cette folle idée. Et mes amis et ma famille, pour les petits mots qui redonnent l'énergie au bon moment.

Et merci surtout à toi, pour avoir acheté ce livre, et l'avoir lu jusqu'à cette toute dernière ligne ❤️.

À très bientôt.

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: 3.3.4

A.1. 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.2. 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.3. 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.4. 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.5. 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 recommendation 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.6. 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.7. 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.8. 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.9. v3.0.11 - 2021-04-02

A.10. 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.11. 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.12. 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.13. 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.14. 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.15. v3.0.0-beta.10 - 2020-05-11

Global

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