Become a

NINJA

with

Vue.js

ninja squad

Become a ninja with Vue

Ninja Squad

Table of Contents

Chapter 1. Introduction

So you want to be a ninja, huh? Well, you're in good hands! We have a long road, you and me, with lots of things to learn.

We're living exciting times in Web development. There is a new Vue!

As frontend developers, we have tons of JavaScript frameworks available. React and Angular are still going strong. As you may know, we are big fans of Angular. I am a regular contributor to the framework, close to the core team. In our small company, we have built several projects with it, trained hundreds of developers (yes, really), and even written a book about it.

Angular is incredibly productive once you have mastered it. Despite all of this, it doesn't prevent us from seeing what a wonderful tool Vue is.

My adventure with Vue started a few years ago, as a "let's have fun with Vue 2" day. I just wanted to see how Vue worked, and wanted to build a small application to make my own opinion. After years of teaching Angular, I immediately realized Vue was much easier to learn but still quite powerful. A lot of things are similar between the two frameworks, but Vue strikes a nice balance.

So I liked Vue 2.x very much. I even started to write this book around that time. One thing was bothering me though: the TypeScript integration was… not great, to say the least. And if there is one thing I definitely love when I work with Angular, it's its nearly perfect integration with TypeScript.

That's why when Vue 3.0 was announced as a complete rewrite in TypeScript in September 2018, I was super happy, and started working again on what you read now.

I've followed the development of Vue 3 very closely, reading every commit (yes, really), and even modestly contributed some bug fixes and tiny features. In the same time we started some customers project in Vue, and next thing you know, I was contributing to the framework, the testing library, the form library… sharing my "open-source free time" between Vue and Angular.

Vue has a lot of interesting points, and a vision that few other frameworks have. This ebook is a side effect of my open-source contributions, and my desire to share what I love about Vue. It is fascinating for me to see how different frameworks solve similar problems, and I hope I'll share my enthusiasm with you ❤.

The ambition of this ebook is to evolve with Vue. You will receive (free!) updates with the best practices, and some new features as they emerge (and with fewer typos, because, despite our countless reviews, there are probably some left…). I would love to hear back from you - if some chapters aren't clear enough, if you spot a mistake or if you have a better way for some parts.

I'm fairly confident about the code samples, though, as they are all in a real project, with several dozens of unit and end-to-end tests. It was the only way to write an ebook with a newborn framework, and to be able to catch all the problems that inevitably arose with each release.

All the code is written in TypeScript, because we strongly believe it is an amazing tool. You can of course write your Vue applications in JavaScript, but you'll see that TypeScript is not very intrusive when you use Vue, and can provide great value. Even if you are not convinced by TypeScript (or Vue) in the end, I'm pretty sure you will have learnt a thing or two along your read.

If you have bought our online training, the "Pro package" (thank you!), you'll build a small application piece by piece along the book. This application is called **PonyRacer**, and it is a web application where you can bet on pony races. You can even test the application here! Go on, I'll wait for you.

Fun, isn't it?

But it's not just a fun application, it's a complete one. You'll have to write components, forms, tests, use the router, call an HTTP API (that we have already built) and even do Web Sockets. It uses Vite and has all the pieces you'll need for writing a real app. Each exercise will come with a skeleton, a set of instructions and a few tests. Once all the tests pass, you have completed the exercise!

The first 6 exercises of the Pro Pack are free and available on vue-exercises.ninja-squad.com. The other ones are only accessible if you buy our online training. At the end of every chapter, we will link to the exercises of the Pro Pack that are related to the features explained in the chapter, mark the free ones with the following label: 🐴, and mark the other ones with the following label: 🦄.

If you did not buy the "Pro package" (but really you should), don't worry: you'll learn everything that's needed. But you will not build this awesome application with beautiful ponies in pixel art. Your loss 😊!

You will quickly see that, beyond Vue itself, we have tried to explain the core concepts the framework uses. The first chapters don't even talk about Vue: they are what I call the "Concept Chapters", which will help you level up with the new and exciting things happening in our field.

Then we will slowly build our knowledge of the framework (and of its most popular libraries), with components, templates, directives, forms, the new Composition API, HTTP communication, routing, how to write tests…

And finally we will learn about the advanced topics. But that's another story for later.

Enough with the introduction, let's start with the features introduced in the recent ECMAScript versions, and then we'll learn about TypeScript.

The ebook is using Vue version 3.4.34 for the examples.

Chapter 2. A gentle introduction to ECMAScript 2015+

If you're reading this, we can be pretty sure you have heard of JavaScript. What we call JS is one implementation of a standard specification, called ECMAScript. The spec version you know the most about is version 5, that has been used these last years.

But, in 2015, a new version of the spec was released, called ECMAScript 2015, ES2015, or sometimes ES6, as it was the sixth version of the specification. And since then, we have had a yearly release of the specification (ES2016, ES2017, etc.), with a few new features every year. From now on, I'll mainly say ES2015, as it is the most popular way to reference it, or ES2015+ to reference ES2015, ES2016, ES2017, etc. It adds A LOT of things to JavaScript, like classes, constants, arrow functions, generators… It has so much stuff that we can't go through all of it, as it would take the whole book. But Vue has been designed to take advantage of the brand-new version of JavaScript. And, even if you can still use your old JavaScript, things will be more awesome if you use ES2015+. So we're going to spend some time in this chapter to get a grip on what ES2015+ is, and what will be useful to us when building a Vue app.

That means we're going to leave a lot of stuff aside, and we won't be exhaustive on the rest, but it will be a great starting point. If you already know ES2015+, you can skip these pages. And if you don't, you will learn some pretty amazing things that will be useful to you even if you end up not using Vue in the future!

2.1. Transpilers

The sixth version of the specification reached its final state in 2015. So it's now supported by modern browsers, but there are still browsers in the wild that don't support it yet, or only support it partially. And of course, now that we have a new specification every year (ES2016, ES2017, etc.), some browsers will always be late. You might be thinking: what's the point of all this, if I need to be careful on what I can use? And you'd be right, because there aren't that many apps that can afford to ignore older browsers. But, since virtually every JS developer who has tried ES2015+ wants to write ES2015+ apps, the community has found a solution: a transpiler.

A transpiler takes ES2015+ source code and generates ES5 code that can run in every browser. It even generates the source map files, which allows you to debug directly the ES2015+ source code from the browser. Back in 2015, there were two main alternatives to transpile ES2015+ code:

- Traceur, a Google project, historically the first one but now unmaintained.
- Babeljs, a project started by a young developer, Sebastian McKenzie (17 years old at the time, yeah, that hurts me too), with a lot of diverse contributions.

The source code of Vue itself was at first transpiled with Babel, before switching to TypeScript for the version 3.0. TypeScript is an open source language developed by Microsoft. It's a typed superset of JavaScript that compiles to plain JavaScript, but we'll dive into it very soon.

Let's be honest: Babel has waaaay more steam than Traceur nowadays, so I would advise you to use it. It is now the de-facto standard in this area.

So if you want to play with ES2015+, or set it up in one of your projects, take a look at these transpilers, and add a build step to your process. It will take your ES2015+ source files and generate the equivalent ES5 code. It works very well but, of course, some new features are quite hard or impossible to transform in ES5, as they just did not exist. However, the current state is largely good enough for us to use without worrying, so let's have a look at all these shiny new things we can do in JavaScript!

2.2. let

If you have been writing JS for some time, you know that the `var` declaration is tricky. In pretty much any other language, a variable is declared where the declaration is done. But in JS, there is a concept, called "hoisting", which actually declares a variable at the top of the function, even if you declared it later.

So declaring a variable like `name` in the `if` block:

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

is equivalent to declaring it at the top of the function:

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

ES2015 introduces a new keyword for variable declaration, `let`, behaving much more like what you would expect:

```
function getPonyFullName(pony) {
  if (pony.isChampion) {
    let name = 'Champion ' + pony.name;
    return name;
  }
  // name is not accessible here
  return pony.name;
```

```
}
```

The variable `name` is now restricted to its block. `let` has been introduced to replace `var` in the long run, so you can pretty much drop the good old `var` keyword and start using `let` instead. The cool thing is, it should be painless to use `let`, and if you can't, you have probably spotted something wrong with your code!

2.3. Constants

Since we are on the topic of new keywords and variables, there is another one that can be of interest. ES2015 introduces `const` to declare… constants! When you declare a variable with `const`, it has to be initialized, and you can't assign another value later.

```
const poniesInRace = 6;
```

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

As for variables declared with `let`, constants are not hoisted and are only declared at the block level.

One small thing might surprise you: you can initialize a constant with an object and later modify the object content.

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

But you can't assign another object:

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

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

Same thing with arrays:

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

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

2.4. Shorthands in object creation

Not a new keyword, but it can also catch your attention when reading ES2015 code. There is now a shortcut for creating objects, when the object property you want to create has the same name as the variable used as the value.

Example:

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

can be simplified to:

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

Similarly, when you want to define a method in the object:

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

you can simplify it to:

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

2.5. Destructuring assignment

This new feature can also catch your attention when reading ES2015 code. There is now a shortcut

for assigning variables from objects or arrays.

In ES5:

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

Now, in ES2015, you can do:

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

And you will have the same result. It can be a little disturbing, as the key is the property to look for in the object, and the value is the variable to assign. But it works great! Even better: if the variable you want to assign has the same name as the property, you can simply write:

```
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
```

The cool thing is that it also works with nested objects:

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

And the same is possible with arrays:

```
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
```

Of course, it also works for arrays in arrays, or arrays in objects, etc.

One interesting use of this can be for multiple return values. Imagine a function `randomPonyInRace`

that returns a pony and its position in a race.

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

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

The new destructuring feature assigns the position returned by the method to the position variable, and the pony to the pony variable! And if you don't care about the position, you can write:

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

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

And you will only have the pony.

It is also possible to assign a value to the destructured variable if it is `undefined` in the object:

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

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

The `position` variable will now contain 1.

2.6. Default parameters and values

One of the characteristics of JavaScript is that it allows developers to call a function with any number of arguments:

- if you pass more arguments than the number of the parameters, the extra arguments are ignored (well, you can still use them with the special `arguments` variable, to be accurate).
- if you pass fewer arguments than the number of the parameters, the missing parameter will be

set to `undefined`.

The last case is the one most relevant to us. Usually, we pass fewer arguments when the parameters are optional, like in the following example:

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

The optional parameters usually have a default value. The OR operator will return the right operand if the left one is `undefined`, as will be the case if the parameter was not provided (to be completely accurate, if it is *falsy*, i.e `0`, `false`, `""`, etc.). Using this trick, we can then call the function `getPonies` with:

```
getPonies(20, 2);
getPonies(); // same as getPonies(10, 1);
getPonies(15); // same as getPonies(15, 1);
```

This worked alright, but it was not really obvious the parameters were optional ones with default values, without reading the function body. ES2015 introduces a more precise way to have default parameters, directly in the function definition:

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

Now it is perfectly clear the `size` parameter will be `10`, and the `page` parameter will be `1` if not provided.

There is a small difference though, as now `0` or `""` are valid values and will not be replaced by the default one, as `size = size || 10` would have done. It will be more like `size = size === undefined ? 10 : size;`.

The default value can also be a function call:

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

or even other variables, either global variables, or other parameters of the function:

```
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);
}
```

This mechanism for parameters can also be applied to values, for example when using a destructuring assignment:

```
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 introduces a new syntax to define variable parameters in functions. As said in the previous part, you could always pass extra arguments to a function and get them with the special `arguments` variable. So you could have done something like this:

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

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

But I think we can agree that it's neither pretty nor obvious: since the `ponies` parameter is never used, how do we know that we can pass several ponies?

ES2015 gives us a way better syntax, using the rest operator `...`:

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

`ponies` is now a true array on which we can iterate. The `for ... of` loop used for iteration is also a new feature in ES2015. It makes sure that you iterate over the collection values, and not also over its properties as `for ... in` would do. Don't you think our code is prettier and more obvious now?

The rest operator can also work when destructuring data:

```
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
```

The rest operator is not to be confused with the spread operator which, I'll give you that, looks awfully similar! But the spread operator is the opposite: it takes an array and spreads it in variable arguments. The only examples I have in mind are functions like min or max, that receive variable arguments, and that you might want to call on an array:

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

2.8. Classes

One of the most emblematic new features: ES2015 introduces classes to JavaScript! You can now easily use classes and inheritance in JavaScript. You always could, using prototypal inheritance, but that was not an easy task, especially for beginners.

Now it's very easy, take a look:

```
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
```

Class declarations, unlike function declarations, are not hoisted, so you need to declare a class before using it. You may have noticed the special function `constructor`. It is the function being called when we create a new pony, with the `new` operator. Here it needs a color, and we create a new Pony instance with the color set to "blue". A class can also have methods, callable on an instance, as the method `toString()` here.

It can also have static attributes and methods:

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

Static methods can be called only on the class directly:

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

A class can have getters and setters, if you want to hook onto these operations:

```
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'
```

And, of course, if you have classes, you also have inheritance out of the box in 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 is called the base class, and Pony the derived class. As you can see, the derived class has the methods of the base class. It can also override them:

```
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
```

As you can see, the keyword `super` allows calling the method of the base class, with `super.speed()` for example.

The `super` keyword can also be used in constructors, to call the base class constructor:

```
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

Promises are not so new, and you might know them or use them already, as they were available via third-party libraries. But since you will use them a lot in Vue, and even if you're just using JS, I think it's important to make a stop.

Promises aim to simplify asynchronous programming. Our JS code is full of async stuff, like AJAX requests, and usually we use callbacks to handle the result and the error. But it can get messy, with callbacks inside callbacks, and it makes the code hard to read and to maintain. Promises are much nicer than callbacks, as they flatten the code, and thus make it easier to understand. Let's consider a simple use case, where we need to fetch a user, then their rights, then update a menu when we have these.

With callbacks:

```
getUser(login, function (user) {
  getRights(user, function (rights) {
```

```
    updateMenu(rights);
  });
});
```

Now, let's compare it with promises:

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

I like this version, because it executes as you read it: I want to fetch a user, then get their rights, then update the menu.

As you can see, a promise is a 'thenable' object, which simply means it has a `then` method. This method takes two arguments: one success callback and one reject callback. The promise has three states:

- pending: while the promise is not done, for example, our server call is not completed yet.
- fulfilled: when the promise is completed with success, for example, the server call returns an OK HTTP status.
- rejected: when the promise has failed, for example, the server returns a 404 status.

When the promise is fulfilled, then the success callback is called, with the result as an argument. If the promise is rejected, then the reject callback is called, with a rejected value or an error as the argument.

So, how do you create a promise? Pretty simple, there is a new class called `Promise`, whose constructor expects a function with two parameters, `resolve` and `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');
    }
  });
};
```

Once you have created the promise, you can register callbacks, using the `then` method. This method can receive two parameters, the two callbacks you want to call in case of success or in case of failure. Here we only pass a success callback, ignoring the potential error:

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

Once the promise is resolved, the success callback (here simply logging the user on the console) will be called.

The cool part is that it flattens the code. For example, if your resolve callback is also returning a promise, you can write:

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

but more beautifully:

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

Another interesting thing is the error handling, as you can use one handler per promise, or one for all the chain.

One per 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);
    }
  )
```

One for the chain:

```
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
  })
```

You should seriously look into Promises, because they are going to be the new way to write APIs, and every library will use them. Even the standard ones: the new Fetch API does for example.

2.10. Arrow functions

One thing I like a lot in ES2015 is the new arrow function syntax, using the 'fat arrow' operator (⇒). It is SO useful for callbacks and anonymous functions!

Let's take our previous example with promises:

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

can be written with arrow functions like this:

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

How cool is it? THAT cool!

Note that the return is also implicit if there is no block: no need to write `user => return getRights(user)`. But if we did have a block, we would need the explicit return:

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

And it has a special trick, a great power over normal functions: the `this` stays lexically bounded, which means that these functions don't have a new `this` as other functions do. Let's take an example, where you are iterating over an array with the `map` function to find the max.

In 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);
```

You would expect this to work, but it doesn't. If you have a good eye, you may have noticed that the `forEach` in the `find` function uses `this`, but the `this` is not bound to an object. So `this.max` is not the `max` of the `maxFinder` object... Of course, you can fix it easily, using an 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);
```

or binding the `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);
```

or even passing it as a second parameter of the `forEach` function (as it was designed for):

```
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);
```

But there is now an even more elegant solution with the arrow function syntax:

```
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);
```

That makes the arrow functions the perfect candidates for anonymous functions in callbacks!

2.11. Async/await

We were talking about promises earlier, and it's worth knowing that another keyword was introduced to handle them more synchronously: `await`.

This is not a feature introduced in ECMAScript 2015 but in ECMAScript 2017, and to use `await`, your function must be marked as `async`. When you use the `await` keyword in front of a Promise, you pause the execution of your `async` function, wait for the Promise to resolve, and then resume the execution of the `async` function. The returned value will be the resolved value.

So we can write our previous example using `async/await` like this:

```
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();
```

And your code now looks like it is synchronous! Another cool feature of `async/await` is that you can use a simple `try/catch` to handle errors:

```
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 that async/await is still asynchronous, although it looks like synchronous. The function execution is paused and resumed, but just like with callbacks, this doesn't block the thread: other JavaScript events can be handled while the execution is paused.

2.12. Sets and Maps

This is a short one: you now have proper collections in ES2015. Yay \o/! We used to have dictionaries filling the role of a map, but we can now use the class `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
```

We also have a class `Set`:

```
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
```

You can iterate over a collection, with the new syntax `for ⋯ of`:

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

You'll see that the `for ⋯ of` syntax is also supported by Vue to iterate over a collection in a template.

2.13. Template literals

Composing strings has always been painful in JavaScript, as we usually have to use concatenation:

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

Template literals are a new small feature, where you have to use backticks (`) instead of quotes or

simple quotes, and you have a basic templating system, with multiline support:

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

The multiline support is especially great when you are writing HTML strings, as we will do for our Vue components:

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

One last feature is the ability to tag them. You can define a function, and apply it to a template string. Here `askQuestion` adds an interrogation point at the end of the string:

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

So what's the difference with a simple function? The tag function in fact receives several arguments:

- an array of the static parts of the string
- the values resulting from the evaluation of the expressions

For example if we have a template string containing expressions:

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

then the tag function will receive the various static and dynamic parts. Here we have a tag function to uppercase the names of the protagonists:

```
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?'
```

Let's now talk about one of the big changes introduced: modules.

2.14. Modules

A standard way to organize functions in namespaces and to dynamically load code in JS has always been lacking. Node.js has been one of the leaders in this, with a thriving ecosystem of modules using the CommonJS convention. On the browser side, there is also the AMD (Asynchronous Module Definition) API, used by RequireJS. But none of these were a real standard, thus leading to endless discussions on what's best.

ES2015 aims to create a syntax using the best from both worlds, without caring about the actual implementation. The Ecma TC39 committee (which is responsible for evolving ES2015 and authoring the specification of the language) wanted to have a nice and easy syntax (that's arguably CommonJS's strong suit), but to support asynchronous loading (like AMD), and a few goodies like the possibility to statically analyze the code by tools and support cyclic dependencies nicely. The new syntax handles how you export and import things to and from modules.

This module thing is really important in Vue, as pretty much everything is defined in modules, which you have to import when you want to use them. Let's say I want to expose a function to bet on a specific pony in a race, and a function to start the race.

In races.service.js:

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

As you can see, this is fairly easy: the new keyword `export` does a straightforward job and exports the two functions.

Now, let's say one of our application components needs to call these functions.

In another file:

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

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

That's what is called a *named export*. Here we are importing the two functions, and we have to specify the filename containing these functions - here 'races.service'. Of course, you can import only one method if you need, and you can even give it an alias:

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

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

And if you want to use all the exported symbols (functions, constants, classes etc.) from the module, you can use a wildcard '*'.

As you would do with other languages, use the wildcard with care, only when you really want all the functions, or most of them. As this will be analyzed by our IDEs, we will see auto-import soon and that will free us from the bother of importing the right things.

With a wildcard, you have to use an alias, and I kind of like it, because it makes the rest of the code clearer:

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

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

If your module exposes only one function or value or class, you don't have to use named export, and you can leverage the default keyword. It works great for classes for example:

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

Notice the lack of curly braces to import a default. You can import it with the alias you want, but to be consistent, it's better to call the import with the module name (except if you have multiple modules with the same name of course, then you can choose an alias that allows you to distinguish them). And of course, you can mix default export with named ones, but obviously with only one default per module.

In Vue, you're going to use a lot of these imports in your app. Each component will be an object, generally isolated in its own file and exported, and then imported when needed in other components.

2.15. Conclusion

That ends our gentle introduction to ES2015+. We skipped some other parts, but if you're comfortable with this chapter, you will have no problem writing your apps in ES2015+. If you want

to have a deeper understanding of this, I highly recommend Exploring JS by Axel Rauschmayer or Understanding ES6 from Nicholas C. Zakas… Both ebooks can be read online for free, but don't forget to buy it to support their authors. They have done great work! Actually I've re-read Speaking JS, Axel's previous book, and I again learned a few things, so if you want to refresh your JS skills, I definitely recommend it!

Chapter 3. Going further than ES2015+

3.1. Dynamic, static and optional types

You may have heard that Vue apps can be written in ES5, ES2015+ or TypeScript. And you may be wondering what TypeScript is, or what it brings to the table.

JavaScript is dynamically typed. That means you can do things like:

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

And it works. That's great for all sort of things, as you can pass pretty much any object to a function, and it works, as long as the object has the properties the function needs:

```
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
```

This dynamic nature allows wonderful things, but it is also a pain for a few other reasons compared to more statically-typed languages. The most obvious might be when you call an unknown function in JS from another API, you pretty much have to read the doc (or, worse, the function code) to know what the parameter should look like. Take a look at our previous example: the method `printColor` needs a parameter with a `color` property. That can be hard to guess, and of course it is much worse in day-to-day work, where we use various libraries and services developed by fellow developers. One of Ninja Squad's co-founders is often complaining about the lack of types in JS, and finds it regrettable he can't be as productive and write as good code as he would in a more statically-typed environment. And he is not entirely wrong, even if he is sometimes ranting for the sake of it too! Without type information, IDEs have no real clue if you're doing something wrong, and tools can't help you find bugs in your code. Of course, we have tests in our apps, and Vue has always been keen on making testing easy, but it's nearly impossible to have a perfect test coverage.

That leads to the maintainability topic. JS code can become hard to maintain, despite tests and documentation. Refactoring a huge JS app is no easy task, compared to what could be done in other statically-typed languages. Maintainability is a very important topic, and types help humans and tools to avoid mistakes when writing and maintaining code.

Vue started as a pure JavaScript framework, but the core team wanted to help us to write better JS, by adding some type information to our code. It's not a very new concept in JS. It was even the subject of the ECMAScript 4 specification, which was later abandoned.

That's why Vue supports TypeScript, the Microsoft language, since Vue 2.0. The support for TypeScript in Vue 2.0, however, was not perfect. So when Vue 3.0 was announced, one of the big features was an improved support of TypeScript: in fact the framework itself is now written in TypeScript!

3.2. Enters TypeScript

I think this was a smart move for several reasons. TypeScript is very popular, with an active community and ecosystem. We use it a lot to build front-end applications, and, honestly, it's a great language.

TypeScript is a Microsoft project. But it's not the Microsoft you have in mind, from the Ballmer and Gates years. It's the Microsoft of the Nadella era, the one opening up to its community, and, well, open-source.

The main reason to bet on TypeScript is the type system it offers. It's an optional type system that helps without getting in the way. In fact, after coding some time with it, it's hard to go back to raw JavaScript. You can still write Vue apps just using JavaScript, but using TypeScript will improve the experience.

As I said, I really enjoy this language, and we will have a look at what TypeScript offers in the next section. At the end, you'll have enough understanding to read any Vue code, and you'll be able to choose whether you want to use it or not (or just a little), in your apps.

You may be wondering: why use typed code in Vue apps? From our experience, the main reason is the ease to refactor. You will usually have types representing your entities: a `User`, an `Account`, an `Invoice`, whatever… But it's quite rare to have a perfect model on the first go. Over the time, the entities evolve, or grow, or split into several ones. Fields get new names, and in a JavaScript application, you get no guarantee that you haven't broken half your pages… This is where TypeScript shines: the compiler guides you to change everything that needs to be changed, and you'll sleep better at nights.

That's why we're going to spend some time learning TypeScript (TS). And maybe one day the type system will make its way through the standard committee, we'll have types in JS, and all this will be usual.

Ready? Let's dive in!

Chapter 4. Diving into TypeScript

TypeScript has been around since 2012. It's a superset of JavaScript, adding a few things to ES5. The most important one is the type system, giving TypeScript its name. From version 1.5, released in 2015, the library is trying to be a superset of ES2015+, including all the shiny features we saw in the previous chapter, and a few new things as well, like decorators. Writing TypeScript feels very much like writing JavaScript. By convention, TypeScript files are named with a `.ts` extension, and they will need to be compiled to standard JavaScript, usually at build time, using the TypeScript compiler. The generated code is very readable.

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

But let's start with the beginning.

4.1. Types as in TypeScript

The general syntax to add type info in TypeScript is rather straightforward:

```
let variable: type;
```

The types are easy to remember:

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

In such cases, the types are optional because the TS compiler can guess them (it's called "type inference") from the values.

The type can also come from your app, as with the following class `Pony`:

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

TypeScript also supports what some languages call "generics", for example for an array:

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

The array can only contain ponies, and the generic notation, using `<>`, indicates this. You may be wondering what the point of doing this is. Adding types information will help the compiler catch possible mistakes:

```
ponies.push('hello'); // error TS2345
```

```
// Argument of type 'string' is not assignable to parameter of type 'Pony'.
```

So, if you need a variable to have multiple types, does it mean you're screwed? No, because TS has a special type, called `any`.

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

It's really useful when you don't know the type of a value, either because it's from a dynamic content or from a library you're using.

If your variable can only be of type `number` or `boolean`, you can use a union type:

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

4.2. Enums

TypeScript also offers `enum`. For example, a race in our app can be either `ready`, `started` or `done`.

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

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

The enum is in fact a numeric value, starting at 0. You can set the value you want, though:

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

Since TypeScript 2.4, you can even specify a string value:

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

```
}
```

To be honest though, we don't use enums a lot in our projects: we use union types. They are simpler and cover roughly the same use-cases:

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

TypeScript even allows you to create your own types, so you could do something like:

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

4.3. Return types

You can also set the return type of a function:

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

If the function returns nothing, you can show it using `void`:

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

4.4. Interfaces

That's a good first step. But as I said earlier, JavaScript is great for its dynamic nature. A function will work if it receives an object with the correct property:

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

This function can be applied to any object with a `score` property. How do you translate this in TypeScript? It's easy: you define an interface, which is like the "shape" of the object.

```
function addPointsToScore(player: { score: number }, points: number): void {
```

```
  player.score += points;
}
```

It means that the parameter must have a property called `score` of the type `number`. You can name these interfaces, of course:

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

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

You'll see that we often use interfaces throughout the book to represent our entities. We use interfaces for our models in our other projects as well. We usually append a `Model` suffix to make it clear. It's then very easy to create a new entity:

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

4.5. Optional arguments

Another treat of JavaScript is that arguments are optional. You can omit them, and they will become `undefined`. But if you define a function with typed parameter in TypeScript, the compiler will shout at you if you forget them:

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

To show that a parameter is optional in a function (or a property in an interface), you can add `?` after the parameter. Here, the `points` parameter could be optional:

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

4.6. Functions as property

You may also be interested in describing a parameter that must have a specific function instead of a property:

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

The interface definition will be:

```
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

A class can implement an interface. For us, the `Pony` class should be able to run, so we can write:

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

The compiler will force us to implement a `run` method in the class. If we implement it badly, by expecting a `string` instead of a `number` for example, the compiler will yell:

```
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.
```

You can also implement several interfaces if you want:

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

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

And an interface can extend one or several others:

```
interface Animal extends CanRun, CanEat {}

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

When you're defining a class in TypeScript, you can have properties and methods in your class. You may realize that properties in classes are not a standard ES2015 feature. It is only possible in TypeScript.

```
class SpeedyPony {
  speed = 10;

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

Everything is public by default, but you can use the `private` keyword to hide a property or a method. If you add `private` or `public` to a constructor parameter, it is a shortcut to create and initialize a private or public member:

```
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
```

Which is the same as the more verbose:

```
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`);
  }
}
```

These shortcuts are really useful, and we'll rely on them a lot in Vue!

4.8. Working with other libraries

When working with external libraries written in JS, you may think we are doomed because we don't know what types of parameter the function in that library will expect. That's one of the cool things with the TypeScript community: its members have defined interfaces for the types and functions exposed by the popular JavaScript libraries!

The files containing these interfaces have a special `.d.ts` extension. They contain a list of the library's public functions. A good place to look for these files is DefinitelyTyped. For example, if you want to use the testing library Jest in your TypeScript application, you can download the proper file from the repo directly with NPM:

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

Even cooler, since TypeScript 1.6, the compiler will auto-discover the type definitions of an NPM library if they are packaged with the library itself. More and more projects are adopting this approach, and so is Vue. So you don't even have to worry about including the interfaces in your Vue project: the TS compiler will figure it out by itself if you are using NPM to manage your dependencies!

So my advice would be to give TypeScript a try! All my examples from here will be in TypeScript, as

Vue and all the tooling around are really designed for it.

Chapter 5. Advanced TypeScript

If you're just starting to learn TypeScript, you can safely skip this chapter for now and come back later. This chapter is here to showcase some more advanced usages of TypeScript. They'll only make sense if you already have some familiarity with the language.

5.1. readonly

You can use the `readonly` keyword to mark the property of a class or interface as... read only! That way, the compiler will refuse to compile any code trying to assign a new value to the property:

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

const config: Config = { timeout: 2000 };
// `config.timeout` is now readonly and can't be reassigned
```

5.2. keyof

The `keyof` keyword can be used to get a type representing the union of the names of the properties of another type. For example, you have a `PonyModel` interface:

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

You want to build a function that returns the value of a property. You could implement a naive version:

```
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');
```

Two problems here:

- you can give any value to the `key` parameter, even keys that don't exist on `PonyModel`.
- the return type being `any`, you are losing a lot of type information.

This is where `keyof` can shine. `keyof` allows you to list all the keys of a 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
```

So we can use this type to make `getProperty` safer, by declaring that:

- the first parameter is of type `T`
- the second parameter is of type `K`, which is a key of `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');
```

We killed two birds with one stone here:

- `key` can now only be an existing property of `PonyModel`
- the return value will be inferred by TypeScript (which is pretty awesome!)

Now let's see how we can leverage `keyof` to do even more.

5.3. Mapped type

Let's say you want to create a type that has exactly the same properties as `PonyModel`, but you want every property to be optional. You can of course define it manually:

```
interface PartialPonyModel {
  name?: string;
  color?: string;
  speed?: number;
}
```

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

But you can do something more generic with a mapped type:

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

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

The `Partial` type is a transformation that applies the `?` modifier to every property of a type! In fact, you don't have to define the type `Partial` yourself, because since version 2.1, it's part of the language itself, and it's declared exactly like in the above example.

TypeScript offers others mapped types out of the box.

5.3.1. Readonly

`Readonly` makes all the properties of an object `readonly`:

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

5.3.2. Pick

`Pick` helps you build a type with only some of the original properties:

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

5.3.3. Record

`Record` helps you build a type with the same properties as another type, but with a different type:

```
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 }
};
```

There are even more than that, but these are the most useful.

5.4. Union types and type guards

Union types are really handy. Let's say your application has authenticated users and anonymous users, and sometimes you need to do a different action depending on that. You can model this as:

```
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;
}
```

I don't know about you, but I don't like these `as ⋯` explicit casts. Maybe we can do better?

One possibility is to use a type guard, a special function whose sole purpose is to help the

TypeScript compiler.

```
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;
}
```

This is better! But we still need to return a default value, even if we covered all the possibilities.

We can slightly improve the situation if we drop the type guards and use a union type instead.

```
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!
}
```

This is even better, as TypeScript automatically narrows the type in the `else` branch.

Sometimes you know that the model will grow in the future, and that more cases will need to be handled. For example if you introduce an `AdminUser`. In that case, you can use a `switch`. A `switch` statement will break if one of the cases is not handled. So introducing our `AdminUser`, or another type of user later, would automatically add compilation errors in every place you need to handle it!

```
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;
  }
}
```

I hope these patterns will help you. Now let's focus on Web Components.

Chapter 6. The wonderful land of Web Components

Before going further, I'd like to make a brief stop to talk about Web Components. You don't have to know about Web Components to write Vue code. But I think it's a good thing to have an overview of what they are, because some choices in Vue have been made to facilitate the integration with Web Components, or to make the components we will build similar to Web Components. Feel free to skip this part if you have no interest in this topic; however, I do believe you'll learn a thing or two that will be useful for the rest of the road.

6.1. A brave new world

Components are an old fantasy in development. Something you can grab off the shelves and drop into your app, something that would work right away and bring a needed functionality to your users.

My friends, this time has come.

Well, maybe. At least, there is the start of something.

That's not completely new. We have had components in web development for quite some time, but they usually require some kind of dependency, like jQuery, Dojo, Prototype, AngularJS, etc. Not necessarily libraries you wanted to add to your app.

Web Components attempt to solve this problem: let's have reusable and encapsulated components.

They rely on a set of emerging standards that browsers don't perfectly support yet. But, still, it's an interesting topic, even if there's a chance we'll have to wait a few years to use them fully, or even if the concept never takes off.

This emerging standard is defined in 3 specifications:

- Custom elements
- Shadow DOM
- Template

Note that the samples are most likely to work in a recent Chrome or Firefox browser.

6.2. Custom elements

Custom elements are a new standard allowing developers to create their own DOM elements, making something like `<ns-pony></ns-pony>` a perfectly valid HTML element. The specification defines how to declare such elements, how to make them extend existing elements, how to define your API, etc.

Declaring a custom element is done using `customElements.define`:

```
class PonyComponent extends HTMLElement {

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

}

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

And you can then use it:

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

Note that the name must contain a dash, so that the browser knows it is a custom element. Of course, your custom element can have properties and methods, and it also has lifecycle callbacks, to be able to execute code when the component is inserted or removed, or when one of its attributes changes. It can also have a template of its own. Maybe the `ns-pony` displays an image of the pony or just its name:

```
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>';
  }

}
```

If you try to look at the DOM, you'll see `<ns-pony><h1>General Soda</h1></ns-pony>`. But that means the CSS and JavaScript logic of your app can have undesired effects on your component. So, usually, the template is hidden and encapsulated in something called Shadow DOM, and you'll only see `<ns-pony></ns-pony>` if you inspect the DOM, despite the fact that the browser displays the pony's name.

6.3. Shadow DOM

With a mysterious name like this, you expect something with great powers. And surely it is. The

Shadow DOM is a way to encapsulate the DOM of our component. This encapsulation means that the stylesheet and JavaScript logic of your app will not apply on the component and ruin it inadvertently. It gives us the perfect tool to hide the internals of a component, and be sure nothing leaks from the component to the app, or vice versa.

Going back to our previous example:

```
class PonyComponent extends HTMLElement {

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

}
```

If you try to inspect it now you should see:

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

Now, even if you try to add some style to the `h1` elements, the visual aspect of the component won't change at all: that's because the Shadow DOM acts like a barrier.

Until now, we just used a string as a template of our web component. But that's usually not the way you do that. Instead, the best practice is to use the <template> element.

6.4. Template

A template specified in a `<template>` element is not displayed in your browser. Its main goal is to be cloned in an element at some point. What you declare inside will be inert: scripts don't run, images don't load, etc. Its content can't be queried by the rest of the page using usual methods like `getElementById()` and it can be safely placed anywhere in your page.

To use a template, it needs to be cloned:

```
<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. Frameworks on top of Web Components

All these things put together make the Web Components. I'm far from being an expert on this topic, and there are all sorts of twisted traps on this road.

As Web Components are not fully supported by every browser, there is a polyfill you can include in your app to make sure it will work. The polyfill is called web-component.js, and it's worth noting that it is a joint effort from Google, Mozilla and Microsoft among others.

On top of this polyfill, a few libraries have seen the light. All aim to facilitate working with Web Components, and often come with some ready-to-use Web Components.

Among the most notable initiatives, you find:

- Polymer, the first attempt from Google
- LitElement, a more recent project from the Polymer team ;
- X-tag from Mozilla and Microsoft
- Stencil.

I won't go into the details, but you can easily use an already existing component. Let's say you want a Google Map in your app:

```
<!-- 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>
```

There are a LOT of components out there. You can have an overview on https://www.webcomponents.org/.

You can do a lot of cool things with LitElement and other similar frameworks, like two-way data binding, default values for attributes, emit custom events, react to attribute changes, repeat elements if we give a collection to a component, etc.

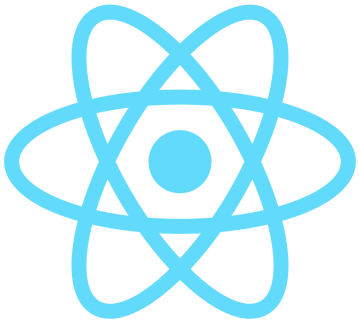
That's obviously far too short a chapter to tell you everything there is to say on Web Components, but you'll see that some concepts are going to pop out along your read. And you'll definitely see that Vue has been designed to make it easy to use Web Components with our Vue components. It is even possible to export our own Vue components as Web Components.

Chapter 7. Grasping Vue's philosophy

To write a Vue application, you have to grasp a few things on the framework's philosophy.

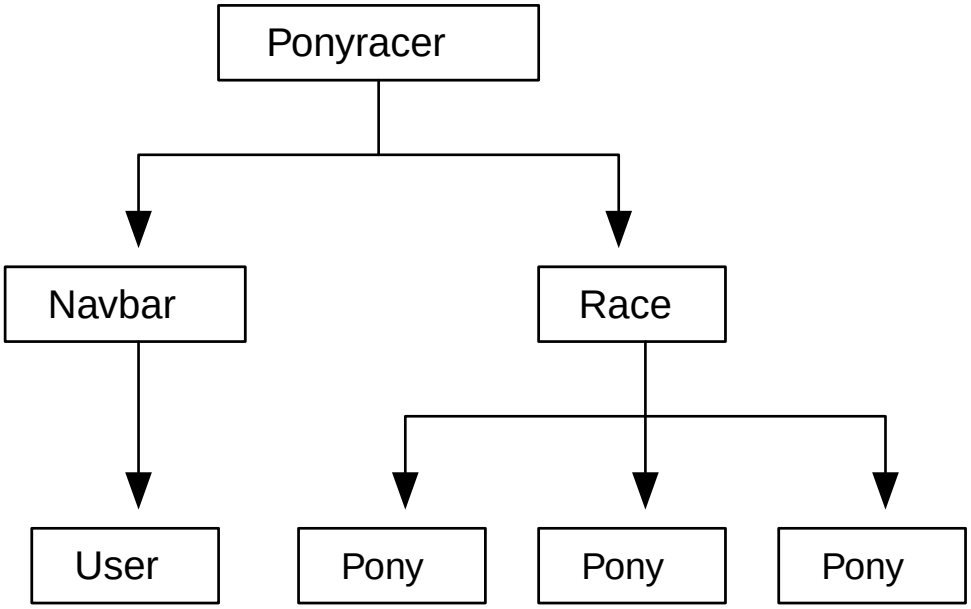
First and foremost, Vue is component-oriented. You will write tiny components and, together, they will constitute a whole application. A component is a group of HTML elements in a template, dedicated to a particular task. For this, you will usually also need to have some logic linked to that template, to populate data, and react to events for example.

This component orientation is something that is becoming widely shared across front-end frameworks: React, the cool kid from Facebook, has been doing it that way from the beginning; Ember and AngularJS have their way of doing something similar; and others like Svelte or Angular are betting on building small components too.



Vue is not alone in this, but it is among the first to really care about the integration of Web Components (the standard ones). But let's forget about this for now, as it is a more advanced topic.

Your components will be arranged hierarchically, like the DOM is. A root component will have child components, each of them will also have children, etc. If you want to display a pony race (who wouldn't?), you'll have something like an app (`Ponyracer`), displaying a navbar (`Navbar`) with the logged-in user (`User`), and a child view (`Race`), displaying, of course, the ponies (`Pony`) in the races:



Writing components will be your everyday work, so let's see what it looks like. The Vue team wanted to harness another goodness of today's web development: ES2015+. But to have the best experience possible, it's also possible to write our applications using TypeScript. I hope you already know all of that, as I just spent two chapters on these things!

Vue has the specificity to offer the possibility to write all the pieces of a component in the same file, containing at the same time the view part in HTML and the behavior logic in TypeScript. You can also add the styling part in CSS, or SCSS, etc. These components are called *Single File Components* (SFC is their cute shorter name). The file extension is then `.vue`. It is slightly different from what most frameworks do, but you get used to it.

For example, if we simplify, the Race component could look like this:

Race.vue

```
<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>
```

If you already know another templating language, the template should look familiar, with expressions in curly braces {{ }}, which will be evaluated and replaced by the corresponding values. I don't want to go too deep for now, merely just give you a feel of what the code looks like. Of course, we'll study components and templates in the following chapters.

A component is a very isolated piece of your app. Your app *is* a component like the others.

You can also take available components from the community and just put them in your app, and be able to enjoy their features. These components and features are usually packaged as a Vue plugin.

Such plugins can offer UI components, or drag and drop capability, or validation for your forms, or whatever you can think of. We'll talk about a few useful plugins later.

In the next chapters, we are going to explore how to get started, how to build a small component, your first module and the templating syntax.

We'll also spend some time to learn how to test a Vue application. I love writing tests, and watching the progress bar go all green in my IDE. It makes me feel I'm doing a good job. So there will be an entire chapter on testing everything: your components, your services, your UI…

Vue has a magic feeling, where changes are automatically detected by the framework and applied to the model and the views. Each framework has its own mechanic to do so: we'll check out how Vue works, and we'll explain concepts like proxies, Virtual DOM and other black magic terms behind all that (don't worry, it's not that complicated).

Vue is also a complete ecosystem which provides a lot of help for performing common tasks in web development. Writing forms, calling an HTTP backend, routing, animations, you name it: you're covered.

Well, that's a lot of things to learn! We should start with the beginning: bootstrap an app and write our first component.

Chapter 8. From zero to something

Now that we know a bit more about ECMAScript, TypeScript and the philosophy of Vue, let's get our hands dirty and start a new application.

8.1. The progressive framework

Vue always marketed itself as a progressive framework that, unlike other alternatives like Angular or React, you can adopt progressively. You can take your existing static HTML, or jQuery application and easily sprinkle a bit of Vue on top of it.

So first I'd like to demonstrate how easy it is to set up Vue.

Let's make an empty `index.html` file:

index.html

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

Now let's add some HTML for Vue to handle:

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>
```

The curly braces around `user` are specific to the Vue templating syntax, indicating that `user` should be replaced by its value. We'll explain everything in details in the following chapter, don't worry.

If you load the page in your browser, you'll see that it displays `Hello {{ user }}`. That's normal, as we haven't used Vue yet.

Now let's add Vue. Vue is released on NPM and some sites (called CDNs, for Content Delivery

Network) make NPM packages available for inclusion in our HTML pages. Unpkg is one of them. We can use it to add Vue to our page. Of course, you could also choose to download the file and serve it by yourself.

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>
```

We are using the latest version of Vue in this example. You can specify any version you want by adding `@version` after `https://unpkg.com/vue` in the URL. This version of the ebook is using `vue@3.4.34`.

If you reload the application, you'll see that Vue emits a warning in the console, informing us that we are using a development version. You can use `vue.global.prod.js` to use the production version, and make it disappear. The production doesn't do any checks in our code, is minified, and is a bit faster.

We now need to create our application. Vue offers a `createApp` function to create an application. To call it, we need a root component.

To create a component, we simply need to create an object that defines it. This object can have various properties, but for now we'll just add a `setup` function. Again we'll explain thoroughly later, but the name is explanatory enough: this function is here to set up the component, and Vue is going to call it for us when the component is initialized.

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>
```

The `setup` function just returns an object with a property `user` and a value for this property. But if you reload your page, still nothing happens: we need to call `createApp` with our component.

You need a recent enough browser to load this page, as, as you can see, it uses a "modern" JavaScript syntax.

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` creates an application that needs to be "mounted" in some place in the DOM: here we use the div with the ID `app`. If you reload the page, you should now see `Hello Cédric`. Congratulations, you have your first Vue application.

Maybe we can add another component? We'll build another one, displaying the number of unread messages.

Let's add a new object called `UnreadMessagesComponent`, with a similar `setup` property:

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>
```

Unlike the root component which is using the template inside the `#app` div, we want to define a template for `UnreadMessagesComponent`. This can be done by adding a `script` tag with a special type `text/x-template`. This type guarantees that the browser won't care about this script. You can then reference the template by its ID inside the component definition:

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>
```

We want to be able to insert the unread messages component inside our main template. To do that, we need to tell the root component it's allowed to use the unread messages component, and we need to assign it a PascalCase name:

index.html

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

```
      };
      const app = Vue.createApp(RootComponent);
      app.mount('#app');
    </script>
  </body>
</html>
```

We can now use the tag `<unread-messages></unread-messages>` (which is the dash-case version of `UnreadMessages`) to insert the component where we want:

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>
```

Comparing to other frameworks, a Vue application is super easy to start: just pure JavaScript and HTML, no tooling, components are simple objects. Even someone that doesn't know Vue can understand what's going on. And this is one of the strengths of the framework: it's easy to start,

easy to grasp, and you can progressively learn the features.

We *could* stick to this minimal setup for our projects, but, let's face it, it will not scale for long. We will soon have too many components to fit in one file, we would really love to use TypeScript instead of JavaScript, to add tests, to add some kind of code analysis, etc.

We *could* set up all the needed tools by hand, but instead let's leverage the work of the community and use the Vue CLI (that has been the standard for many years), or the now recommended tool Vite.

8.2. Vue CLI

The CLI is now in maintenance mode, and the recommended tool is Vite, that we present below. As a lot of existing projects use the CLI, we still think it's worth introducing, and it can help to grasp the differences with Vite.

The Vue CLI (Command Line Interface) was born to help developers build Vue applications. It can scaffold an application and build it, and offers a large ecosystem of plugins. Each plugin offers some kind of features, like unit testing or linting or TypeScript support. It also offers a graphical user interface!

One of the cool features of the CLI is the ability to develop each component in a dedicated file, with a `.vue` extension. In this file you can define everything related to this component: its JavaScript/TypeScript definition, its HTML template, and even its CSS styles. This is called a Single File Component, or SFC.

The CLI is overall super handy to avoid having to learn and configure all the underlying tools (Node.js, NPM, Webpack, TypeScript, etc...). It is still very flexible, and you can configure most behaviors.

But the CLI is now in maintenance mode, and Vite is the recommended alternative. Let's talk about the underlying reasons.

8.3. Bundlers: Webpack, Rollup, esbuild

When writing modern JavaScript/TypeScript applications, you often need a tool that can bundle all the assets (code, styles, images, fonts).

For a long time, Webpack was the undisputed favorite. Webpack comes with a simple but super handy feature: it understands all the JavaScript module types that exist (modern ECMAScript modules, but also AMD or CommonJS modules, formats that were existing before the standard). This understanding makes it easy to use pretty much any library you can find on the Internet (most often on NPM): you just install it, import it in one of your files, and Webpack takes care of the rest. Even if you use libraries with completely different formats, Webpack happily converts them and packages all your code and the code of the libraries together into one giant JS file: a bundle. This is a super important task, because even if the standard defined ES Modules back in 2015, most browsers have been supporting them very recently!

The other task of Webpack is to help during development, by providing a dev server and watching your project (it can even do HMR, a fancy word that stands for Hot Module Reloading). When something changes, Webpack reads the entrypoint of our application (`main.ts` for example), then it reads its imports and loads these files, then it reads the imports of the imported files and loads them… You get the idea! When everything is loaded, it re-bundles everything into one large file, both your code and the imported libraries from your `node_modules`, changing the module format if needed. The browser then reloads to display our changes . This can be time-consuming when working on large projects with hundreds or thousands of files, even if Webpack comes with caches and heuristics to be as fast as possible.

Vue CLI (like a lot of tools out there) is using Webpack for most of its work, both when building the application with `npm run build`, or when running the dev server with `npm run serve`.

This is great as the Webpack ecosystem is incredibly rich in plugins and loaders: you can do pretty much what you want with it. On the other hand, a Webpack configuration can quickly get a bit overwhelming with all these options.

If I talk about Webpack and what bundlers do, it's because we have serious alternatives nowadays, and it can be hard to understand what they do, and what are their differences. To be honest, I'm not sure that I understand all the details myself, and I've contributed quite a lot to Vue and Angular CLIs, both heavily based on Webpack! But let me try to explain anyway.

A serious contender is Rollup. Rollup intends to keep things simpler than Webpack, by not doing so much out of the box, but often doing it faster than Webpack. Its author is Rich Harris, who is also the author of the Svelte framework. Rich wrote a famous article called "Webpack and Rollup: the same but different". His guideline is "Use Webpack for apps, and Rollup for libraries". In fact, Rollup can do a lot of what Webpack does for production builds, but it does not come with a dev server that can watch your files during development.

Another incredible alternative is esbuild. Unlike Webpack and Rollup, esbuild itself is not written in JavaScript. It is written in Go and compiled to native code. It has also been designed with parallelism in mind. That makes it way faster than Webpack and Rollup. Like 10x-100x faster .

So why don't we use esbuild instead of Webpack? That's exactly what Evan You, the author of Vue, thought when developing Vue 3. He had another brilliant idea. In 2018, Firefox shipped the support of native ECMAScript Modules (often called native ESM). In 2019, it was Node.js, and then most browsers followed. Nowadays, your personal browser can probably understand native ESM without issues. Evan imagined a tool that would serve files as native ESM to the browser, doing the heavy lifting with esbuild to transform source files into ESM files if needed (for example for TypeScript or Vue files or legacy module formats).

Vite (the French word for "fast") was born.

8.4. Vite

The idea behind Vite is that, as modern browsers support ES Modules, we can now use them directly, at least during development, instead of generating a bundle.

So when you load a page in the browser when developing with Vite, you don't load a single large

file of JS containing all the application: you load just the few ES modules needed for this page, each in their own file (and each over their own HTTP request). If an ES module has imports, then the browser loads these imports as well.

So Vite is mainly a dev server, in charge of answering the browser requests, and responding with the requested ES modules. As we may have written our code in TypeScript, or using SFC in `.vue` extension (see below), Vite sometimes needs to transform the files on our disk into a proper ES module that the browser can understand. This is where esbuild comes into play! Vite is built on top of esbuild, and when a requested file needs to be transformed, it asks esbuild to do the job and then sends the result to the browser. If you change something in a file, then Vite only sends the updated module to the browser, instead of having to rebuild the whole bundle as Webpack-based tools do!

Vite also uses esbuild to optimize a few things. For example if you use a library with a ton of files, it "pre-bundles" it into a single file using esbuild and serves it to the browser in one request instead of a few dozens/hundreds. This pre-bundling is done once when starting the server, so you don't pay the cost every time you refresh.

The fun thing is that Vite is not tied to Vue: it can be used with Svelte, React and others. In fact some other frameworks now recommend to use Vite! Svelte, from Rich Harris, was one of the first to do so, and now officially recommends it.

esbuild is really good for the JS bundling part, but it is not (yet) capable of splitting the application in several bundles, or properly handling CSS (whereas Webpack and Rollup do it out of the box). So it is not suited for bundling the application for production. That's where Rollup comes into play: Vite relies on esbuild during development, but uses Rollup to bundle for production. Maybe in the future it'll use esbuild for everything.

Vite is more than just an esbuild wrapper. As we saw, esbuild transforms files really fast. But Vite does not ask esbuild to transpile the requested files on every reload: it leverages the browser cache to do as little as possible. So if you load a page that you already loaded, it will be displayed in an instant. Vite also comes with a ton of other features, and a rich plugin ecosystem.

An important note: esbuild transpiles TypeScript to JavaScript, but it does not compile it: it completely ignores the type-checking part! That makes it super fast, but it also means that you have no typechecking from Vite during development. To check that your application properly compiles, you have to run Volar (`vue-tsc`), usually when building the application.

Are you excited? Because I am! Vite comes with project templates for React, Svelte and Vue, but the Vue team started a small project on top of Vite called `create-vue`. And that project is now the official recommendation when you start new Vue 3 projects.

8.5. create-vue

create-vue is built on top of Vite, and provides templates for Vue 3 projects.

To get started, you simply use:

```
npm create vue@3
```

The `npm create something` command in fact downloads and executes the `create-something` package. So here `npm create vue` executes the `create-vue` package.

You then have to choose:

- a project name
- if you want TypeScript or not
- if you want JSX or not
- if you want Vue router or not
- if you want Pinia (state management) or not
- if you want Vitest for unit testing or not
- if you want Cypress for e2e testing or not
- if you want ESLint/Prettier for linting and formatting or not

and your project is ready!

We will of course deep-dive into all these technologies along the book.

Want to give it a try?

To build your first Vite app, follow our online exercise Getting Started 🏇! It's part of our Pro Pack, but is accessible to all of our readers.

Done?

If you followed the instructions (and reached a 100% score I hope!), you have an application up and running. Let's look at a few files. The entry point of the application is a file called `main.ts`:

main.ts

```
import './assets/main.css';

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

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

It mounts an `App` component defined in `App.vue`, looking similar to this when created:

App.vue

```
<script setup lang="ts">
import HelloWorld from './components/HelloWorld.vue';
import TheWelcome from './components/TheWelcome.vue';
</script>

<template>
  <header>
```

```
    <img alt="Vue logo" class="logo" src="./assets/logo.svg" width="125" height="125"
/>

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

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

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

`App.vue` is a Single File Component. Let's look into it!

8.6. Single File Components

A Single File Component (SFC) defines:

- the template in the `template` element
- the component definition in a `script` element. Note the `lang="ts"` attribute indicating we are using TypeScript.
- the CSS of the component in a `style` element.

The tooling compiles the TypeScript code to JavaScript for us. It also compiles the template to JavaScript (we'll explain in a later chapter what happens under the hood). The Vue compiler, unlike the browser, understands the PascalCase syntax, so we can use either `<hello-world>` or `<HelloWorld>` for our child component. We'll use the PascalCase version from now on.

As you saw in the exercise, Vite can execute the unit tests, end-to-end tests, linter… Each exercise comes with the unit tests and e2e tests already written, so you'll be able to check your code at every step. And Vite is super fast, so the developer experience is really enjoyable 🚀.

Now that we peeked into what the tooling can do for us, and are ready to build more components, let's move on to the template syntax.

Chapter 9. The templating syntax

We've seen that a component needs to have a template. To simplify things, a template helps us to render HTML with some dynamic parts depending on our data. And to do so Vue offers a templating syntax, that we need to learn before going further.

Let's take a simple example, that just displays a static title:

App.vue

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

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

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

As you can see, the `<template>` element in our Single File Component contains a very simple `<h1>`.

Now we want to display some dynamic data on this first page, maybe the number of users registered into our app.

Later we'll see how to get data from a server, but for now we'll say that this number of users is directly hard-coded in our component.

If you want the template to have access to some data defined in the component, this data must be a property of the object returned by the `setup` function:

App.vue

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

export default defineComponent({
  name: 'App',

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

Now, how do we change our template to display this value? The answer is *interpolation*.

9.1. Interpolation

Interpolation is a big word for a simple concept.

Quick example:

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

Note that the wrapping `div` is not necessary as Vue templates now allow having several root elements (that was not the case until Vue 3.0).

We have a component that will be activated every time Vue finds an `App` tag. The component has a property, `numberOfUsers`. And the template has been augmented with an `<h2>` tag, using the famous double curly braces (a.k.a. "mustaches") to indicate that an expression has to be evaluated. This kind of templating is called interpolation.

We should now see in the browser:

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

`{{ numberOfUsers }}` is replaced by the value of the expression inside the mustaches. When Vue detects an `App` element in the page, it creates an instance of the component, and this instance is the evaluation context of the template's expressions. Here the instance defines a `numberOfUsers` property with the value 146, so we have '146' displayed on screen.

As it is, if the value of `numberOfUsers` changes later, it is not reflected in the 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>
```

If you want Vue to reflect the changes of a variable, you have to create a reactive property using `ref`. We'll come back to the `ref` syntax in details in a following chapter dedicated to components. But for now the only thing you need to know is that a `ref` is just an object with a `value` property.

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
    setTimeout(() => (numberOfUsers.value = 147), 3000);
    return { numberOfUsers };
  }
```

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

So when you want to update your reactive property, you have to update its `value`. But you can still use the `ref` in the template as is, without having to use `.value`: Vue is automatically unwrapping the value for you.

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

Now, whenever the value of `numberOfUsers` changes in our component, the template will be automatically updated! It's one of the great features of Vue. As I was mentioning, we'll come back to the reactivity part in a following chapter: let's keep our focus on templates.

One important fact to remember: if we try to display a property that does not exist, then, instead of displaying `undefined`, Vue is going to display an empty string. The same will happen for a `null` property.

Let's say that, instead of a simple value, our first component has a more complex `user` object, reflecting the current user.

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>
```

As you can see, we can interpolate more complex expressions, like accessing the property of an

object defined in our `setup`.

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

Fun fact: if you interpolate a complete object, you'll see the JSON structure of this object, which is super handy when developing:

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>
```

What happens if we have a typo in our template, with a property that does not exist in the component?

App.vue

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

When loading the app, you will have an error, telling you that this property does not exist:

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

You only get this warning when you open your application in your browser. But, sadly, the compilation raises no warning if you have an error in a template, unlike some other frameworks like Angular. Hopefully you also have unit tests in your application that would catch the problem early enough, before going to production. We'll see how to unit test components and their template in a few chapters.

There are some initiatives from the community to have template type-checking at compile time though:

- Volar (the most advanced currently, that we use in our projects, this ebook examples and the Pro Pack exercises)
- VTI from the Vetur team
- VueDX All these projects are experimental but worth giving a try.

Let's go back to our example. We are now displaying a greeting message. Maybe we can go a step further and display the upcoming pony races. We could start by adding more HTML and TypeScript to our `App` component, but we want to keep it small and testable, and we might want to re-use the races list later.

That should lead us to write our second component. For now, we'll just make it simple:

Races.vue

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

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

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

Nothing fancy: a simple component, with a name `Races`, and a template displaying a `h2` title.

Now we want to include this component in our `App` template. What do we need to do?

9.2. Using other components in our templates

We have our app component, `App`, where we want to display the pony races component, `Races`.

App.vue

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

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

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

As you can see, we added the `Races` component in the template, by including a tag whose name matches the name we defined for the `Races` component.

Buuuuuut, that will not work: your browser will not display the races component.

If you open the developer console, you'll see:

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

Why is that? The reason is simple: Vue doesn't know about this `Races` component yet.

9.2.1. Local registration

But the fix is simple. In our `App` component, we can add a `components` field, to register the components we are using in its template. The syntax is very straightforward: `components` is an object, and you can add a key, which is the name of the element you want to use in your template, and a corresponding value, which is the type of the component you want to use to replace this element.

Here we add a `Races` key, as this is the element we added in our template, and a `Races` value, the component we want to use. Of course, we have to import the `Races` component from the file we declared it.

App.vue

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

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

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

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

It would also work to use `<races></races>` in the template, but most developers stick to the original PascalCase name.

The key could be completely different though. You *could* choose to use `{ RacesList: Races }`, and then use `<RacesList></RacesList>` in the template. You could also use `<races-list></races-list>`, the kebab-case version, in the template. It would definitely work, but it can be a bit surprising for your colleagues. We usually stick to the name of the component.

Note that the ES6 syntax allows writing `{ key }` instead of `{ key: key }` as you may recall from our first chapters. We can thus slightly simplify our declaration to `{ Races }` instead of `{ Races: Races }`. We'll use the shorter version from now on.

9.2.2. Global registration

If you use a component in a lot of other components, it gets a bit boring to have to register your component *every single time*.

Vue offers a global registration that can be leveraged in that case. In your `main` file, you can use the `component` function to register one or several components globally. These components are then available in all the templates without having to register them in each component.

main.ts

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

9.3. Property binding with `v-bind`

Interpolation is only one of the ways to have dynamic parts in your template.

In Vue, every attribute of HTML elements can take a dynamic value, not with the mustaches, but by prefixing the attribute with `v-bind:`.

`v-bind` is what we call a *directive* in Vue. Vue offers several directives that we'll see in details, but you can easily recognize them from standard HTML as they begin with `v-`.

So if you want an image with a dynamic source attribute, for example a property of our component called `dynamicUrl`, you can write:

```
<img v-bind:src="dynamicUrl" />
```

It also works with boolean attributes, like `disabled` or `selected`, which are slightly different, as what matters in HTML is if they present or not. With Vue, you can give a value to these boolean attributes, and if the value is "truthy" in the JavaScript sense, i.e. not `false`, `null`, `undefined`, an empty string or 0, then the attribute is considered present:

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

In this example, `isDisabled` is a property declared in our component, and can be `true` or `false`. If it's true, the button is disabled, and the user can't click on it, if it's false, the button is enabled and clickable.

9.3.1. Shorthand syntax

You can also write a shorter version of `v-bind:attribute` by using `:attribute`. This is super handy, and most Vue developers prefer this shorthand syntax:

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

Since Vue v3.4, it is possible to use an even shorter syntax, when the key and value have the same name!

```
<button :disabled>Log in</button>
```

9.3.2. Properties and attributes

I've been talking about attributes since the beginning of this section, but its title is "Property binding" and not "Attribute binding". Why is that?

The title is referring to DOM properties, but maybe this is not clear for you what the difference is. What we write is HTML markup, containing tags with attributes, right? Let's take this simple HTML:

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

The `input` tag above has two *attributes*: a `type` attribute, and a `value` attribute. When the browser parses this tag, it creates a corresponding DOM node (an `HTMLInputElement` if we want to be

accurate), which has the matching *properties* `type` and `value`. Each standard HTML attribute has a corresponding property in the DOM node. But the DOM node also has additional properties, which don't have a corresponding attribute. For example: `childElementCount`, `innerHTML` or `textContent`.

The interpolation we had above to display the user's name:

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

is very similar to writing the following:

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

The `v-bind` directive allows you to modify the DOM property `textContent`, and we give it the value `user.name` which will be evaluated in the context of the current component instance, as it was for the interpolation.

The mechanic implemented for `v-bind:something="value"` in Vue is very simple:

- first it tries to set the property `something` if it exists
- otherwise it sets the attribute `something` to `value`, or removes it if the `value` is null.

Note that this mechanic is case-sensitive, so you have to write the property name with the correct case: `textcontent` or `TEXTCONTENT` will not work. It has to be `textContent`.

9.3.3. `v-bind` with an object

It is also possible to use `v-bind` without a parameter (no `:property`), and to give it an object directly. In that case, the directive binds all the keys of the object:

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

with a property named `buttonProperties`, which can have as many key/value as you want:

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

9.3.4. Classes and styles

It is fairly common in Web development to compute the CSS classes or styles that need to be applied

to an element. You can certainly do so manually in your component, for example, build a string `computedClasses`, containing all the classes you want to apply and then use `:class="computedClasses"` in your template. Or `:style="computedStyles"` with a string containing all the styles.

But `v-bind` is a bit smarter for these two common use-cases. You can use `:class` and `:style` with an object, to dynamically toggle the classes and styles you want to apply.

For example,

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

with the properties `isPrimary` and `isSmall`:

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

will add the class `btn-primary` to the button, and remove the class `btn-sm`.

Note that an alternative is to expose, directly from the component, an array of classes, or an object similar to the one we defined in the template:

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

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

Of course the same is possible with `:style`:

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

```
setup() {
  return {
    textStyle: ref({
```

```
      color: 'red',
      fontWeight: 'bold' as const
    })
  };
}
```

9.4. Events with `v-on`

If you're developing a webapp, you know that displaying things is just one part of the job: you also have to deal with user interactions. To allow this, the browser fires events, which you can listen to: `click`, `keyup`, `mousemove`, etc. Vue is here to help us!

Let's say we want to have a button that saves an action when the user clicks it.

Reacting to an event can be done by using the `v-on` directive as follows:

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

A click on the button of the example above will trigger a call to the component function `save()`.

Let's add this to our component:

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

As for data, Vue components expose functions to the template by returning them as properties in the `setup` function.

As for `v-bind`, most Vue developers prefer to use the shorthand syntax for `v-on:event`: `@event`.

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

This is what we'll use throughout this book.

The value passed to `v-on` can be a function call, but it can be any executable statement, like:

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

Note that `firstname` needs to be a property of your component.

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

However, I would not advise you to do this. Using functions is a better way of encapsulating the behavior: it makes your code easier to maintain and test, and it makes the template simpler.

The event can be emitted by the element itself or by one of its descendants. Vue will react to events that bubble up. Consider the template:

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

Even though the user clicks on the button embedded inside the div, the `save()` function will be called, because the event bubbles up.

You can give parameters to the function call of course. Oh, and you can access the event in the function called! You just have to pass `$event` to your function:

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

Then you can handle the event in your component function:

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

If you don't stop its propagation, the event will continue to bubble up, potentially triggering other event listeners up in the hierarchy.

You can use the event to prevent the default behavior and/or cancel propagation if you want, as shown above. Or you can use the `.stop` or `.prevent` directive modifier to do so.

9.4.1. `.prevent` and `.stop` modifiers

Instead of handling the event in the function, you can add a modifier to the `v-on` directive.

`.prevent` will prevent the default behavior. For example if you want to listen to the `submit` event of a `form` and not reload the page on submission:

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

Or you can stop the event propagation with `.stop`:

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

You can chain as many modifiers as you want:

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

There are a few other interesting modifiers for `v-on`, especially for the mouse and keyboard events.

9.4.2. Keyboard modifiers

One cool feature is that you can also easily handle keyboard events with a few modifiers you can add to the keyboard events, like `keyup`, `keydown`, `keypress`:

- `.esc` which activates only if the key is `esc`
- `.delete` which activates only if the key is `delete`
- `.space` which activates only if the key is `space`
- `.up` which activates only if the key is `up`
- `.down` which activates only if the key is `down`
- `.left` which activates only if the key is `left`
- `.right` which activates only if the key is `right`
- `.[whatever character you want]`, for example `.b` which activates only if the key is `b`

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

Every time you will press the `space` key, the `onSpacePress()` function will be called. And you can do a crazy combo, with: `.ctrl`, `.shift`, `.alt`, `.meta`, like `@keydown.ctrl.space`, etc. These modifiers are called *System modifiers*.

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

9.4.3. `.exact` modifier

If you want an event to be handled only if this is *exactly* this event, and not this event + another system modifier, you can add the `.exact` modifier.

```
<!-- `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. Mouse modifiers

You also have the `.left`, `.middle` and `.right` modifiers if you want to only listen to a specific mouse event.

For example:

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

9.4.5. `.once` modifier

If you want an event listener to execute only once, you can add the `.once` modifier:

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

9.4.6. `.passive` modifier

The `.passive` modifier is a bit different: it's here to tell the browser that the default behavior shouldn't be prevented, and that it should be done without waiting for the listener function to return.

It can be very handy if you want to do something on an event that happens very often, like `scroll` or `mousemove`, but don't want to slow down the scrolling of your application:

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

9.4.7. `.self` modifier

If you want an event listener to execute only when the event is from the element where you declared it, and not bubbling from one of its children, you can add the `.self` modifier:

```
<!-- 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` modifier

To be exhaustive, there is one last modifier, that we don't often use: `.capture`. It allows capturing an event bubbling from a child, and to handle it first in its handler before letting the child handling it itself:

```
<!-- 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 and TypeScript

As we mentioned earlier, there is no compilation error if you make a mistake in an expression of your template. You only have a runtime error. Volar can help a lot, as it analyzes your templates to let you know about potential errors! It relies on TypeScript to do so, and is not perfect, but it's the best you can do (we use it everywhere).

Note that Vue 3.2+ allows writing TypeScript directly in templates. It can be handy to hint to Volar that a variable is not null, or of a particular type:

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

9.6. Summary

The Vue templating system gives us a powerful declarative syntax to express the dynamic parts of our HTML. It allows expressing attribute and property binding, event binding and templating concerns, clearly, each with their own symbols:

- `{{}}` for interpolation
- `v-bind:property` or `:property` for attribute and property binding
- `v-on:event` or `@event` for event binding

It takes some time to be fluent in this syntax, but you will soon be up to speed, and then it's easy to read and write.

If you want to put this lesson in practice, check out the *Templates* exercise in our online training!

Try our quiz 🐴 and the exercise Templates 🐴 It's free and part of our online training (Pro Pack), where you'll learn how to build a complete application step by step. The exercise is all about building a small component, a responsive navbar, and play with its template.

Let's now talk about directives!

Chapter 10. Directives

Vue comes with directives, a powerful feature that we already used with `v-bind` and `v-on`. But there are some others, some really useful, and others which are good to know but less frequently used.

Let's start with the one you're going to use daily: `v-if`.

10.1. Conditions in templates with `v-if`

If you want to display part of the template only if a certain condition is true, you can use `v-if`:

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

The `div` itself is only displayed if the user is an admin.

You can also use it with the `v-else` directive, to display an alternative if the condition is false.

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

And you can even go one step further with one or several `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. Hide content with `v-show`

Another directive is slightly similar to `v-if`: `v-show`. `v-show` applies a `display: none;` CSS style to an element if the condition you give it is false. `v-if` also hides the content, but does more than that, as

it really removes it from the DOM, and re-creates it if necessary.

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

You can often use one or the other without noticing any difference.

That's not always the case though. For example, hiding an invalid form element with `v-show` won't make the form valid. Expressions evaluated in the hidden section might also throw errors when the condition is false. In such cases, using `v-if` is imperative.

There might be a slight performance difference between them, too:

- `v-if` really destroys the DOM and listeners, and re-creates them. So it has a slightly bigger cost when the condition changes, but when it is false, there is no DOM to watch for Vue. It is usually better for hiding vast part of a template, when the condition does not change really often.
- `v-show` only hides with CSS the elements, so Vue still watches them, but it is faster to hide/show, as it is just CSS.

Don't bother too much with these performance differences though. Use what is correct, and what feels fine.

10.3. Render only once with `v-once`

As we are touching the performance topic, the `v-once` directive can be useful. When applied to an element, the element and all its children will only render once, and then Vue will stop caring about updating this part.

It can be super useful for applications displaying big pages with components that display data that do not change:

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

10.4. Repeating elements with `v-for`

Working with real data will inevitably lead you to display a list of something. That's when `v-for` proves very useful: it allows instantiating one DOM element per item in a collection. Our `Races` component contains a property `races`, returned by the setup() function, which, as you can probably guess, is an array of races to display.

```
setup() {
  const races = ref([
```

```
    { id: 1, name: 'Lyon' },
    { id: 2, name: 'Amsterdam' }
  ]);
  return {
    races,
  };
}
```

This array can thus be displayed using `v-for`:

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

Note that you can use `in` or `of` as you prefer:

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

Now we have a beautiful list, with one `li` tag per race in our array!

If our collection gains a new item, Vue automatically adds a new `li`. Same thing if the `races` collection is re-ordered, or if one or several items are dropped.

Note that the good practice is to help Vue keep track of which item from the array corresponds to which DOM element. We'll come back to that, but you can add a `:key` binding to let Vue know which is which. Here, the ID of the race is a good candidate:

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

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

It is also possible to get the index of the current element:

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

The parentheses are optional, but I like having them. `index` receives the index of the current element, starting at zero.

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

Iterating on an array is by far the most common usage of `v-for`, but you can also iterate over the properties of an object:

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

where we define `podium` like this in the `setup` of our component:

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

You can also get the index of the property:

```
<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>
```

Be careful though: if the object comes from the server as JSON, there is no guarantee regarding the order of the properties in the object. So it's usually best to stick to arrays when the order matters.

You can also iterate on a range super easily:

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

Note that it begins at 1:

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

10.5. HTML content with `v-html`

If you want to insert dynamic HTML, you might be tempted to do:

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

with:

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

But that does not work and displays the raw HTML… We can *render* the HTMl by using the property `innerHTML`:

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

Or even more easily with `v-html`:

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

Vue does not interpret the content though, so you can't have interpolation or other directives inside. Be also super careful with what dynamic HTML you want to render: if you let your users enter it, you can have some security troubles.

10.6. Raw content with `v-pre`

Practically the opposite of `v-html`, `v-pre` allows displaying something you do *not* want Vue to interpret, for example an interpolation:

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

displays:

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

10.7. Other directives

There are three other directives. The first one is `v-cloak` which allows hiding a template while Vue is compiling it, but this is not useful if you use Vite or the CLI, as the templates are not displayed before compilation.

And we are going to see the two others, `v-model` and `v-slot`, in later chapters, as they are super powerful and deserve their own sections. Suspense, suspense…

In the meantime, we have an exercise for you!

Try our third exercise List of races 🐴! It is free and part of our Pro Pack, where you'll learn how to build a complete application step by step. The exercise guides you to build another component: the list of races. Smells like a good use-case for `v-for`!

Chapter 11. How to build components

There are many ways to build a component since Vue 3. The book uses the new Composition API introduced by Vue 3, whereas you may be used to the Options API used in Vue 1 and 2.

Up until now, we mainly focused on how to write the templates, without caring too much about how to write the component itself. But we already mentioned that if we want Vue to update the template of a component, then we have to declare a reactive property.

The core principle of a Vue application is to have a declarative template, displaying data exposed by the component, and refreshing the view every time the data changes. In order for Vue to be able to implement this magic, it needs to be aware of the changes made to the values displayed in the template. Some frameworks, like Angular, do that by detecting changes every time an event occurs, by comparing the new values of the expressions used in the template with the previous ones. Vue uses another strategy. It uses what it calls "reactive properties".

Reactive properties are properties which can signal that they have been modified. By making properties reactive, you make it possible for Vue to be aware of the changes you make to the properties, and thus to update the DOM when and where necessary.

Let's start by digging into that.

11.1. Reactive property with `ref`

Vue 3 introduced a function called `ref` that you can use to declare a reactive property. Vue will then update the template every time the `value` of the `ref` property changes without any action on your part. You can use the `ref` property directly in the template: Vue unwraps the value for you:

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
    setTimeout(() => (numberOfUsers.value = 147), 3000);
    return { numberOfUsers };
```

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

This is where TypeScript and its implicit typing shine: `numberOfUsers` is of type `Ref<number>`. So if you try to write `numberOfUsers = 147` or `numberOfUsers.value = 'hello'`, TypeScript will nicely tell you that it does not make sense .

11.2. Reactive property with `reactive`

`ref` is really handy when working with primitives, as there is no other way to tell Vue that a primitive is reactive. But when working with objects, it can be a bit cumbersome to always write `.value.property` to update one property of the underlying object:

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 };
  }
});
```

That's where `reactive()` can help! It is very similar to `ref` in the sense that it declares a property as reactive, and Vue will watch the changes and refresh the template accordingly.

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 };
  }
});
```

As you can see, no need for `.value` here when we want to update a property, and you can still use `{{ pony.name }}` in the template!

The cool thing is that works with all the nested properties of the object!

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 };
  }
});
```

It also works with collections and their elements, or when you add and delete property to an object (which was one of the limitations of Vue 2.x, we'll talk about how this can work in Vue 3 in a later chapter).

As it is quite handy, you may be tempted to encapsulate your primitives into an object to get rid of `ref()` and `.value`, and use `reactive()` all the time. You definitely can. Let's say we built a `Product` component. Instead of:

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>
```

you can use a wrapper object and `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>
```

You may be tempted to destructure the `state` object to directly use `price` and `quantity` in the template instead of `state.price` and `state.quantity`. But this breaks the reactivity, so the template will not reflect any update of `state`.

So this fails:

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>
```

But all is not lost, as Vue offers a `toRefs` function that you can use to transform a reactive object into references:

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` or `reactive`

So which one to pick? As they are pretty similar, in most cases, it is a matter of taste. Something that helped us understand is:

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

A `ref` to an object is pretty much the same as a `reactive` object containing a `value` with this object. If that's still a bit hard to wrap your head around, don't worry. It took us ages to figure this out 🤯.

Check out the following examples, they may help.

11.3.1. With primitives

No need to think twice, you have to use a `ref`:

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

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

11.3.2. With arrays

Arrays can work with both, but can only be reassigned when using a `ref`.

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

With `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. With objects

It's very similar to what we did with arrays.

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

With `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
```

As you can see, you can't really be wrong. In the future, we may have an official recommendation, but for now, pick what you prefer 😉.

11.4. Derive state with `computed`

You will sometimes need to re-compute a value every time another one changes. For example, in the `Product` component I used above, it would make sense to compute the `total`, every time the `price` or `quantity` changes.

This is super easy in Vue with the `computed()` function. `computed` takes a getter function and returns a (readonly) reference:

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>
```

And this works right away: Vue detects the reactive properties that `total` depends on, and tracks them. Every time one of the dependencies updates, Vue runs the getter function again to get the new value. It only works with reactive properties: if `state` was not a reactive object, but just an object, then the *computed property* would never be reevaluated.

Computed properties are evaluated lazily: if a computed property is never displayed or use elsewhere, then Vue does not even recompute its value.

11.5. Execute a side effect with `watchEffect` and `watch`

Sometimes, it's necessary to know when the value of a property changes, but not necessarily to compute the value of another one.

For example, we may want to produce a side effect every time a property changes.

The typical example is a search UI, where you need to ask the server for new results every time the user enters a new query. Or if we go back to our `Product` component, maybe we want to keep track of every change of the total value.

That's what the function `watchEffect` allows doing:

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}`);
  }
);
```

As you can see, we can just define an effect to run, and Vue automatically tracks its dependencies, and re-runs the effect when a dependency changes.

We can access the new value and the old one if we need to, by using `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 }
);
```

You can also watch an array of reactive values, in order to produce a side effect when any of the values changes. Note that, by default, `watch` is lazy and won't run immediately, whereas `watchEffect` always runs eagerly (as Vue needs to run the effect to find the reactive dependencies it needs to watch, whereas we manually define them with `watch`).

`watch` is also handy if you want to run the effect only for certain changes, and not when any of the reactive dependencies changes (as it does with `watchEffect`).

If you want, you can stop the watcher by calling the returned function:

Product.vue

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

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

You can even ask the watcher to do some cleanup when stopping it:

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. Passing `props` to components

Let's say we want to build a `Race` component, displaying the ponies engaged in the race.

Such a component can look like:

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>
</template>

<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>
```

The template is quickly becoming a bunch of copy/pasted HTML…

So if we need to display the `color` of a pony in some other way, then we need to change it 3 times. In a more complex application, you can see how this can become problematic.

You might say that we should use an array of ponies, and use `v-for` to avoid the duplications, and you would be right. But what if we need to show the winning pony at the top of the template, in a separate section? `v-for` can't help us anymore here. So we need a solution to keep our code DRY.

But we saw that we can use a component inside another one: this is usually the way to go to keep components small enough and have components that you can easily reuse. Here it's becoming quite obvious that a `Pony` component would greatly help us. Once created, we would simply need to use a `<Pony />` element in our `Race` template each time we want to display a pony, and thus avoid the copy-paste mess:

Race.vue

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

But how do we tell each Pony what name and color to display?

That's where `props` come into play!

In a component, you can define `props` in order to be able to receive data from the parent component. These `props` become part of the state of the component.

So our `Pony` component should look like:

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>
```

As you can see, our component object has a `props` object, where we list what the component accepts. For each prop, you specify the type it accepts. Awkwardly, you can't use TypeScript types here: this API has been around for a while, before TypeScript support, and only supports JavaScript "types": `String`, `Number`, `Boolean`, `Date`, `Function`, `Symbol`, `Object` or `Array`. We'll see later that we can do slightly better.

At runtime, Vue checks the values given to the prop: if a value does not match with the expected type, we have a nice warning in the console like:

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

If you check our `Pony` component, you'll see that we can use the props directly in the template.

Now, how do we give values to these props? Back into our `Race` component, we can use `<Pony name="Rainbow Dash" color="BLUE" />` if we have static values, or, as we have dynamic ones:

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>
```

If you want to pass a number, a boolean, an array, etc. you *must* use `v-bind:`, even for a static data. Otherwise, Vue will pass the string value: `<Pony speed="16" isRunning="false" />` would pass `"16"` as the speed and `"false"` as the `isRunning` flag, not `16` and `false`. Note that for a `true` boolean value, you can also simply use `<Pony isRunning />`

This is nicer, but we can do slightly better. Instead of passing every property of our ponies to the `Pony` component, we should pass the whole object:

Race.vue

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

and have only one prop in our `Pony` component:

Pony.vue

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

export default defineComponent({
  name: 'Pony',

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

It looks good, but we are loosing typing information: we *know* that `ponyModel` is not any object, it's an entity of our application, defined in a `PonyModel` interface:

PonyModel.ts

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

We can use `PropType<T>` to properly type our prop:

Pony.vue

```
import { defineComponent, PropType } from 'vue';
import { PonyModel } from '@/models/PonyModel';

export default defineComponent({
  name: 'Pony',

  props: {
    ponyModel: Object as PropType<PonyModel>
  }
```

```
});
```

Now we properly typed our prop, which is very handy if we need to use it in our component code, as we'll see later.

11.6.1. Required props

Vue also provides a way to define that a prop is mandatory with `required`:

Pony.vue

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

The declaration of the prop takes a longer form: instead of `prop: type` we now use `prop: { type: type, required: true }`. By marking the prop as required, we ask Vue to perform a check at runtime when the component is used, and to throw an error to warn us if we forgot to give a required prop. The warning should look like:

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

11.6.2. Props with default values

We can also give a default value to a prop:

Pony.vue

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

As my prop is an object, I need to give a factory as the `default` value: it would be simple `default: 42` if it was a number for example.

11.6.3. Props with validators

Last but not least, you can define a custom validator for the prop. For example, if we have a `speed`

prop, and we want to make sure it is greater than 5:

Pony.vue

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

At runtime, Vue validates the prop and warns us if the validation fails:

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

Since Vue v3.4, the `validator` function is now called with a second argument containing all the props, allowing to validate the value against other props:

Pony.vue

```
min: {
  type: Number,
  required: true,
  validator: (value: number) => value >= 0
},
max: {
  type: Number,
  required: true,
  validator: (value: number, props: { min: number }) => value >= props.min
}
```

All these runtime checks (types, required, validation) are omitted when running the application in production mode.

11.6.4. Using props in the `setup` function

We saw how to use the props in the template, but how can we use them in the `setup()` function we saw earlier?

Well, it turns out the first parameter of the `setup()` function can be `props`! And we can then use our props like any reactive property:

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 };
}
```

Here we define a `ponyImageUrl` computed property based on the `color` of the `ponyModel` prop: every time the prop changes, the image URL will change too.

And as we properly typed our props, the TypeScript compiler will warn us if we misuse our `PonyModel` entity. 🕶

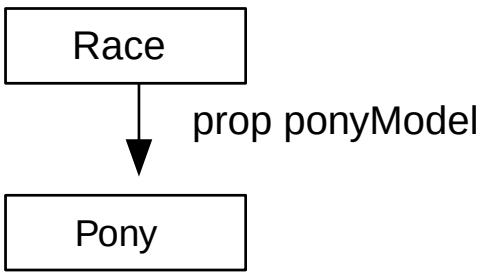
Note that props are read-only: you can't modify them inside the component receiving the prop (but the parent component can, of course). So what can we do if we want to update the prop given to us by a parent component? We use an event!

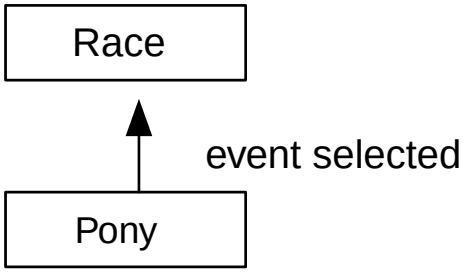
Try our exercises Race details 🐴 and Computed properties 🐴! These exercises will guide you to build a more advanced component, with props and computed properties.

11.7. Custom events with `emit`

The `props` object can't be mutated in the child component. You *can* mutate the content of a prop if it is an object or an array, but this is considered a bad practice. When a component wants to notify its parent component that something has to be changed, however, it can emit an event. The parent can listen to this event and update its state accordingly.

For example, our `Pony` component can receive a `ponyModel` prop, and can emit a `selected` event.





How do we emit such an event? It turns out that `setup(props)` can have a second parameter, the `context`: `setup(props, context)`. `context` is an object containing several things, and we'll see some of them later. But right now, we're only interested in the `emit` function it contains, so we can use destructuring:

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>
```

The `emits` property is optional, but it's nice to have. At compile time, TypeScript will check that you only emit the events that are declared in the `emits` property, and that the events you emit have the appropriate type. In this example, the compilation would thus fail if you emit an event that isn't named `selected`, or if the event is not a `PonyModel`. At runtime, during development only, Vue also checks that the given predicate accepts the emitted events. If it doesn't, the event will still be emitted, but you'll get a warning in the console. In this example, you'll thus get a warning if you emit a purple pony. If you don't want to validate the event value, you can just return `true` in the predicate function.

As you can see, `emit` allows to... emit an event when we want to, with the name and the payload we want. Then we can listen to this event in the template of the parent component, and get its payload by using `$event` if we need to as we do with standard DOM events.

So here we can listen to the `selected` event in `Race`, and change the color of the pony for example:

Race.vue

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

with `changeColor` a function that assigns a random color to the pony:

Race.vue

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

Now when a user clicks on the `Pony` component, it emits a custom `selected` event, that the `Race` component catches and calls `changeColor()`. The color of the selected pony changes, and as it is a prop of our `Pony` component, the component refreshes and displays the pony in its shiny new color!

Try our exercise Pony component 🦄 to build a new component with a custom event.

11.8. Lifecycle functions

Very often, you need to do some work when your component is initialized. Vue calls the `setup` function right when it creates the component as we saw. It contains most of the setup process. So how can we call an API to populate a property after displaying the template for example? That's

where lifecycle functions are useful. You can use them inside the `setup` function, and they all work the same: they accept a function (that can be an `async` one) that will be executed by Vue at a specific time. We'll see later that the `setup` function itself can be async in a specific case.

There are several of them that you can use, each corresponding to a particular instant of the component's life. The main ones are:

- `onBeforeMount`/`onMounted`, called before/after the component has been mounted (inserted in the DOM); (there's probably a joke about "mount the pony", but let's stay focus)
- `onBeforeUpdate`/`onUpdated`, called before/after the component has been updated; (something changed, and the DOM re-rendered)
- `onBeforeUnmount`/`onUnmounted`, called before/after the component has been destroyed; (removed from the DOM)
- `onErrorCaptured` called if there is an error in one of the descendent components.

Let's see a few examples.

11.8.1. Initialize your component with `onMounted()`

A very frequent usage of `onMounted` is to call an asynchronous API to initialize one of the reactive property of the component. If `fetchRace()` is an asynchronous function (it is, as we are asking the backend via an HTTP call), then we can write:

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 };
}
```

It can also be useful if you need to interact with the created DOM, as we'll see later.

11.8.2. Cleanup with `onUnmounted()`

A fairly common source of memory leaks in applications is to forget to stop background tasks.

Let's say that we want the `Race` component to refresh the race displayed every 10 seconds. We could write:

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 };
}
```

But this is buggy: every time the component is instantiated, it starts a background process that never stops...

So after a while you may end up with a ton of calls to your API, the server would be brought down to its knees, and the browser would consume more memory and bandwidth than necessary. We can stop the process by using `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 };
}
```

Try our quiz 🦄 to check if you got everything!

Chapter 12. Style your components

Let's stop to talk about styles and CSS for a minute. I know right? Why talk about CSS?

Well because Vue (and Vite or Vue CLI) can do a lot of things for us behind the scenes.

As a Web developer, you often add CSS classes to elements. And the essence of CSS is that it will *cascade*. That's sometimes what you want (to change the font everywhere in your app for example), and sometimes not. Imagine you want to add a style on a selected element in a list: you will usually use a very narrow CSS selector in your CSS, like `li.selected`. Or an even narrower one, using conventions like BEM, because you just want to style the selected element in a specific part of your app.

By default, when you add styles to your component (with the `style` part of an SFC), the styles are global, and applied to the complete application.

That's where Vite or Vue CLI can be useful. The styles can be applied to this component and only this one. How does it achieve this?

12.1. Scoped styles

It starts with you writing some styles. We'll take a component you're starting to know well, i.e. our `Pony` component. This is a really simple version of the component, only displaying the pony's image and its name in a `span` element. For the purpose of the example, we add a CSS class `highlight` to this `span`:

```
<div>
  <img :src="ponyModel.imageUrl" :alt="ponyModel.name" />
  <span class="highlight">{{ ponyModel.name }}</span>
</div>
```

This class is defined in the styles of the component, but note that we added `scoped` to the `style` element:

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

As you can see, we want to display the pony's name in red.

To make sure our `highlight` class is only applied to the `Pony` component, Vue takes the CSS defined for the component, and inlines it inside the `<head>` element of the page. But before inlining it, it rewrites the CSS selector, to append a unique attribute identifier. This unique attribute is then added to all the elements of our component's template! That way the style will only apply to our component:

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

The `highlight` class selector has been rewritten to `.highlight[data-v-0a17a18c]`, so it will only apply on elements that have both the class `highlight` and the attribute `data-v-0a17a18c`. You can see that this attribute has also been added to our `span` automatically, so that works perfectly.

This feature is very handy. It allows us, for example, to use components from a third-party library, without fearing that our CSS classes clash with theirs!

12.2. Module styles

Vue also supports `module` styles. This feature is based on the CSS Modules specification and allows using `$style` in your template and in your code! This is a fairly popular solution in some ecosystems (like React), but I've never used it 😬.

12.3. `v-bind` in CSS

You can also use `v-bind` directly in the CSS to bind a CSS value to a property of your component:

```
<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>
```

This is of course reactive: if `ponyModel.color` changes, then the color updates.

12.4. `v-deep`, `v-global` and `v-slotted`

SFC scoped style offers some custom CSS extensions for specific edge cases.

Sometimes you want to explicitly style a child component. As a `scoped` style is only applied to the current component, how can we do that?

Enters `::v-deep()`, or `:deep()` as a shorthand alternative:

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

On the opposite, if you want a rule to be global, you can use `::v-global()` or `:global()`:

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

The last extension allows targeting slot content. In Vue 3, the slot content is not affected by the scoped styles. If you want to explicitly target a slot content, you can use `::v-slotted()` or `:slotted()`:

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

12.5. PostCSS

Vite and Vue CLI use PostCSS under the hood to transform our CSS.

But they slightly differ here: Vite considers you are targeting modern browsers, so it doesn't do much out-of-the-box. But you can provide a PostCSS configuration if you need to, and it will be automatically applied.

12.6. CSS Pre-processors

CSS is getting better and better. It is now possible to use CSS variables for example. That's why Vite recommends to stick to it.

But if you want to use more than raw CSS, then Vite and the CLI also have a built-in support for the

most common pre-processors:

- SCSS
- Less
- Stylus

You just have to install the pre-processor you want to use (and its Webpack loader if you're using the CLI):

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

You can then set the `lang` attribute of the `style` element:

```
<style lang="scss">

</style>
```

And the build will do the rest.

To sum up, Vite and the CLI are very helpful for the style part an application. You probably want to use `scoped` styles most of the time, and Vue will do the heavy lifting 🏋!

Chapter 13. Composition API

We already introduced the Composition API in the previous chapter in the context of building components.

Let's first talk about how the Composition API can help us have a clean design. Then we'll see how we can share code between components.

13.1. Clean design with the Composition API

Let's imagine a component `Order` where a user can order a product with a certain quantity. We want to display the product information and the total price. We also want to let the user increase or decrease the quantity. And finally, we want to track when the user enters and exits the component, for analytics purpose.

Such a component can look like:

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 };
}
```

This is fairly understandable, but the different concerns of the component are mixed with each other.

It would have been even worse in Vue 2.x. As you may know, the Composition API has been introduced in Vue 3.0. Previously, you had to use (and you can still use) the Options API with several fields to declare:

- the reactive properties in `data`;
- the methods in `methods`;
- the watchers in `watch`;
- the computed properties in `computed`;
- the lifecycle hooks each in a dedicated field like `mounted`, `destroyed`, etc.

So for a big component, you ended up with the logic split across the different definition fields.

One of the interesting aspects of the Composition API is that all the code related to a given concern can be written in a single place, separated from the code related to other concerns. And we can even extract some shared component logic *outside* of components.

Let's refactor our `Order` component, by extracting the related business logic in dedicated functions.

Vue has a convention that you can follow: implement a function named `use⋯` returning an object with the different reactive properties and functions. We'll see in later chapters that a lot of libraries use that convention. The Vue router, for example, exposes a `useRouter` function.

So here we could extract a `useTracking` function, outside our component, dedicated to tracking our users:

Order.vue

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

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

A `useProduct` function, dedicated to the business logic related to the product:

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 getProduct(productId.value);
  });
  return product;
}
```

And finally a `useOrderService` function, related to all the actions for ordering:

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 };
}
```

Now that's done, our component is better looking:

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 };
}
```

It's more concise, and we know where to look when we want to modify something. What's even nicer is that these `use···` functions, that we often call *composables*, can be shared across components!

13.2. Extracting common logic in `use···`

Fairly often in an application, you'll end up with components that need to share the same logic, like calling the same HTTP API.

Instead of copy/pasting the code, we can extract the common logic in a separate file. For example if several components need to do some user-related business, like `authenticate`, `logout`, `register`, etc... we can write these functions in a dedicated file.

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

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

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

// ...
```

To take advantage of the composition API, we should however return these functions as part of an object, returned by a `useUserService` function:

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
  };
}
```

You can then use the *composable* in your components as follows:

Login.vue

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

13.3. Composition API outside of components

Until now, wrapping the user-related functions in an object returned by `useUserService()` doesn't bring much, because the object doesn't have any state, nor any component-related logic. But it would be interesting to return the logged-in user as well, as some components may want to display her/his information.

As the Composition API is not bound to components, we can directly use it in our `useUserService` function:

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
  };
}
```

Now every component that needs it can access the logged-in user and use it in its template directly as it is a reactive value!

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>
```

To come back to our analytics use-case, let's say that you want to trace the activity of your users every time they enter or exit a view.

You can manually handle that with `onMounted` and `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(async () => await trace('home:enter'));
    onUnmounted(async () => await trace('home:exit'));
    return { userModel };
  }
});
</script>
```

with `trace` looking like:

UserService.ts

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

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

But it can be cumbersome to do so in *every* component. We can do something better: move the `onMounted` and `onUnmounted` calls to `useUserService`:

UserService.ts

```
export function useUserService(view?: string) {
  if (view) {
    onMounted(async () => await trace(`${view}:enter`));
    onUnmounted(async () => await trace(`${view}:exit`));
  }
  return {
    userModel,
    authenticate,
    logout
  };
}
```

Now the components can simply call `useUserService`, and the `trace` calls are done automatically! How cool is that?

Home.vue

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

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

To sum up, you can extract logic in a `use⋯` function returning an object, exposing functions (synchronous or asynchronous) and even reactive references with `ref` or `reactive`. The `use⋯` function can also add watchers, computed properties or lifecycle hooks. These functions can be used to simply group related logic together and simplify a component, or can be shared between components, to avoid copy/pasting, and even to let them easily communicate between them.

Try the exercise Lifecycle hooks 🦄 to create your first `use⋯` function and use the `onMounted` lifecycle hook.

13.4. A community example: VueUse

The great thing with this pattern is that these *composables* can be extracted and re-used. VueUse is an open-source library from the Vue community which collects a *ton* of utility functions.

Here are some of my favorites:

- `useTitle`: to change the title of the page
- `useGeolocation`: to get the location of your user
- `useQRCode`: to generate QR codes
- `useBreakpoints`: to be notified of a change of the viewport in your code (similar to a media query in CSS)
- `useClipboard`: to read or write some content to the clipboard easily

VueUse.vue

```
<template>
  <div>{{ coords.latitude }} - {{ coords.longitude }}</div>
  <img :src="qrcode" alt="Become a Ninja with Vue 3" />
</template>

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

There are also integrations for firebase, axios, etc. A lot of other tiny little gems are available, think about it next time you need something!

Chapter 14. The many ways to define components

As we were saying in the previous chapter, there are many ways to build a component since Vue 3.

The book uses the new Composition API introduced by Vue 3, whereas you may be used to the Options API used in Vue 1 and 2, which is still usable in Vue 3. Vue 3 also offers another approach, heavily inspired by the Svelte framework. Let's review these alternatives.

14.1. Options API

This is the historical way to build Vue components. You have an object, in which you can define reactive data with `data`, methods with `methods`, computed data with `computed`, etc. This is totally fine and still supported in Vue 3, but is mainly around for compatibility purpose. It does not offer the same flexibility as the Composition API, forcing developers to use Mixins to share behavior.

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. Composition API

Vue 3 introduced the Composition API after an intense discussion and feedback from the community. The idea is to provide a flexible API with a top-notch TypeScript experience. This API concentrates most work in the `setup` property of the component definition.

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 introduced another way to declare your component, based on the Composition API, but slightly less verbose, by adding a `setup` attribute to the `script` element of your 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>
```

As you can see, this syntax removes a lot of boilerplate, as everything in the `script` section is automatically exposed to the template. It draws its inspiration from the Svelte framework, and is currently the preferred way to define components. This is what we'll use throughout the rest of the book and of the exercises. We'll dive into this in the next chapter! Let's finish our tour for the moment.

14.4. Class API

Vue 2 had a class API to declare components, which used to be the best way to use TypeScript, and this is still supported in Vue 3. But the tooling doesn't support this as well as the other options, as the recommended way to use TypeScript is the Composition API with `script setup` nowadays.

This API was even considered to be the default one in Vue 3, but after a long discussion within the community, the Composition API was preferred.

Chapter 15. `script setup`

Vue 3.2 introduced the `script setup` syntax, a slightly less verbose way to declare a component. You enable it by adding a `setup` attribute to the `script` element of your SFC, and you can then remove a bit of boilerplate in your component.

Let's take a practical example, and migrate it to this syntax!

15.1. Migrate a component

The following `Pony` component has two props (the `ponyModel` to display, and a `isRunning` flag). Based on these two props, a URL is computed for the image of the pony displayed in the template (via another `CustomImage` component). The component also emits a `selected` event when the user clicks on it.

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>
```

As a first step, add the `setup` attribute to the `script` element. Then, we just need to keep the content of the `setup` function: all the boilerplate can go away. You can remove the `defineComponent` and `setup` functions inside `script`:

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. Implicit return

We can also remove the `return` at the end: all the top-level bindings declared inside a `script setup` (and all imports) are automatically available in the template. So here `ponyImageUrl` and `clicked` are available without needing to return them.

This is the same for the `components` declaration! Importing the `CustomImage` component is enough, and Vue understands that it is used in the template: we can remove the `components` declaration.

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');
    }
```

We're nearly there: we now need to migrate the `props` and `emits` declarations.

15.3. `defineProps`

Vue offers a `defineProps` helper that you can use to define your props. It's a compile-time helper (a macro), so you don't need to import it in your code: Vue automatically understands it when it compiles the component.

`defineProps` returns the props:

Pony.vue

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

`defineProps` receives the former `props` declaration as a parameter. But we can do even better for TypeScript users!

`defineProps` is generically typed: you can call it without a parameter, but specify an interface as the "shape" of the props. No more horrible `Object as PropType<Something>` to write! We can use proper TypeScript types, and add `?` to mark a prop as not required 😮.

Pony.vue

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

We lost a bit of information though. In the previous version, we could specify that `isRunning` had a default value of `false`. To have the same behavior, we can use the `withDefaults` helper:

Pony.vue

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

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

Or even simpler, since Vue 3.2.20 and its `reactivityTransform` option, we can destructure and give a default value to a prop directly:

Pony.vue

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

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

This allows to use the props directly, without the need of `props.`:

Pony.vue

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

The last remaining syntax to migrate is the `emits` declaration.

15.4. defineEmits

Vue offers a `defineEmits` helper, very similar to the `defineProps` helper. `defineEmits` returns the `emit` function:

Pony.vue

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

Or even better, with TypeScript:

Pony.vue

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

The full component declaration is 10 lines shorter. Not a bad reduction for a ~30 lines component! It's easier to read, and plays better with TypeScript. It does feel a bit weird to have everything automatically exposed to the template, without writing return though, but you get used to it.

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 introduced a new macro called `defineOptions`. It can be handy to declare a few things like the name of a component, if the inferred name based on the file name is not good enough or to set the `inheritAttrs` option:

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

15.6. Closed by default and `defineExpose`

There is a more subtle difference between the two ways to declare components: a `script setup` component is "closed by default". This means other components don't see what's defined inside the component.

For example, the `Pony` component can access the `CustomImage` component (by using refs, as we'll see in a following chapter). If `CustomImage` is defined with `defineComponent`, then everything returned by the `setup` function is also visible for the parent component (`Pony`). If `CustomImage` is defined with `script setup`, then *nothing* is visible for the parent component. `CustomImage` can pick what is exposed by adding a `defineExpose({ key: value })` helper. Then the exposed `value` will be accessible as `key`.

This syntax is now the recommended way to declare your components. The rest of this book, and the exercises of our online training, use it.

Try the exercise Script setup 🦄 to migrate our application to this syntax.

Chapter 16. Testing your app

16.1. The problem with troubleshooting is that trouble shoots back

I love automated testing. My professional life revolves around the test progress bar going green in my IDE, patting me in the back for doing my job properly. And I hope you do care about tests too, as they are the only safety net we have when we write code. Nothing is more tedious than manually testing code.

Vue does a great job to let us easily write tests, with the help of a few tools.

We can write different types of tests:

- unit tests
- end-to-end tests

The first ones are there to verify that a small unit of code (a component, a service…) works correctly in isolation, i.e. without considering its dependencies. Writing such a unit test requires you to execute each of the component or service functions, and check the outputs are what we expected regarding the inputs we fed it. We can also check the dependencies used by this unit are correctly called: for example we can check that a service will do the correct HTTP request.

We can also write end-to-end tests. Their purpose is to emulate a real user interacting with your app, by starting a real instance and then driving the browser to enter values in inputs, click on buttons, etc. We'll then check the rendered page is in the state we expect, that the URL is correct - whatever you can think of.

We're going to cover all this, but let's begin with the unit test part.

16.2. Unit test

As we saw earlier, unit tests are there to check a small unit of code in isolation. These tests can only verify that a small part of your app works as intended, but they have several advantages:

- they are really fast - you can run several hundreds of them in a few seconds.
- they are a very efficient way to test (nearly) all your code, especially the tricky cases, which can be hard to manually test in the real app.

One of the core concepts of unit testing is **isolation**: we don't want our test to be biased by the unit's dependencies. So we usually use "mock" objects as dependencies. These are fake objects that we create just for testing purposes.

To do this, we are going to rely on a few tools. First we need a library to write tests. One of the most popular (if not the most popular), is Jest, but we are going to use my favorite, Vitest!

16.3. Vitest

Vitest is a really cool solution: it's really similar to Jest (same API, based on jsdom) but faster and more modern.

If you generate your project with `create-vue` instead of Vue CLI, this is the unit testing solution you'll get. Our pro pack exercises all use Vitest of course: you'll find all kinds of tricks (and configuration tips) to test everything in your projects!

Vitest has been built to leverage Vite to handle all the transformations when running the unit tests. So you don't need `ts-jest` or `@vue/vue3-jest`. And Vitest works great with ECMAScript Modules, whereas Jest… not so much.

Vitest gives us a few functions to declare our tests:

- `describe()` declares a test suite (a group of tests)
- `test()` declares a test
- `expect()` declares an assertion

All the configuration options goes into either a `vitest.config.ts` file, or directly inside the Vite configuration, in a `test` property.

A basic JavaScript test for a simple class:

Pony.ts

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

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

using Vitest looks like:

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);
  });
});
```

The `expect()` call can be chained with a lot of methods, named *matchers*, like `toBe()`, `toBeLessThan()`, `toBeUndefined()`, etc. Every method can be negated with the `not` property of the object returned by `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);
  });
});
```

The test file is a separate file from the code you want to test, usually with an extension like `.spec.ts`. The test for a Pony component written in a `Pony.vue` file will likely be in a file named `Pony.spec.ts`. Where to put the `.spec.ts` files is a matter of preferences: some people like to put them next to the file being tested in a `__tests__` directory, others prefer putting them in a separate, dedicated directory. I tend to do what `create-vue` does by default, and use a dedicated `__tests__` directory.

Often, all the tests written by calling `test()` need a common context to be prepared before the test. You can use the `beforeEach()` function to set up this common context before each test. If I have several tests on the same pony, it makes sense to use `beforeEach()` to initialize the pony, instead of copy/pasting the same thing in every test.

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);
```

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

There is also an `afterEach` function, but I rarely use it…

One last trick: Vitest lets us create fake objects (mocks or spies, as you want), or even spy on a method of a real object. We can then do some assertions on these methods, like with `toHaveBeenCalled()` that checks if the method has been called, or with `toHaveBeenCalledWith()` that checks the exact parameters of the call to the spied method.

For example, let's say we have a `Race` class with a `start()` method, that calls `run()` on every pony in the race, and returns the ponies that accepted to run (ponies can be stubborn: `run()` returns a boolean):

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))
    );
  }
}
```

We want to test the `start()` method, and see if it properly calls `run()`. So we spy on the `run()` method of all the ponies in the race:

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]);
```

```
  });

});
```

and test if the methods are called:

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]);
});
```

When you write unit tests, keep in mind that they should be small and readable. And don't forget to make them fail at first, to be sure you're testing the right thing.

The next step is to run our tests. Vitest executes the tests inside jsdom, a browser-like environment that runs in Node.js. `create-vue` offers the `npm run test:unit` command to run the unit tests. It watches your files to re-run the tests on every save.

As running the tests is really fast, it's actually really nice to do this and have (almost) instant feedback on your code. You can also filter the tests you need to run if you want to.

I won't dive into the details on how to set up Vitest, but it's a very interesting project with a lot of plugins you can use, to have a coverage report, etc. If you're writing your code in TypeScript like me, it comes with a nice built-in support.

So we now know how to write a unit test in JavaScript. Let's add Vue to the mix.

16.4. `@vue/test-utils`

Vue has an official testing library called `@vue/test-utils`.

It is super handy to unit test Vue components, as you can *mount* a component, and then test not only its functions and state, but also its template and events.

Let's say we have built a nice `Pony` component. It has a prop, and it displays an image depending on its color. It also emits an event when the user clicks on it:

Pony.vue

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

```
    <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>
```

We want to test that the template is displaying an image with the correct source. vue-test-utils offers a `mount` function to create an instance of the component:

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

It returns a `wrapper`, an object that offers several interesting properties and methods.

For example, it offers a `vm` property, which represents the instance of the component, on which you can check the value of the various fields of the component:

Pony.spec.ts

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

It also offers the `get`, `find` and `findAll` methods, to check elements in the template:

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');
```

Our first component test finally looks like:

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');
});
```

Similar functions (`getComponent`, `findComponent`, `findAllComponents`) can also be used to find Vue

subcomponents used by the template, for example `wrapper.getComponent(CustomImage)` if `CustomImage` was a component of our application.

Now we need to test the event emission. To do so, vue-test-utils offers a `trigger` method to... trigger an event:

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();
});
```

If you want to check the DOM after changing the state, or after triggering an event, you have to wait for the DOM to be updated. You can either:

- call `nextTick()` after the action to refresh the DOM;
- directly `await` the action like in the following example (which calls `nextTick()` for you behind the scenes).

Pony.spec.ts

```
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');
});
```

As you can see, you can use the `wrapper.setProps` method to update the props, and await it. You can do the same with `trigger`.

`@vue/test-utils` also offers mounting options in addition to `props` Among them the most useful are:

- the `stubs` option to mount a component, and mock some of its subcomponents;
- the `plugins` option to add a plugin, like the router, to the test.

You have example of usage of all these functions in our online exercices of course.

16.5. Snapshot testing

Another way to test your templates is to use snapshot testing. Vitest can create a snapshot of the DOM created (in a pretty format) and store it in a file along your tests, so it can be reviewed.

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();
});
```

When you execute the test for the first time, the test won't actually verify anything, but will generate the snapshot. You can then check if it contains the expected DOM structure (and fix the code until it does).

Once you're happy with the result, you can save the snapshot in your VCS system (it becomes part of your sources), so that every subsequent test execution verifies that the DOM generated by the test still matches the saved snapshot.

If you use this feature, you'll see a new `snapshots` directory appear, containing the test snapshot:

Pony.spec.ts.snap

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

exports[`Pony.vue > should match the snapshot 1`] = `
<figure>
  <img
    alt="Fast Rainbow"
    src="/images/pony-purple.gif"
  />
  <figcaption>
    Fast Rainbow
  </figcaption>
</figure>
`;
```

Then if you update your template, the test fails and shows you what changed in the snapshot. Here I added an exclamation point after the name of the pony:

```
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
```

You can then spot you have an issue and fix it, or accept the changes if you want to update the

snapshot with the new result.

This can be a nice tool, but I'm not a huge fan of this practice, as developers sometimes blindly accept the changes without really paying attention. I consider them more like a complement for conventional tests than a replacement. They are nice to have and super easy to write though!

16.6. End-to-end tests (e2e)

End-to-end tests are the other type of tests we can run. An end-to-end test consists in really launching your app in a browser and emulating a user interacting with it (clicking on buttons, filling forms, etc.). They have the advantage of really testing the application as a whole, but:

- they are slower (several seconds per test)
- it can be hard to test the edge cases.

As you may guess, you don't have to choose between unit tests and e2e tests: you will combine both to have great coverage, and some guarantees that your complete application runs as intended.

Vue CLI offers integration with two e2e testing libraries: Cypress and Nightwatch.js, and create-vue offers support for Cypress out-of-the-box as well. And I am madly in love with Cypress 😍.

You can run your e2e tests with:

```
npm run test:e2e
```

16.7. Cypress

When you execute your tests, it launches either Electron, Chrome, Firefox or Edge. Or you can launch them in *headless* mode (i.e. without displaying any window) with:

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

which is very handy for running them on a Continuous Integration platform.

Cypress is full of really nice features:

- easy to set up
- easy to mock HTTP responses

- easy to test different viewports (awesome for responsive applications)
- nice enough DSL

The time-travel debugging is the feature that won my heart: Cypress takes snapshot at each step of your tests, so you can debug very easily. Just by hovering the step of the failing test, you see exactly the state of the application and can play with it.

You will write your test suite using Mocha, and you'll see that it really looks like what we had in unit tests with Vitest, plus the Cypress API to interact with your app.

Cypress gives us a `cy` object, with a few utility methods like `visit()` to go to a page. Then you have `get(selector)` to select all the elements matching a CSS selector. Or `contains(selector, text)` if you want an element with a specific text.

You can use several methods on the object returned by these methods, like `should('contain', text)` or `should('be', 'not.displayed')`. Or methods like `click()` and `type()` to act on the element.

You can also fake the API calls, and check if the requests are sent at the right time.

For example, let's say the search page of the application displays a title, and a search field allowing to find a race by calling a distant API when a user clicks on the search button.

A simple test would look like this:

```
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');
  });
});
```

These tests can be quite long to write, but they are really useful. You can do all sorts of great things with them, like recording videos, taking a screenshot and comparing them over time, at the pixel level, with something like Percy, etc.

With unit tests and e2e tests, you have the keys to build a robust and maintainable application!

All our Pro Pack exercises come with unit and e2e tests! If you want to learn more, we strongly encourage you to take a look at them: we tested every possible part of the application (100% code coverage)! In the end you'll have dozens of test examples, which you can get inspiration from for your own projects. More advanced use cases are even covered, like time manipulation in the exercise Boost a pony 🦄, or Websocket communication in the exercise Reactive score 🦄.

Chapter 17. Send and receive data through HTTP

It won't come as a surprise, but a lot of our job consists in asking a backend server to send data to our webapp, and then sending data back.

Usually this is done over HTTP, even though you have other alternatives nowadays, like WebSockets. Vue does not provide an HTTP module, but there are more and more libraries that we can use to send HTTP requests.

One of the possibilities is the fetch API, which is available in most standard browsers. You can perfectly build your app using `fetch`.

But many developers are using the axios library to build their applications. Axios is based on promises and can be used both in Node.js or browser applications.

And it comes with some nice features that `fetch` does not offer. Let's dive in!

17.1. Getting data

Axios offers a service you can import in any file.

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

It offers several methods, matching the most common HTTP verbs:

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

All these methods return a `Promise`, so you can either chain a `then` call, or use `async/await`, which is my favorite.

Let's start by fetching the races available in PonyRacer. We'll assume that a backend is already up and running, providing a RESTful API. To fetch the races, we'll send a GET request to a URL like 'http://backend.url/api/races'.

Usually, the base URL of your HTTP calls will be stored in a variable or a service, that you can easily configure depending on your environment. Or, if the REST API is served by the same server as the Vue application, you can simply use a relative URL: '/api/races'.

Using `axios`, such a request is straightforward:

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

`axios.get` returns a `Promise` with the response in case of success, so we can use `await` to get the result.

The response body is the most interesting part, represented by the `data` property:

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

As `axios` has no idea of the data you're trying to fetch, the `get` method is generic, and you can (and generally should) specify the type of data returned.

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

Note that you don't need to deserialize the response body from a string to a JavaScript array or object. That is done automatically by Axios. However, Axios won't make any check to verify that the JSON in the response indeed conforms to the generic type you specified. It's up to you to make sure you're using the correct generic type, and that the `RaceModel` interface indeed matches with the JSON that the server sends back.

Of course, you also have access to the full HTTP response, with a few fields like the `status` code, `headers`, etc.

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

The promise will be in error if the response status is different from 2xx or 3xx, and the error is then of type `AxiosError`.

Sending data is fairly easy too. Just call the `post()` or `put()` method, with the URL and the object to post:

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

Once again, no need to serialize the race object being sent to JSON. Axios does that for you. The generic type `RaceModel` here is, just as with the `get()` method, the type of the response body. So this example endpoint takes a `RaceModel` as input and returns the created `RaceModel`.

I won't show you the other methods, I'm sure you get the idea.

17.2. Advanced options

Of course, you can tune your requests more finely. Every method takes a `config` object as an optional parameter, where you can configure your request. A few options are really useful, and you can override everything in the request.

`params` represents the URL search parameters (also known as the query string) to add to the URL.

```
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
```

The `headers` option is often useful to add a few custom headers to your request. It happens to be necessary for some authentication techniques like JSON Web Token for example:

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

We'll put that in practice in the exercises very soon.

17.3. Interceptors

One of the reasons to use Axios over the Fetch API is the "interceptors" feature. Interceptors are interesting when you want to... intercept requests or responses in your application.

For example, if you want to intercept every request to add a specific header to some of them, you can now write an interceptor like this one:

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

Now every request will go through the interceptor, and receive the custom header if needed (here if there is a user logged in).

You can also intercept the response, which can be handy to handle errors generically:

```
axios.interceptors.response.use(
```

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

17.4. Tests

We now have an application calling an HTTP endpoint to fetch the races. How do we test it?

In a unit test, you don't want to really call the HTTP server: that's not what we are testing. We want to "fake" the HTTP call to return fake data. To do this, we can use `vi.spyOn` to "replace" the Axios methods and make it return what you want:

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

And we're done!

Try our exercise HTTP 🦄! We prepared a full REST API, ready for you to use. Let's fetch some races using Axios. Later you'll learn how to call a secured API with an authentication mechanism and interceptors in the exercise HTTP with authentication 🦄. Slightly related, we'll also use WebSockets 🦄.

Chapter 18. Slots

18.1. Content projection with `slot`

A common thing that we need as developers is the ability to build UI components whose content is dynamic.

For example, let's say you want to build a "card" component using the Bootstrap 4 CSS framework. The template of such a card looks like this:

```
<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>
```

You can, of course, duplicate this HTML every time you need it in your application. But at this point of your reading, you are probably thinking about building a component. Two parts are dynamic in the card (the title and the content), so this is probably what you will come up with:

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>
```

And then use it like this:

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

This works perfectly. But looking more closely to your need, you realize that the content of the card can also be complex HTML, possibly containing Vue components, directives and bindings, and not just text.

Of course, Vue supports such a use-case. It's easy to "pass" a template snippet to a child component, thanks to `<slot>`.

`slot` is a special tag you can use in your templates to include a piece of template provided by the parent component:

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>
```

And you can now use the component like this:

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

You can also pass a Vue component, for example:

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

Event though, in the final DOM, the `Pony` component is displayed inside the `Card` component, all the expressions (like `pony` in the above example) are evaluated against the parent component, not against the `Card` component. The piece of template passed to the `Card` component doesn't have access to the state of the `Card` component.

18.2. Named slots

Later, you realize that the title can also be some complex HTML. Of course, there is a way to pass multiple contents to the card component, using multiple *named* slots.

Card.vue

```
<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>
```

The slot with no name is the `default` one. To provide the title and the content to the `Card` component, you wrap them into `template` elements, and name these templates as the slot that they must fill. A `template` is just an invisible wrapper. To specify a name, you can use `v-slot:name`:

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

You can also use the shorter (and usually preferred) syntax `#name`:

```
<Card>
  <template #title>Card title</template>
  <template #default>Some quick <strong>example</strong> text</template>
</Card>
```

Note that you don't have to specify `#default`:

```
<Card>
  <template #title>Card title</template>
  <template>Some quick <strong>example</strong> text</template>
</Card>
```

And you can even get rid of the `template` element around the default slot:

```
<Card>
  <template #title>Card title</template>
  Some quick <strong>example</strong> text
</Card>
```

Some built-in components of Vue rely on this mechanism. We will use some of them in the following chapters. As an application developer, you'll probably end up using it in your own components as well.

Try our exercise Slots 🦄 to build a new component with this feature.

18.3. Fallback content

If the parent doesn't provide any content for a slot, the component can display a fallback content instead of leaving this slot empty. We can, for example, give a default title in our `Card` component:

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>
```

If we use the `Card` component and omit to specify the title, then *Default title* will be displayed.

18.4. Slot props

As we explained above, the content of the slot can access the **current** component scope. For example, if we use our card component, we can write:

```
<Card>
  <template #title>Card title</template>
  Some quick <strong>{{ example }}</strong> text
</Card>
```

where `example` is a property of the **current** component, ant not a property of `Card`.

But sometimes, you also want to access a property or a method offered by the `Card` component itself. Let's imagine that `Card` has a function to hide the component, called when the user clicks on the "Close" button:

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>
```

You can see that the component is wrapped in a `v-if`. When a user clicks on the button, the condition becomes false, and the `v-if` removes the card.

It works great, but, a few weeks later, the product owner comes back, and asks you to make some cards closable by clicking on a button inside the title passed to the card. To do that, the template passed to the `Card` component needs to call the `close` method of the `Card` component.

This is where we can use *slot props*. We can expose whatever property or function on the slot by using the usual `v-bind`:

Card.vue

```
<h4 class="card-title">
  <slot name="title" :closeCard="close">Default title</slot>
</h4>
```

This means that the `Card` component decides to make its `close` function available in the template of its title slot, under the name `closeCard` (we could of course use the same name).

Another term for those properties is *scoped props*: their scope is limited to a specific slot template.

Now, when we use our `Card` component, we can access the title props (i.e. the `closeCard` function) in the slot content:

```
<Card>
  <template #title="titleProps">
    <div class="title" @click="titleProps.closeCard()">Card title</div>
  </template>
  Some quick <strong>example</strong> text
```

```
</Card>
```

When you just want to access one or a few properties, you can leverage destructuration to make it slightly prettier:

```
<Card>
  <template #title="{ closeCard }">
    <div class="title" @click="closeCard()">Card title</div>
  </template>
  Some quick <strong>example</strong> text
</Card>
```

Note that when you expose slot properties on the default slot, you can either use a `template`:

```
<Card>
  <template #default="{ closeCard }">
    Some quick <strong>example</strong> text
    <button class="close" @click="closeCard()">x</button>
  </template>
</Card>
```

or use `v-slot` directly on the component:

```
<Card v-slot="{ closeCard }">
  Some quick <strong>example</strong> text
  <button class="close" @click="closeCard()">x</button>
</Card>
```

We will encounter these syntaxes again in the following chapters!

18.5. Typed slots with `defineSlots`

Since Vue v3.3, it is possible to use a new macro called `defineSlots` (or a `slots` option if you use `defineComponent`). This macro has been added to the framework to help declare typed slots. When doing so, Volar will be able to check the 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;
}>();
```

If the `Card` component is not used properly, then Volar throws an error:

```
<Card>
  <template #title="{ close }">...</template>
</Card>
<!-- error TS2339: Property 'close' does not exist on type '{ closeCard: () => void;
}'. -->
```

The returning value of `defineProps` can be used and is the same object as returned by `useSlots`.

Chapter 19. Suspense

This API is experimental and may change in the future.

When a component needs some asynchronous data, we saw that we can wait for these data in the `onMounted` hook.

For example, let's say we want to display a pony and its track record.

The `Pony` component could look like:

Pony.vue

```
<div>
  <h1>Pony {{ ponyModel.name }}</h1>
  <TrackRecord :ponyId="ponyModel.id" />
</div>
```

`TrackRecord` is a component with the following template:

We're not using `script setup` here on purpose, to point out what's going on more easily.

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>
```

and it fetches its track record from the server in its `onMounted` hook:

TrackRecord.vue

```
setup(props) {
  const trackRecord = ref<Array<TrackRecordModel>>([]);
  const ponyService = usePonyService();
  onMounted(async () => (trackRecord.value = await ponyService.getTrackRecord(props
.ponyId)));
  return { trackRecord };
}
```

What happens if we remove the `onMounted` call and directly await the data in the `setup`?

TrackRecord.vue

```
async setup(props) {
  const ponyService = usePonyService();
  const trackRecord = ref(await ponyService.getTrackRecord(props.ponyId));
  return { trackRecord };
}
```

It's slightly less verbose, but we now need to add `async` to our `setup` function. If you try this in your application, you'll see that Vue complains in the console and does not display the component.

Indeed, when we have a component with an asynchronous setup, Vue requires to wrap its element in a `Suspense` component. Vue provides `Suspense` and globally registers it, so you can use it in any template. Let's add it in our `Pony` component:

Pony.vue

```
<div>
  <h1>Pony {{ ponyModel.name }}</h1>
  <Suspense>
    <div>
      <TrackRecord :ponyId="ponyModel.id" />
    </div>
  </Suspense>
</div>
```

It does work again! `Suspense` waits for the completion of the asynchronous component setup and then displays it when it is ready.

You recognize (I hope!) that the `Suspense` component has a *slot*, and that we give it `TrackRecord` as the default content. That's why you can also wrap the `TrackRecord` tag in a `template` element with the name `#default`, and it would work as well.

But what is really nice is that `Suspense` provides a few more features!

19.1. Display a fallback content

We can add fallback content to our `Suspense`, by providing a `template` element named `fallback`:

Pony.vue

```
<div>
  <h1>Pony {{ ponyModel.name }}</h1>
  <Suspense>
    <div>
      <TrackRecord :ponyId="ponyModel.id" />
    </div>
    <template #fallback>Loading...</template>
  </Suspense>
```

```
</div>
```

`Suspense` now displays `Loading···` until the `TrackRecord` component is ready, and then replaces `Loading···` with it!

`Suspense` can even wait for several async components, and displays the fallback content until they are all available.

For example if we also want the `BirthCertificate` of the pony:

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>
```

Here the loading indicator is displayed until *both* the birth certificate *and* the track record are ready.

19.2. Error handling with `onErrorCaptured`

But what happens if one of the async components fails to retrieve its data? Are we stuck with the loading indicator?

No, the fallback content disappears when all components are resolved, successfully or not. So we end up with just the component in success showing up.

We can catch the potential errors though, by using the `onErrorCaptured` lifecycle hook. `onErrorCaptured` can be used in a component to capture any error from a descendent component. You can do whatever you need to handle the error, and return `true` (or don't return) to let it propagate up the chain of components, or return `false` to stop its propagation.

Pony.vue

```
const errorMessage = ref<string | null>(null);
onErrorCaptured((e: unknown) => {
  console.warn(e);
  errorMessage.value = (e as Error).message;
  return false;
});
```

A classic use-case is to then display the error to our user using a `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>
```

If one of the components fails, the error message will be displayed, and the other component as well. You can of course use a `v-else` on the `Suspense` element if you don't want to display any of the components if one fails.

19.3. Suspense events

Note that `Suspense` offers two events that you can listen to:

- `resolve`, emitted when all the child components to display are resolved;
- `pending`, emitted when a new child component in a pending state appears;
- `fallback`, emitted when the fallback appears.

You could for example combine these two events to display a loading spinner every time a child component loads, and hide it when nothing is being loaded anymore.

19.4. `Suspense` vs `onMounted`

So what's the point of using `Suspense` instead of `onMounted`? Both are fine to be honest. An `async` setup is slightly less verbose than using `onMounted`, but you then have to use a `Suspense` to display it. The biggest strength of `Suspense` is that it works with one, several, or even deeply nested async components. It provides a nice way to handle the loading and error states in a centralized way, even if you are loading a big tree of async components that won't resolve at the same time.

`onMounted` is still useful though. It is still required if you want to manipulate the DOM when the component is created. It also behaves differently:

- when using `Suspense`, the async component won't show until the data are loaded
- when using `onMounted`, the component shows right away with no data, then the data are displayed when they are loaded.

`Suspense` is still experimental, and is expected to be stable with the Vue 3.1 release.

19.5. `script setup` and `await`

You may have noted that this chapter uses components defined with `defineComponent`. This is on purpose to explain the `async` keyword we need to add. But what happens if we use `script setup`? Then there is no need to add `async`: the Vue compiler takes care of it for us!

Here is the same `TrackRecord` component written with `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>
```

You can of course use this version in your applications, and that's what we do in the following exercise.

Try our exercise Suspense 🦄 to see how we can handle our asynchronous data fetching with class!

Chapter 20. Router

It is fairly common to want to map a URL to a component of the application. That makes sense: you want your user to be able to bookmark a page and come back, and it provides a better experience overall.

The piece in charge of doing this job is called a router, and every framework has its own (or several ones). Vue is no exception and `vue-router` is the official router.

The router has a simple goal: the creation of meaningful URLs reflecting the state of our app, and each URL knowing what component should be initialized and inserted in the page. It will navigate from route to route without refreshing the page and without triggering a new request to our backend server: this is the whole point of having a Single Page Application (*SPA*).

20.1. En route

Let's start using the router. It is an optional plugin, that is thus not included in the core framework.

As we saw for the other third-party dependencies, you have to install it and include it in your application if you want to use it. But for that, we need a configuration to define the mapping between URLs and components. We can do this with a dedicated file, for example named like `router.ts`, and containing an array representing the configuration:

router.ts

```
import { createRouter, createWebHistory } from 'vue-router';
import Home from '@/views/Home.vue';
import Races from '@/views/Races.vue';
const routerPlugin = createRouter({
  history: createWebHistory(),
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/races',
      name: 'races',
      component: Races
    },
  ]
});

export default routerPlugin;
```

The exported `routerPlugin` object must then be used when we create our application, in `main.ts`:

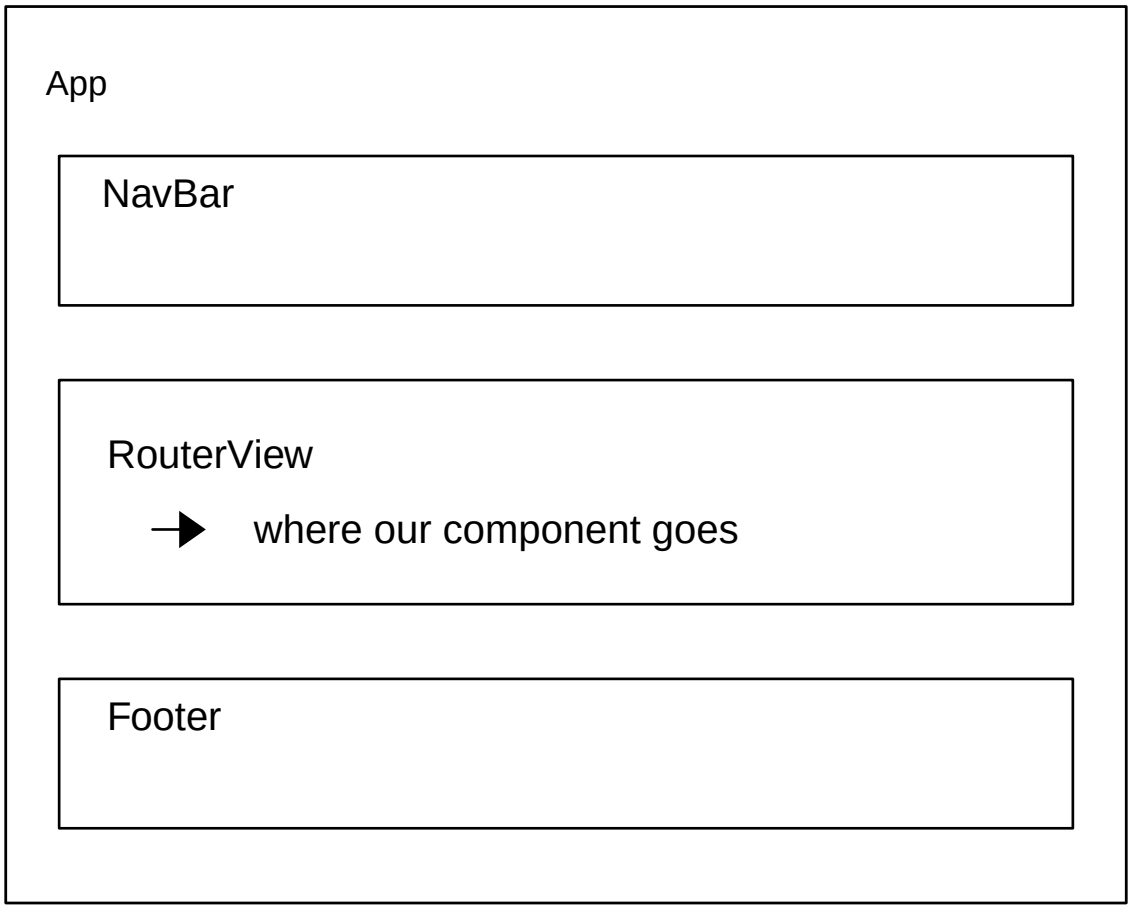
main.ts

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

As you can see, the `routes` is an array of objects, each one being a... route. A route configuration is usually a triplet of properties:

- `path`: what URL will trigger the navigation to that specific route
- `name`: a friendly name that we can use later
- `component`: which component will be initialized and displayed

You may be wondering where the component will be inserted in the page, and that's a good question. For a component to be included in our app, like the `Races` in the example above, we must use a special tag, usually in the template of the root component: `RouterView`.



This is, of course, a Vue component, provided by `vue-router`, whose only job is to act as a placeholder for the template of the component of the current route. Our app template would look like:

```
<div>
  <Navbar />
  <RouterView />
  <Footer />
</div>
```

When we navigate, everything will stay (i.e. the navbar and the footer here) and the component matching the current route will be inserted inside the `RouterView` component. You don't need to declare `RouterView` in your `App` component: when we wrote `createApp(App).use(routerPlugin)` earlier, we registered the `routerPlugin` plugin in our Vue application, and the `RouterView` component is now globally registered.

20.2. Navigation

How can we navigate between the different components? Well, you can manually type the URL and reload the page, but that's not very convenient. And we don't want to use "classic" links, with `<a href="…"></a>`. Indeed, clicking on such a link makes the browser load the page at that URL, and restart the whole Vue application. But the goal of Vue is to avoid such page reloads: we want to create a Single Page Application. Of course, there is a solution built-in.

In a template, you can insert a link by using the component `RouterLink` pointing to the path you want to go to. The `RouterLink` component expects a prop, named `to`, representing the path you want to go to, or an array of strings, representing the path and its parameters. For example in our `Races` template, if we want to navigate to the `Home`, we can imagine something like:

Races.vue

```
<RouterLink to="/">Home</RouterLink>
```

You can also use the `name` of the route:

Races.vue

```
<RouterLink :to="{ name: 'home' }">Home</RouterLink>
```

The `RouterLink` component sets a CSS class `.router-link-active` (or a class of your choice) automatically if the link points to the current route. This allows you, for example, to style a menu item as selected when it points to the current page.

It's also possible to navigate from the code, by using the `router` object returned by `useRouter()` and its method `push()`. It's often handy when you want to redirect your user after an action:

Races.vue

```
const router = useRouter();
function saveAndMoveBackToHome() {
  // ...
```

```
  router.push('/');
  // or
  router.push({ name: 'home' });
}
```

The method takes either the path as a string, or an object with the path you want to navigate to, or the name of the route.

The `router` object returned also offers a `replace()` method, if you don't want a new entry in the browser history and want to replace the current one.

20.3. Parameters

It is also possible to have parameters in the URL, and it's really useful to define dynamic URLs. For example, we want to display a detail page for a pony, with a meaningful URL for this page, like `races/id-of-the-race/ponies/id-of-the-pony`.

To do so, let's define a route in the configuration with two dynamic parameters.

router.ts

```
{
  path: '/races/:raceId/ponies/:ponyId',
  name: 'pony',
  component: Pony
},
```

We can then define dynamic links with `RouterLink`:

Races.vue

```
<RouterLink :to="{ name: 'pony', params: { raceId: race.id, ponyId: pony.id } }">
Pony</RouterLink>
```

Of course, the target component needs to access those parameters to be able to load and display the pony with the given identifier. To get the value of the parameters, the `route` object returned by `useRoute()` represents the current route and can be used inside your `setup`. It is a reactive reference, with a `params` property. This field has all the parameters of the URL.

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));
```

You can also access the query parameters with the `query` property.

The router will reuse your component if it can! Let's say our app has a "Next" button to see the next pony. The URL will change from `/ponies/1` to `/ponies/2` for example when the user clicks. In that case, the router won't destroy and re-create a new `Pony` component. Instead, it will reuse the Pony component already displayed. That means the `setup` will not be called again! If you want your component to update for this kind of navigation, you have to use a watcher, or a computed property based on `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);
  }
);
```

20.4. Router and Suspense

`RouterView` also offers a `v-slot` API, allowing us to write:

```
<RouterView v-slot="{ Component }">
  <component :is="Component" />
</RouterView>
```

`<component is="Races">` is a syntax offered by Vue to load a component, here `Races`. `is` can of course be dynamic, allowing us to dynamically load a component: `<component :is="Component">` with `Component` the variable containing the component to load.

It might not seem really useful, but it's very handy for some advanced use-cases, like for animations or when using `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>
```

Try our quiz 🦄 and the exercise Router 🦄 to learn how to configure the router, navigate between components, and test all this.

20.5. Passing parameters as props

One cool thing with the router is that you can pass the parameters as props. It decouples the component from the router, and makes it usable as a route, or a simple component used in another one.

To do so, simply add `props: true` in your route definition:

```
{
  path: '/users/:userId',
  name: 'user',
  component: User,
  props: true
},
```

Then our `User` component must declare the `userId` prop, and can use it:

User.vue

```
const props = defineProps<{
  userId: string;
}>();

const user = ref<UserModel | null>(null);
// ... fetch the user details
watchEffect(async () => (user.value = await getUser(props.userId)));
```

As you can see, there is no reference to the router in the component anymore.

You can also use `props` in the route declaration to directly pass static values to your component if you give it an object.

20.6. Redirects

A common use-case is to have a URL simply redirect to another URL in the application. This can happen because you want, for example, the root URL of your news app to redirect to the `/breaking` news category, or an old URL to redirect to a new one after a refactoring. This is possible using:

```
{
  path: '/news',
  redirect: '/breaking'
},
```

20.7. Route matching

You can also define a route that would catch all non matched URLs:

```
{
  path: '/:catchAll(.*)',
  redirect: '/404'
}
```

The star can also be used in patterns like `path: '/pony-*'`, which would match `/pony-rainbow-dash` and `/pony-pinkie-pie`.

Note that the router matches the most specific route it finds.

20.8. Nested routes

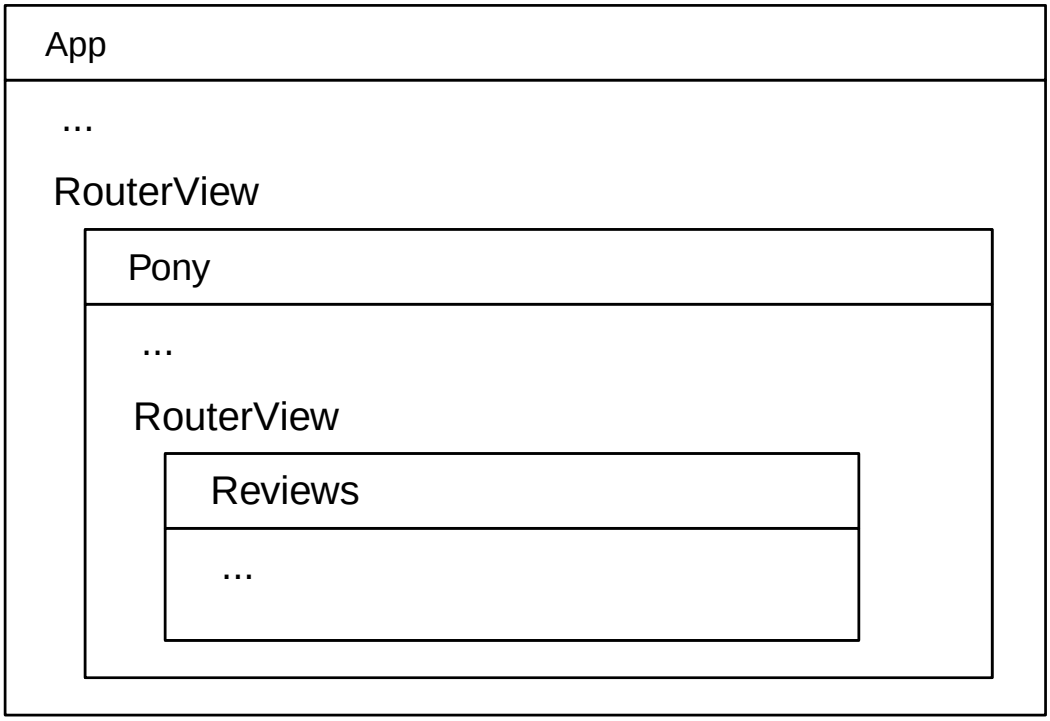
As we have seen before, when the router activates a route, the component of the route is inserted in the page at the location marked by the `RouterView` component.

This mechanism can in fact be used in nested components, too. Suppose you have a complex page to display the profile of a pony. This page would display its name and portrait at the top, and would have several tabs at the bottom: one to display its birth certificate, one to display its track record, and one to display journalist reviews about this pony. You want to have a URL for each tab, in order to be able to directly link to them. But you don't want to reload the pony and repeat its name and portrait on every on these three tab components.

The solution is to use a nested `RouterView` in the template of the `Pony` component, and to define a parent pony route, this way:

```
{
  path: '/ponies/:ponyId',
  component: Pony,
  children: [
    { path: 'birth-certificate', component: BirthCertificate },
    { path: 'track-record', component: TrackRecord },
    { path: 'reviews', component: Reviews }
  ]
},
```

When going to the URL `/ponies/42/reviews`, for example, the router will insert the `Pony` component at the location indicated by the main `RouterView`, in the `App` component. The template of `Pony`, besides the name and the portrait of the pony, contains a second `RouterView`. This is where the child `Reviews` component will be inserted.



When going to the URL `/ponies/42`, the `Pony` component will be displayed, but none of the three children components will. You might want to display the birth certificate tab by default. That can be achieved using an empty-path route, redirecting to the `birth-certificate` route:

```
{
  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 that, in the above example, the redirect is relative to the `/ponies/:ponyId` route, because it doesn't start with a `/`.

Instead of redirecting, you might want to display the birth certificate at the URL `/ponies/42`. This can also be achieved using a child empty-path route:

```
{
  path: '/ponies/:ponyId',
  component: Pony,
  children: [
```

```
    { path: '', component: BirthCertificate },
    { path: 'track-record', component: TrackRecord },
    { path: 'reviews', component: Reviews }
  ]
},
```

20.9. Navigation guards

Some routes of the application should not be accessible to all users, depending on their permissions. Of course, you should hide or disable links pointing to these routes if the user may not access them. You should also make sure the backend doesn't allow accessing or modifying resources that the user isn't authorized to access or modify. But that still won't prevent users from accessing routes that they're not allowed to access, who can simply enter their URL in the address bar.

That's where navigation guards come into play. There are 3 kinds of guards:

- global guards: they apply to all the routes of the application;
- route guards: they apply to the route where they are defined;
- in-component guards: they apply to the component where they are defined.

Let's start with the global ones!

20.9.1. Global guards

Global guards are registered directly on the `routerPlugin` instance. The guards are automatically called by the router when the route changes (i.e. not when the params or query params change, only when the router is about to activate a different route!).

`beforeEach`

The most common one is `beforeEach`:

```
routerPlugin.beforeEach((to: RouteLocationNormalized, from: RouteLocationNormalized)
=> {
  return to.name === 'login' || isLoggedIn;
  // or `return to.name === 'login' || isLoggedIn || '/login'` to redirect
});
```

As you can see, the guard is a function that takes two parameters:

- `from` is the `Route` where we come from;
- `to` is the `Route` where we go to.

The guard showed in the example is pretty simple: if the user tries to navigate to the login page, or is logged in, we return `true` (or nothing) to confirm the navigation. If not, we return `false` to abort it. You can return `'/login'` or `{ name: 'login' }` if you want to redirect the user to specific page. You

can also return a `Promise` if the decision is asynchronous.

This is actually the whole point of a guard: preventing the access to a route without explaining to the users why they can't go there, and what they should do instead (in this case, log in) is quite unfriendly. A guard is a usability tool, not a security tool. Preventing access to restricted data is the job of the server, not the client.

Note that you can define several `beforeEach` guards. The router will run them all, and only proceed if all guards confirmed the navigation.

`afterEach`

There is also a global `afterEach` that you can define. This is not really a guard, as it does not prevent the navigation. It's more a hook that allows you to do something when the navigation is confirmed.

20.9.2. Route guard

There is only one guard that you can define in the route declaration itself: `beforeEnter`

`beforeEnter`

This guard will be checked after the global ones (if any). The signature is the same has the 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. Component guards

Finally, there are two guards that can be called as function inside the component itself:

- `onBeforeRouteUpdate`: this guard is called when the params change. This is a nice alternative way to watch for param changes;
- `onBeforeRouteLeave`: this guard is called once when the component is going to be replaced by another one on a navigation. It is useful to ask for confirmation before leaving a form for example, to avoid losing data if the user didn't really intend to leave the 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

It is also possible to define meta information on a route.

```
{
  path: '/search',
  component: Search,
  meta: {
    requiresAuth: true
  }
},
```

This is a useful pattern that can be used to define a guard.

```
routerPlugin.beforeEach((to: RouteLocationNormalized, from: RouteLocationNormalized)
=> {
  if (to.meta.requiresAuth && !isLoggedIn) {
    return '/login';
  }
  return true;
});
```

20.11. Tests with vue-router-mock

It is possible to test components that use the router with Vitest and mocks of `useRoute`, `useRouter`, etc. But Eduardo "posva" San Martin, the creator of the router, also wrote a tiny library that simplifies testing: `vue-router-mock`.

`vue-router-mock` allows to create a fake router with `createMockRouter`, and to inject it in the tested component with `injectRouterMock`. We can then call `setParams()` in the tests to change the route parameters, and check that the navigation methods have been properly called for example:

```
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('/');
});
```

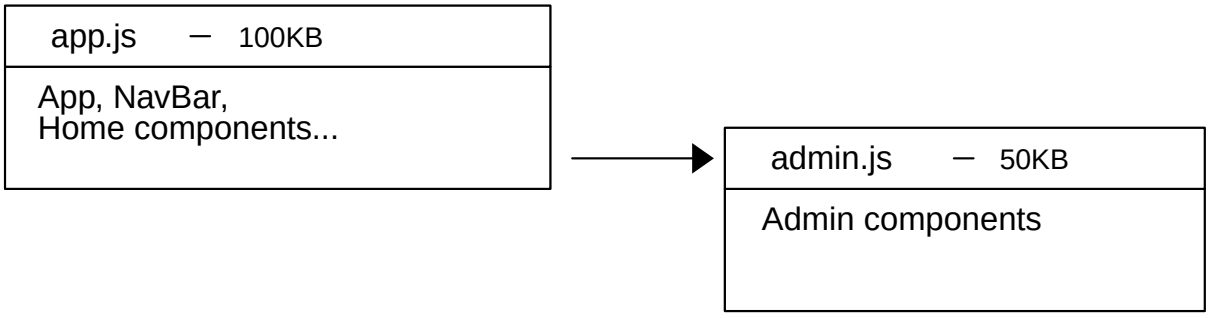
Super handy, and it's what we use in the exercises if you want other examples!

Try our exercises Router guards 🦄 and Nested views and redirections 🦄 to learn how to use this more advanced router features.

Chapter 21. Lazy-loading

When the application grows in size and features, loading the whole application at once can become a problem: the application bundle is too large and takes too much time to load and parse. Moreover, some parts of the application are only used by some users, or are used rarely, and loading them eagerly is a waste of time and bandwidth. This is where lazy-loading is useful.

Lazy loading consists in splitting the application into multiple JavaScript bundles, and loading them only when they are necessary.



Lazy loading is configured at the component level. This means that each component can be loaded lazily, either when it needs to be displayed inside the vue of another component, or when it is the component displayed by a route, and the user navigates to that route. In both cases, the same technique is used to dynamically load the component: a dynamic JavaScript/TypeScript import.

21.1. Async components

Let's start with the first case: a component is initially absent from the vue because it's inside a `v-if`, and we want to load it lazily and display it as soon as the `v-if` condition becomes true.

App.vue

```
<div>
  <h1>Ponyracer</h1>
  <button @click="showRaces = true">Show races</button>
  <div v-if="showRaces">
    <PendingRaces />
  </div>
</div>
```

We saw a while ago that, when defining a component that uses other components in its template, you have to list them in the `components` property (or just import them with `script setup`).

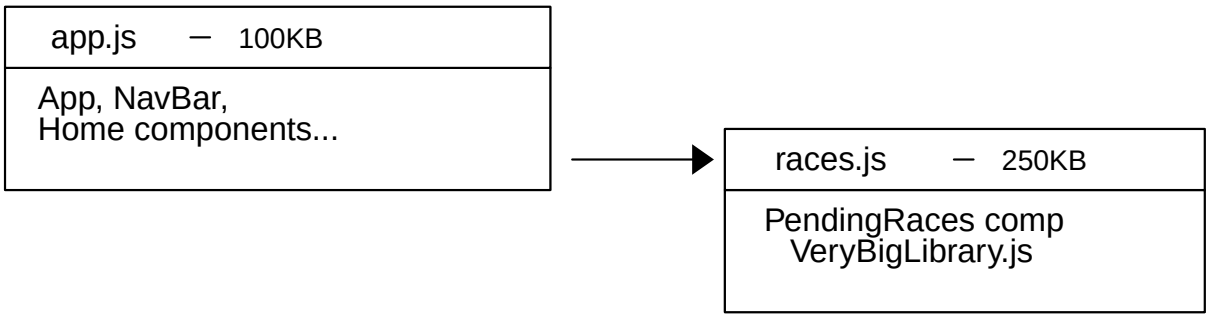
It turns out you can replace this declaration by a call to `defineAsyncComponent` and a dynamic import, instead of a static one:

App.vue

```
const PendingRaces = defineAsyncComponent(() => import('./PendingRaces.vue'));
```

If you use Webpack (and you do if you use the Vue CLI) or Rollup and Vite (if you generated your project with create-vue) then this `PendingRaces` component is going to be bundled in a separate JavaScript file, and will only be loaded when needed.

In this case, it doesn't change much. But imagine `PendingRaces` is a component that uses a very big library: the bundler will be smart enough to package the big library with our component. And all this will be loaded only when the component needs to be displayed. This can trim a lot from your main bundle, and speed up the loading of your application.



21.2. Async components and Suspense

We showed how we can use `Suspense` to dynamically load the *data* needed by a component and display an alternative while the data is loading. We can also use `Suspense` to lazily load the component itself, and display an alternative while the component is loading.

Let's illustrate that with the `PendingRaces` asynchronous component:

App.vue

```
<div>
  <h1>Ponyracer</h1>
  <Suspense>
    <PendingRaces />
    <template #fallback>Loading...</template>
  </Suspense>
</div>
```

This will automatically display `Loading···` while the component is loading.

You can also have a similar behavior without using `Suspense`. `defineAsyncComponent` can also take an object as argument, allowing specifying:

- a `loadingComponent`, to specify what to display while loading;

- an `errorComponent`, to specify what to display in case of an error;
- a `delay`, to specify after how long you want the loading indicator to appear;
- a `timeout`, to specify how long you want to wait before considering that the loading has failed and showing an error (the component still displays after that if it finally resolves).

By default, `delay` is 100ms, and there is no `timeout`, `loadingComponent` or `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. Lazy-loading with the router

A more common use-case is to lazily load a component associated to a route: when the user clicks on a link to navigate to a route, the route's component must be lazily loaded first, before the navigation is done.

It's pretty straightforward: you use the same dynamic import syntax, but in the route declaration:

router.ts

```
{
  path: '/races',
  component: () => import('@/views/Races.vue')
},
```

And you're done! The bundler will create a separate bundle for the `Races` component, and that bundle will be loaded only when the user navigates to the `/races` route.

21.4. Grouping components in the same bundle

It often makes sense to load several components at the same time, instead of having one bundle for each lazy-loaded component.

When using Vite, you'll need to declare which dynamic imports you want to group inside the Vite configuration, using the `build.rollupOptions` property:

vite.config.ts

```
export default defineConfig({
  plugins: [
```

```
    vue({
      features: {
        propsDestructure: 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...
          ]
        }
      }
    }
  }
});
```

Try our exercise Lazy-loading 🦄 to learn how to use lazy-loading and how much you can gain in our application. There is also a quiz 🦄 on lazy-loading and advanced router concepts!

Chapter 22. Forms

Forms are hard: you have to validate the inputs of your user and display errors. Some fields are required, others are not. Some depend on another field, so you want to react on field changes, etc.

Vue offers a directive we have not seen yet, `v-model`, to help us build forms, but not much more. You are on your own for the validation, error messages, etc. Luckily there are some good libraries to help us.

Let's go through an example, first by using `v-model`, and then let's see what a third party library can bring to the table.

22.1. The `v-model` directive

You can apply the `v-model` directive to an `input`, a `textarea` or a `select`. You give it a property to update, and it automatically keeps it in sync between the code and the form field.

This is called *two-way binding*:

- if the property value changes in the code, the value displayed in the template is updated;
- if the user enters a new value, the property value is updated in the code.

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` is in fact just a syntactic sugar, equivalent (for an input) to:

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` -->
```

You can see the two-way binding more clearly:

- if `user.name` changes, the value is updated
- if the user enters a value, the `input` event fires, and updates `user.name`.

It also works with checkboxes:

Register.vue

```
<input v-model="user.isAdmin" type="checkbox" />
<!-- displays "Is admin: true" or "Is admin: false" -->
<p>Is admin: {{ user.isAdmin }}</p>
```

But sometimes you don't want `true` or `false`. In that case `v-model` offers `true-value` and `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>
```

It also works with radio buttons, with static or dynamic options:

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>
```

And, of course, it works with `select`, with static or dynamic options:

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 that it also works with an object as the model.

To submit the form, you can listen to the `submit` event and `prevent` the default submission:

Register.vue

```
<form @submit.prevent="register()">
```

In the `register` function, you'll have access to the `user` property, filled with the values entered.

Vue also offers a few modifiers for `v-model`.

22.1.1. `.trim` modifier

You can add `.trim` to a `v-model` to remove the extra spaces:

Register.vue

```
<input v-model.trim="user.name" />
```

If the user enters a value with spaces at the beginning or at the end, then `user.name` will receive the value without the extra spaces.

22.1.2. `.number` modifier

You can add `.number` to a `v-model` to convert it to a number (`type="number"` is not enough as it usually returns a string):

Register.vue

```
<input v-model.number="user.age" type="number" />
```

If the users enters `22`, then `user.age` is `22` and not `'22'`. But if the value entered cannot be parsed by Vue, then it is returned as is. For example, if the user enters `Not a number`, you will either have an empty string if the `input` is with the attribute `type="number"` or `'Not a number'` if not.

22.1.3. `.lazy` modifier

When I was showing you the version without the syntactic sugar, I used:

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` -->
```

but you can change `v-model` behavior to listen to the `change` event instead by using the `.lazy` modifier.

This means `v-model` will update the value only when a text input loses its focus.

Register.vue

```
<input v-model.lazy="user.name" />
```

`v-model` is a nice directive for handling basic use-cases. It quickly shows its limits though. What about input or form validation? Error messages?

You can handle that all by yourself, as we'll see in the exercise for `v-model`, or use a third party library to help you do the heavy lifting.

There are two popular libraries in the Vue ecosystem:

- Vuelidate
- VeeValidate

They both bring similar needed features, and you would be fine with either one. But VeeValidate is my favorite, so we're going to dive deeper into this one!

Try our exercise Register 🦄! It's part of our Pro Pack, and you'll learn how to build a complete form with `v-model`. This exercise lets you handle the validation and error messages yourself!

22.2. Better forms with VeeValidate

We very often need client-side validation in forms. This is not a replacement for server-side validation, but it's nice to tell our users that some values are incorrect in the current form. It also gives instant feedback, unlike server-side validation.

VeeValidate is here to help us. It works with the `v-model` directive we just discovered and adds some nifty features to our forms.

VeeValidate comes with a lot of validation rules, but none of them are included by default. They are provided in a separate package called `@vee-validate/rules`. To make your application as small as

possible, you have to declare what validation rules you want to use. You can import all of them at once, but this is not a good idea, as you'll probably just use a few.

Let's list the most common validation rules available:

- `required` to make a field required
- `min:N` and `max:N` to make sure the value has at least/at most N characters
- `min_value:N`, `max_value:N`, `between` to make sure the value is at least/at most N for a number, or between two values
- `email` to check if the value is a valid email
- `url` to check if the value is a valid URL
- `alpha`, `alpha_num`, `alpha_dash`, `alpha_space` to check that the value is alphabetic only/with numbers/with dashes/with spaces
- `numeric` to check that the value is a number
- `regex` to define your own regular expression

You have some specific rules for inputs of `type="file"`:

- `mimes` allows defining what MIME types you want to accept for this file
- `ext` allows defining what extensions you want to accept for this file
- `size` allows defining the maximum file size
- `image` only allows image formats
- `dimensions` allows defining the exact image dimensions

Some rules are meant to cross-validate two fields:

- `required_if` makes a field required if another one is filled with one of the given values
- `confirmed` checks if a field has the same value as another one (classical password/confirm password check).

When you want to use a rule in your application, you have to load it:

main.ts

```
import { defineRule } from 'vee-validate';
import { confirmed, min, required } from '@vee-validate/rules';

defineRule('min', min);
defineRule('required', required);
defineRule('confirmed', confirmed);
```

As you can see, we could rename the rules, but I usually use the default rule names. Once we defined the rules we want to use, we can use them in our components to validate inputs.

You can also configure VeeValidate, for example to indicate *when* the validation happens. By

default, it validates on the events `blur` and `change`. But I like to validate on the `input` event as well:

main.ts

```
import { configure } from 'vee-validate';
configure({
  validateOnInput: true,
});
```

VeeValidate offers two ways to add validations to your forms:

- one based on the Composition API
- one based on components to use in your templates (a pattern called *Higher Order Components*, or *HOC*)

22.2.1. VeeValidate with Composition API

VeeValidate offers a function to declare a field: `useField()`.

Register.vue

```
import { useField } from 'vee-validate';

const { value: name, errorMessage: nameErrorMessage, meta: nameMeta } = useField('
name', { required: true });
```

The object returned by `useField()` has several interesting properties, like the `value` of the field, that we can use in a `v-model`. VeeValidate also handles the validation of the field, and offers an `errorMessage` property, which contains the potential error message for our input.

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` in fact offers more properties than that on `meta`:

- `valid` is a boolean indicating if the field is valid or invalid

- `dirty` is a boolean indicating if the field is pristine (no value was entered) or dirty (the user entered a value different than the initial one)
- `touched` is a boolean indicating if the field has been blurred or not

VeeValidate also helps to validate the whole form state, and provides another composition API function: `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);
});
```

You can register as many fields as you want. `useForm` returns an object with a function `handleSubmit` that we can use to handle the submission, an object `errors`, containing the validation errors, and the same properties as `useField` in a property `meta`. With a logic fairly easy to follow: `valid` is `true` if every field is valid, and `false` if at least one field is invalid; `dirty` is `false` if every field is pristine, and `true` if at least one field is dirty, etc.

For example, you can disable the submit button of the form if it is invalid:

Register.vue

```
<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>
```

The composition API offered by VeeValidate does the heavy lifting of validating our form and computing its state. It's powerful and easy to use. And, as we'll see in the next section, there is an even simpler way to use VeeValidate.

22.2.2. VeeValidate with Higher Order Components

Instead of using the Composition API, VeeValidate offers a component to define a form field and to add validation on this field. This component is named... `Field`! `Field` on its own doesn't do much. But you can use it with the other component VeeValidate provides: `Form`. `Form` emits a `submit` event with the values of the form:

Register.vue

```
<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>
```

You just have to code the function that handles the submission:

Register.vue

```
import { Field, Form } from 'vee-validate';

function register(values: Record<string, unknown>) {
  // { name: string; password: string }
  console.log(values);
}
```

As you can see, `Field` just expects a name, and (by default) displays an `input`. If you want to give the field an initial value, you can use the `value` prop. Or you can also use `v-model`: that can be handy if you want to react on value changes! You just have to use a reactive value and a watcher.

Register.vue

```
const user = reactive({ name: 'Cédric', password: '' });

const passwordStrength = computed(() => computeStrength(user.password));
```

And then add a `v-model` on your `Field`:

Register.vue

```
<Field id="password" v-model="user.password" name="password" type="password"
rules="required" />
<div id="strength">{{ passwordStrength }}</div>
```

You can also specify a `type` prop (for a `type="number"` input), or a different element with the `as` prop (for example, `:as="textarea"`). If you want to display some custom HTML, to maybe dynamically add a class if the field is invalid, you can do it easily:

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>
```

This is a pattern we have already seen in the Slots chapter, and it is a very useful one: the wrapper component, or Higher Order Component (or *HOC*), with a *slot* content.

`Field` offers a slot prop, containing all the properties that `useField()` returns.

We can access the `meta` property as you can see in the previous example, and we can also access the `errorMessage` property, which contains the potential error message for our input. We can do even better, as VeeValidate offers a third component to display error messages: `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 is smart enough to not display the message until your user played with the input. So it does not shout at a poor user who just landed on your page.

The message displayed here is `name is not valid.`, but it can be customized when we configure the library, using the `@vee-validate/i18n` package:

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 0:{min} characters.',
    }
  })
});
```

The message will now be `The name is required.`.

You can also rename the field if you need to, with the `label` prop:

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 that we could use several rules on the same field:

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` also offers a few properties in its slot prop, like `meta` which contains the state of the form (`valid`, `dirty`, etc.).

Register.vue

```
<Form v-slot="{ meta: formMeta }" @submit="register($event)">
  <!-- Several fields... -->
  <button :disabled="!formMeta.valid">Register</button>
</Form>
```

A third component is available, `FieldArray`, to handle array of values. VeeValidate also allows localizing the messages in your language, defining your own validation rules, adding accessibility attributes to your fields, etc.

VeeValidate comes with a devtools plugin, allowing to directly inspect the state of your form in the devtools of your browser when you are developing 😉.

Try our exercise Login 🦄! It's part of our Pro Pack, and you'll learn how to build a complete form with validation rules with VeeValidate.

22.2.3. Custom validators

VeeValidate offers a few built-in rules, but you can also write your own.

A validator rule is fairly simple: it's a function that returns `true` if the value is valid, or `false` if it is not. It can also directly return the error message, but I usually only use a boolean as a return value. The function can also return a `Promise` if you need an asynchronous validation (like checking something with the server).

Let's build a simple validator that checks if a value is between 18 and 130.

validators.ts

```
export function between18And130(value: string) {
  return +value >= 18 && +value <= 130;
}
```

You then add the rule, as you do for the built-in ones:

forms.ts

```
defineRule('between18And130', between18And130);
```

And define a message for this error:

forms.ts

```
between18And130: context => `The ${context.field} must be between 18 and 130.`,
```

We can now use it in a component:

Register.vue

```
<Field v-slot="{ field, meta }" name="age" rules="required|between18And130">
```

A validator can also have parameters. Instead of hardcoding the min and max values, we can give them to the validator.

validators.ts

```
export function betweenParams(value: string, params: Array<string>) {
  const min = +params[0];
  const max = +params[1];
  return +value >= min && +value <= max;
}
```

We pass parameters using `:`, and `,` to separate them.

Register.vue

```
<Field v-slot="{ field, meta }" name="age" rules="required|betweenParams:17,120">
```

The error message can reflect these parameters as well:

forms.ts

```
betweenParams: context => {
  const params = context.rule!.params as Array<string>;
  return `The ${context.field} must be between ${params[0]} and ${params[1]}.`;
}
```

One last cool feature is the possibility to cross-validate fields. For example, our validator could use the values defined in two other fields for the min and max values.

Given a `min` and a `max` field, we can write:

Register.vue

```
<Field v-slot="{ field, meta }" name="age" rules="required|betweenParams:@min,@max">
```

The `@something` syntax indicates to VeeValidate that it needs to use the value of the `Field` named `something`.

To sum up, it's fairly easy to create your own validators. Write a function returning a boolean, import the rule, define an error message and use it, as if it was a built-in validator.

Try our exercise Custom validators 🦄! It's part of our Pro Pack, and you'll learn how to write your own validators in practice.

22.3. Custom form components

HTML defines a large set of input types: text, password, checkbox, etc. But sometimes these standard types don't fit the bill.

Vue allows defining custom components, and it's actually possible to make them act as HTML form controls, i.e. bind them by applying the `v-model` directive.

Fulfilling its contract is relatively straightforward. You have to:

- accept a prop called `modelValue`;
- notify Vue that the user changed the value somehow, by emitting an event called `@update:modelValue`;

We will illustrate all of this using a custom *rating* component. This component allows rating a movie, for example, by giving it a note between 0 and 5. But instead of using a `number` or `range` input, we would like the user to do that by simply clicking one of 6 buttons (that would typically be displayed as star icons, but we'll leave that out in the following example).

Here's the code of such a component:

```
<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>
```

And here is its 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>
```

Voilà. We now have a shiny reusable rating component that can be used to rate something in any form, by simply applying `v-model` to the component:

```
<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. `defineModel` macro

When you have a custom form component that just needs to bind the `v-model` value to a classic input, the prop/event mechanic we saw can be a bit cumbersome:

```
<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>
```

Since Vue v3.3, it is possible to simplify this component, by using the `defineModel` macro:

```
<template>
  <input v-model="modelValue" />
</template>

<script setup lang="ts">
const modelValue = defineModel<string>();
</script>
```

`defineModel` also accepts a few options:

- `required: true` indicates that the prop is required
- `default: value` lets specify a default value

It is also possible to handle modifiers since v3.4:

```
<template>
  <input v-model="count" />
</template>

<script setup lang="ts">
const [count, countModifiers] = defineModel<number, 'number'>({
  set(value) {
    if (countModifiers?.number) {
      return Number(value);
    }
    return value;
  }
});
</script>
```

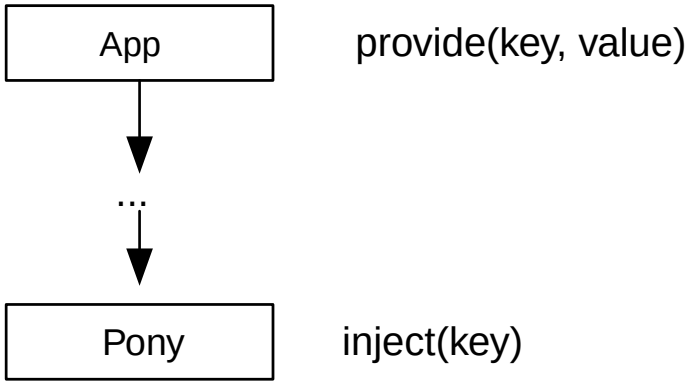
Chapter 23. Provide/inject

Vue has a basic dependency injection system. I hesitated to write a chapter about it, as it's not something you'll use every day. It's mostly used by library authors, but it also has one handy use-case in applications.

23.1. A way to avoid props drilling

When an application grows, we sometimes find ourselves passing props from a component down to its grand child, or great-grand child.

One way to avoid this is to use `provide`/`inject`. `provide` allows to register a value for a key in a component, and `inject` allows to get the value in one of its children down in the component tree.



The key can be a `string` or a `number` (a third type, `InjectionKey`, is allowed, and that it will be explained later), and the value can be whatever you want: a static value, a reactive one, a function… If the value needs to change, then a reactive value makes sense:

App.vue

```
const color = ref('#123456');
provide('color', color);
```

A component down the tree can then inject the value.

Pony.vue

```
const color = inject<Ref<string>>('color');
```

The value can then be used like any reactive property. If the `App` component updates the value of `color`, then the `Pony` component will be refreshed!

As you can see, I needed to type `inject` to indicate to TypeScript that the value is of type

`Ref<string>`. It's possible to do better by using a key of type `InjectionKey<T>`. `InjectionKey` is in fact just a `Symbol`, a special primitive data type in the browser, which has the particularity to create unique values. That makes this type a perfect candidate for a key, as you're sure it will be unique:

```
const symbol1 = Symbol();
const symbol2 = Symbol('hello');
const symbol3 = Symbol('hello'); // symbol2 !== symbol3
```

So we create an `InjectionKey`:

injection-keys.ts

```
import { InjectionKey, Ref } from 'vue';

export const colorKey: InjectionKey<Ref<string>> = Symbol('color');
```

And use it to provide the color:

App.vue

```
const color = ref('#123456');
provide(colorKey, color);
```

And to inject it:

Pony.vue

```
const color = inject(colorKey);
// color is automatically typed as `Ref<string> | undefined`
```

Note that you can add a second parameter to `inject` to have a default value:

Pony.vue

```
const color = inject(colorKey, ref('#999999'));
// color is automatically typed as `Ref<string>`
```

You can also provide a value at the application level:

main.ts

```
createApp(App)
  .provide(colorKey, ref('#987654'))
  .mount('#app');
```

23.2. Testing components with `inject`

When you write unit tests for a component that uses `inject`, you'll need to provide a value. Vue Test Utils allows defining a `provide` option to do so:

Pony.spec.ts

```
const wrapper = mount(Pony, {
  global: {
    provide: {
      color: ref('#999999')
    }
  }
});
```

or if you use an `InjectionKey`:

Pony.spec.ts

```
const wrapper = mount(Pony, {
  global: {
    provide: {
      [colorKey as symbol]: ref('#999999')
    }
  }
});
```

23.3. Hierarchical providers

The first use-case was about avoiding props drilling. But to be honest, you don't really need provide/inject for this use case: you can simply use a function `useColor()` that returns a reactive color, and you have the same result.

But `inject` is in fact a bit more powerful:

- it starts by looking into the parent component if there is a provided value for this key
- if not, it looks into the grand-parent
- etc.
- if not, it looks into the root component
- if not, it looks into the provided values of the application

Of course, it stops looking when one provider is found, and returns its value.

In the end, if no provider was found, it returns an `undefined` value and logs a warning (except if you provided a default value):

```
'[Vue warn]: injection "Symbol(color)" not found'
```

This hierarchy of providers can be handy if you want to override the provided value for a subset of components.

23.4. Plugins and provide/inject

Vue has a plugin mechanism. Plugins are installed using `app.use(plugin)`. The router for example uses this mechanism:

main.ts

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

When you want to grab the router in a component, you use `useRouter()`:

Races.vue

```
const router = useRouter();
function saveAndMoveBackToHome() {
  // ...
  router.push('/');
  // or
  router.push({ name: 'home' });
}
```

Under the hood, the router, like most plugins, uses provide/inject!

When you create the router using `createRouter()`, it provides the router to the application. `useRouter()` is fairly simple and only does `return inject(routerKey)`. In fact, you can even use that in your component to get the router instance .

This is the most useful use-case for provide/inject, and it explains how most plugins work.

Chapter 24. State management

In a well-designed web application, the state should be stored on the server, and in the URL. You should be able to refresh any page (and thus restart the application), and get back to where you were before: the URL allows remembering where you were, and the data displayed by that page is stored on the server and retrieved thanks to the URL parameters.

The various components on that page however, once the data has been loaded from the server, often need to share some state. And part of the state can even be global to the whole application, like for example the currently logged-in user: the navbar needs to display it and allow logging out, the home page needs to display a greeting message, any other component might need it to know if the user is allowed to perform some action.

So, how can components share state?

You can of course use props and emits: a parent component passes some state to a child with props, and the child emits events to its parent so that the state can be modified. That is fine for parent-child associations, but it can be tedious when the state must be shared between many sibling components, or when the tree is deep. And for global state, which needs to be shared by completely unrelated components, displayed on various routes, that doesn't fit the bill.

The solution in such cases consists in using the *store* pattern.

24.1. Store pattern

Let's say that two components need to access the currently logged-in user. We can use the Vue Composition API to declare a reactive property that both components will use. I usually put this piece of state in a "service", but you may call it a store as well.

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
  };
}
```

Then the components which want to access the user can do so very easily:

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>
```

The wonderful thing is that if the `userModel` value changes, then the components will automatically update ✨.

For example, if the `Navbar` component has a "Sign out" button:

Navbar.vue

```
const { userModel } = useUserService();
function signout() {
  userModel.value = null;
}
```

The `Home` component automatically displays `Welcome, anonymous comrade!` when someone clicks the "Sign out" button of the `Navbar`.

But, to simplify debugging, you usually don't want to mutate the reactive property directly in the components. The proper way to update the reactive property is to delegate to the service:

Navbar.vue

```
const { logout } = useUserService();
```

This simple pattern is called the "Store pattern", and it works very well, even for large applications!

Try our exercises State management 🦄 and Remember me 🦄! You'll learn how to

use the Composition API to handle the logged-in user in Ponyracer.

24.2. Flux-like libraries

The store pattern is in fact our favorite way to handle state in an application. But it is not the only one.

Some developers are in love with the Flux pattern, which is heavily inspired by the Elm framework, and was popularized by the React ecosystem. The most known library in the React community is Redux. You've probably heard that name already.

The Flux pattern and libraries like Redux encourage developers to not update the store directly, but rather to dispatch actions. These actions are handled by reducers, in charge of updating the state. Sounds confusing? It is at first, and can lead to over-complicated architectures.

In fact, the author of Redux himself, Dan Abramov, wrote a very famous blog post You might not need Redux, because at some point everybody was putting Redux in their React application, even if it was just complicating things for them. The Angular ecosystem is no stranger to this problem: NgRx is quite popular, but it is very often useless, as Angular can handle state natively (with a similar concept to the store pattern we saw above).

You may understand where this goes: I'm not sure you need something else than Vue to handle the state of your application. You can start without third party libraries, and see how it goes: you'll be fine.

But I can see you're curious, and want to check out the popular state management libraries in Vue. Fine, let's go!

24.3. Vuex

The most popular state management library in the Vue 2 ecosystem was Vuex. Vuex v4 works with Vue 3, but is not ideal, as it is not type-safe, and not super well integrated with the Composition API. In Vue 3, the recommandation is now to use Pinia, which is basically Vuex v5, but with a different name and a cute logo. Feel free to skip this section and go straight to the Pinia section below!

With Vuex, you start by creating a store with `createStore`, usually a single one for the complete application.

A store has an initial state and mutations, which are synchronous modifications of the state.

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;
```

The created store is a Vue plugin, that we need to install in the application (like we did for the Router). Then we can use the store in components with `useStore()`. But TypeScript can't know the exact type of the store like this. To make sure we have a well-typed store, the recommended way to register it is to use an injection token (we talked about that in the previous chapter, if you can remember):

store.ts

```
export const storeKey: InjectionKey<Store<State>> = Symbol();
```

We can now register it:

main.ts

```
createApp(App)
  .use(store, storeKey)
  .use(routerPlugin)
  .mount('#app');
```

In a component, we can get the store with `useStore(storeKey)`, and it will be properly typed! To avoid repeating it in every component, we can even declare a function that returns it in:

store.ts

```
export function useAppStore() {
  return useStore(storeKey);
}
```

And then use this function to get a hold on the store, and access its state:

Home.vue

```
const store = useAppStore();
const { userModel } = toRefs(store.state);
```

When you want to modify the state, you're going to use a mutation to trigger a synchronous update, with `store.commit()`:

Navbar.vue

```
const store = useAppStore();
function logout() {
  store.commit('logout');
}
```

The name of the mutation is sadly not type safe in Vuex v4. So you'll often encounter teams that declare the name of the mutations as an enum in the store, and use this enum everywhere.

You can also handle asynchronous updates with `actions` in the 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;
```

An action usually does an asynchronous call, and then commits the result to update the state.

As the application grows, you can easily see that the store might get bigger and bigger. You can hopefully split the store into "modules". Each module has its own state, mutations and actions.

Even with modules, one of the obvious downsides of Vuex is the boilerplate it needs to perform anything. Another one is the not yet perfect integration with TypeScript and Vue Composition API.

On the plus side, Vuex allows to centralize the state in one place, and can help debugging the application. Vue devtools can for example show the current state in the store, and even allows to "time-travel". Vuex also supports plugins. A `createLogger` plugin is provided, allowing to log every store modification in the console, which can be very handy while developing. You can of course build your own plugins.

24.4. Pinia

The author of the Vue router, @posva, created an alternative to Vuex, with a nice Composition API. The library is called Pinia, and it comes with the cutest logo.

The project started as an experiment for Vuex v5, but it was so good (and with such a cute name and logo) that it ended up being the official recommendation for the state-management library of Vue.

As you can imagine, the philosophy is quite similar to Vuex. You define a store (or many) with `defineStore`. This store has a state and actions. It can have getters too. But it does not have mutations like Vuex: it just has actions. I really like that about Pinia: mutations tend to be a lot of unnecessary and quite verbose code. The other main difference is that Pinia is fully type-safe, and has a very nice composition API. In fact, you can write your store as a function, pretty much how you write your components (the alternative is to use an object with `state`, `getters` and `actions` properties, but I prefer the function version).

If we take the same example, here is how the store looks like:

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 };
});
```

Then we tell our application to use Pinia:

main.ts

```
createApp(App)
  .use(createPinia())
  .use(routerPlugin)
  .mount('#app');
```

And we can use the store in our components (note that to keep the reactivity, you need to use `storeToRefs()` when destructuring the store):

Home.vue

```
import { storeToRefs } from 'pinia';

const store = useAppStore();
const { userModel } = storeToRefs(store);
```

`storeToRefs()` is very similar to `toRefs()` (that you can use here as well), but it only returns the reactive properties (and not the methods of the store).

When you want to modify the state, you don't need a mutation, you can simply mutate the state (without having to bother with `.value`, because the store itself is wrapped with `reactive`):

Navbar.vue

```
const store = useAppStore();
function logoutDirectModification() {
  store.userModel = null;
}
```

or you can call a function exposed by the store:

Navbar.vue

```
const store = useAppStore();
function logout() {
  store.logout();
}
```

Even if Pinia doesn't have a notion of mutation, it remembers all the changes made to the store. And it comes out of the box with a plugin for the Devtools, which lets you see what your state is, inspect how it was updated on a timeline, and of course lets you update it manually.

You can also subscribe to store changes thanks to `$subscribe()`. And Pinia automatically unsubscribes when the component is destroyed.

24.5. Testing Pinia

Pinia comes with a tiny package called `@pinia/testing` that makes your life easier when it comes to testing. The library offers a `createPiniaTesting()` function that does all the heavy lifting. By default, it stubs all the actions, so we can easily test components that trigger them:

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();
  });
});
```

In case that you don't want to stub the actions, you can use the `stubActions: false` option:

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. Why use a store?

As stated above, the Composition API is enough in most cases.

A store like Pinia can be necessary in one scenario: server-side rendering (SSR). If your application is rendered on the server (for example if you use Nuxt), then you'll need to properly handle the global state to avoid leaking the information from a user to another one if you're not careful. As manually written stores are usually singletons, they may lead to a well-known problem called "cross-request state pollution". Pinia can help you to handle that: it even comes with a Nuxt integration.

Other than that, a store like Pinia can be nice to have:

- for debugging, using the Devtools
- for testing, with its `@pinia/testing` package
- for its Hot Module Replacement support, allowing to update values in stores when we develop, and see the results without having to reload the application
- for using existing plugins (or building your own)

To sum up this chapter: you can pick various options to handle the state of your application:

- just use the Composition API: it's good enough in most cases
- or use Pinia (avoid Vuex for Vue 3 applications).

Try our exercise State management with Pinia 🦄 where you'll refactor our state management by using Pinia.

Chapter 25. Animations and transition effects

Animations are a very nice addition to an application. It's not the first thing you do, of course, but it can really improve the user experience.

Vue makes it really easy to add animations and transition effects to an application: let's dig into that.

25.1. Pure CSS animations

Before digging into the transitions effects, let's start with "basic" CSS animations. You'll often encounter a situation where you want to focus the user's attention on a part of the screen. Animations are a very good way to do that! We are going to let our user know that the form he/she filled is ready to submit, for example by adding a shaking effect on the submit button.

The process to add an animation is fairly simple. An animation style relies on the animation CSS property. It usually triggers an animation that you can define using `@keyframes`.

A "shaking" animation would look like:

```
.form-valid {
  animation: shake 300ms ease;
}

@keyframes shake {
  10%,
  50%,
  90% {
    transform: translateX(0.5rem);
  }
  30%,
  70% {
    transform: translateX(-0.5rem);
  }
}
```

`@keyframes` can be disconcerting at first. Here we build a `shake` animation, by defining steps:

1. At the beginning of the animation (0%), the element is on its initial position.
2. Then (10%), we want to translate a bit on the right using `translateX`.
3. Then (30%), we want to translate a bit on the left. Note that we don't need to define where the element is at 20%, the browser will figure it out.
4. Then (50%), we move on the right.
5. Then (70%), we move on the left.

6. Then (90%), we move on the right.
7. At the end (100%), the element goes back to its initial position.

This `shake` animation will be applied on an element that has the `form-valid` CSS class, for 300ms, with an easing effect (slower, then faster, then slower). You can of course choose other effects (`linear`, `ease-in`, `ease-out`, etc.).

We're now going to add/remove the `form-valid` CSS class on the submit button, depending if the form is valid or not:

```
<button :class="{ 'form-valid': valid }" :disabled="!valid">Save</button>
```

When the form becomes valid, the browser shakes the button for 300 milliseconds.

Let's now see what Vue gives us to animate transitions.

25.2. Enter/leave transitions

Transitions are another way that CSS offers to animate elements. They are somewhat simpler than animations. With CSS transitions, you can specify that a change in one or several CSS properties of an element will happen in a progressive way.

For example, if you define the following CSS rules:

```
.pill {
  transition: transform 300ms ease-out;
}

.pill.pill-selected {
  transform: scale(1.1);
}
```

You're saying that, for elements with the class `pill`, every change in the value of the `transform` property will be done in a progressive way, over 300 milliseconds.

So if you dynamically add the `pill-selected` class to a `.pill`, it will become slightly bigger over 300 milliseconds. And if you remove the class `pill-selected` from a `.pill`, then it will get back to its normal size over 300 milliseconds.

The problem however is that what we often want is to apply a transition effect when an element appears in the DOM or disappears from the DOM. But `v-if` and `v-for` don't change the style of an element: they simply add or remove elements from the DOM. So we need something more to, for example, have fade-in and fade-out transitions when elements appear and disappear.

Vue lets you animate `v-if` with the `<Transition>` component. `Transition` expects a single child element (with the `v-if`), and automatically applies CSS classes on it when the child element enters or leaves.

The following CSS classes are applied when the element enters:

- `v-enter-from` just before the element appears, at the very beginning of the transition.
- `v-enter-active` while it appears. This is the one you can use to define the animation of the transition.
- `v-enter-to` when the transition is done and the element appeared (replaces `v-enter-from`).

Then, when the element is removed, the following CSS classes are applied:

- `v-leave-from` when the transition starts
- `v-leave-active` while the transition
- and `v-leave-to` when the transition is done.

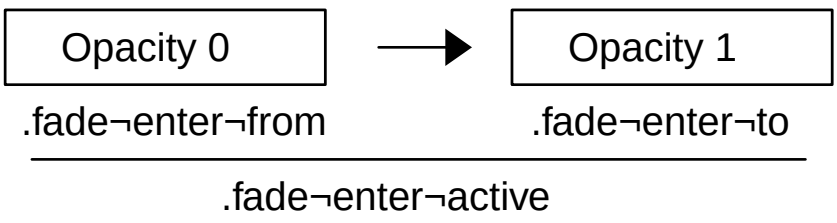
You can also name the transition (`fade` in the following example), and then the classes are prefixed with the name instead of `v-`.

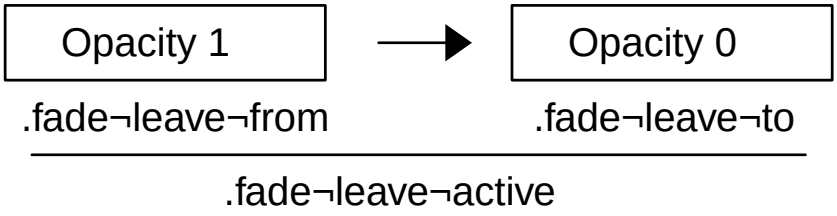
```
<Transition name="fade">
  <div v-if="display">Content</div>
</Transition>
```

So here the classes are going to be `fade-enter-from`, `fade-enter-active`, etc. Let's define a "fading" effect in CSS using these classes:

```
.fade-enter-active,
.fade-leave-active {
  transition: opacity 1s;
}
.fade-enter-from,
.fade-leave-to {
  opacity: 0;
}
```

Here the transition will last one second, and will progressively change the opacity from 0 to 1 when the element enters, and from 1 to 0 when it leaves.





Every time the condition of the `v-if` will change, the element will fade in or fade out over a second, instead of appearing or disappearing brutally.

Your imagination (and CSS skills) is the limit here: you can apply whatever effect you want!

If you give it a try, you'll see that the animation does not trigger when the element is first displayed, but only when it disappears/re-appears again. You can use the `appear` prop to also have the animation when the element first appears:

```
<Transition name="fade" appear>
  <div v-if="display">Content</div>
</Transition>
```

`Transition` also works with `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>
```

Here we have a "log in" or a "log out" button depending on `loggedIn`, a reactive property that's `true` if the user is logged in. The `fade` animation is applied to both buttons every time `loggedIn` changes. Which is... weird, because you get to see the two buttons at the same time, one slowly disappearing, while the other slowly appears.

That's where the `mode` prop of `Transition` can help. You can give it three possible values:

- `default`, which is the default obviously, and has the behaviour explained above.
- `in-out`, which first triggers the entering animation, then the exiting animation. In our case, that means you'll see the new button to show appears, then the other button disappears. Still not ideal.
- `out-in`, which first triggers the exiting animation, then the entering one.

`out-in` is what we need here: the current button will slowly disappear, then when the transition is done, the new button will slowly appear! This is the mode you want in most cases when you work with `v-else`.

25.3. List transitions

It's also possible to animate `v-for`! This time, we'll use `TransitionGroup` to wrap our `v-for`. You'll need to add a `:key` on your `v-for` element for it to work (but that's a good practice anyway). Unlike `Transition`, `TransitionGroup` renders to a `span`, but you can customize it to another element if needed, using its `tag` prop:

```
<TransitionGroup name="list" tag="ul" appear>
  <li v-for="pony of ponies" :key="pony.id">{{ pony.name }}</li>
</TransitionGroup>
```

It's then fairly similar to `Transition`, as it applies similar classes. We can define a `list` animation, where the elements slides in and out:

```
.list-leave-active,
.list-enter-active {
  transition: 1s;
}

.list-enter-from {
  transform: translateX(100%);
}

.list-leave-to {
  transform: translateX(-100%);
}
```

When a pony is added to the collection, then a new `li` element will slide in from the right into the list!

`TransitionGroup` also adds a `v-move` class, when the element changes position:

```
.list-move {
  transition: transform 1s ease;
}
```

If the list is shuffled for example, you'll see the elements sliding to their new positions!

25.4. And more!

It is also possible to write animations in pure JavaScript if you have a more complex use-case, for example using GSAP. Vue can still help here, as `Transition` also emits events:

- `@beforeEnter`/`@enter`/`@afterEnter`
- `@beforeLeave`/`@leave`/`@afterLeave`
- `@enterCancelled`/`@leaveCancelled`

`Transition` also works well with the router, if you want to animate a transition between two views.

You could think that testing a component with animations will be painful, as everything is going to be delayed until the animation ends. But don't worry! Vue Test Utils stubs the transition components, so everything works as if there were no transition 😮.

Try our exercise Animations and transition effects 🐴! You'll add some nice animations and transitions to Ponyracer.

Chapter 26. Advanced component patterns

Sometimes a simple component just won't cut it. Let's see what we can do for more advanced use-cases.

26.1. Template references with `ref`

With modern JavaScript frameworks, we usually don't manipulate the DOM ourselves, and let the framework do the work for us.

But, sometimes, we need to grab a reference to a DOM element. Imagine that we want to focus an input element for example. To do so, we want to call the native `focus()` method on the input. But first, we need to grab a reference to the element.

This can be achieved using `ref()`. Yes, it is the same `ref()` function that we use in the reactivity system, but for a slightly different usage.

Let's start by creating the reference in our script section:

Register.vue

```
const nameInput = ref<HTMLInputElement | null>(null);
```

Then use the name of this property (`nameInput`) as the value of the `ref` attribute of the HTML input in the template:

Register.vue

```
<input id="name" ref="nameInput" v-model="name" />
```

The type of `nameInput` is `Ref<HTMLInputElement | null>`, because its value is only available once Vue has generated the DOM and populated the reference. When can we use `nameInput` then? In the `onMounted()` lifecycle hook!

```
onMounted(() => {
  nameInput.value?.focus();
});
```

Note that this use case can also be solved with a custom directive, as we'll see in the next chapter.

This pattern is really useful in some cases. For example, when you want to insert a chart or a map in a component, you often have to use a third-party library, which doesn't know about Vue. In that case, simply grab the DOM reference you need and feed it to the third-party library.

This is exactly what we showcase in the Pro Pack 🧆. Try our exercise Charts in your app 🦄 to learn how to use template refs to integrate a third-party library.

26.2. Component references

This pattern also works with Vue components! Let's say you have a dropdown component (yours, or a third-party one, it doesn't matter).

This dropdown component has a function to open/close the dropdown.

Dropdown.vue

```
const opened = ref(true);
function toggle() {
  opened.value = !opened.value;
}

defineExpose({
  toggle
});
```

As the component is defined with the `script setup` syntax, the function has to be exposed with `defineExpose` to be visible outside the component.

This dropdown component can be used like this:

Navbar.vue

```
<Dropdown ref="dropdown">
  <button>First choice</button>
  <button>Second choice</button>
</Dropdown>

<button class="toggle" @click="toggleDropdown()">Open the dropdown</button>
```

You can see below that the dropdown has a `ref` called `dropdown`. This `ref` is declared in the script section, and can be used to access the dropdown component instance and its methods.

Navbar.vue

```
const dropdown = ref<typeof Dropdown | null>(null);
function toggleDropdown() {
  dropdown.value?.toggle();
}
```

In this case, the `dropdown` ref is not referencing an HTML element, it is referencing a Vue component.

This is a very handy pattern in certain cases.

Chapter 27. Custom directives

Vue offers a bunch of directives that you can use in your templates (`v-if`, `v-for`, `v-model`, etc.), but you can also build your own!

This can be very handy to encapsulate a behavior that you want to reuse several times, but that is not really a component (because it has no template) nor a composable (because its purpose is to manipulate the DOM).

For example, in the previous chapter, we saw how to focus an element in the template thanks to template references. This code works perfectly, but we would have to duplicate it every time we want to focus an element.

How can we do better? With a directive!

To define a directive, you create an object that can have several properties, corresponding to the directive lifecycle hooks you want to use.

27.1. Lifecycle hooks

A directive has the same lifecycle hooks as a component. But, instead of using functions like `onMount`, `onUnmount`, etc. in a `setup`, we are going to use properties like `mounted`, `unmounted`, etc. These hooks are called with the element they are applied on as the first parameter.

So our focus directive is as simple as:

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();
  }
};
```

We can then use the directive as follows:

Login.vue

```
<input v-focus name="login" />
```

As for components, you have to declare the directive in the script part. If you are using the `script setup` syntax, you simply have to import it. That's why the name matters: this is how Vue understands what is `v-focus` in the template.

Login.vue

```
import { vFocus } from '@/directives/Focus';
```

If you don't use `script setup`, then you have to declare the directive in the `directives` option of the component.

27.2. Directive value

You may have noticed that some Vue directives can have values, for example `v-model="user.name"`. Some can have an argument like `v-bind:src` or `v-on:click`. And some can be used with modifiers, like `v-model.number`.

Our custom directives can have a value, an argument and modifiers too!

We can imagine a really fancy `v-focus` directive that would have all of these! For example, it can have a value to indicate if the element should be focused or not. It would be used like this:

Login.vue

```
<input v-focus="shouldFocus" name="login" @blur="shouldFocus = false" />
```

Each lifecycle hook can receive a second parameter, usually called `binding`. `binding` is an object that contains the value, the parameters and the modifiers. So the directive would be defined like this:

Focus.ts

```
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();
    }
  }
};
```

The directive uses two lifecycle hooks: `mounted` and `updated`. The first one is called when the element is mounted in the DOM. The second one is called when the containing component has re-rendered.

As it is a fairly common use case, Vue allows defining the directive as a function, that will be called on mount and on update:

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();
  }
};
```

This is the form we will use in the rest of the chapter.

27.3. Directive argument

A directive can also have an argument. For example, the `v-bind` directive can have an argument to indicate the attribute to bind with `v-bind:src`. The `v-on` directive can have an argument to indicate the event to listen to with `v-on:click`. Our custom directives can have arguments too!

We can imagine a `v-focus` directive that would have an argument to indicate a delay before focusing the element. For example, it can be used like this:

Login.vue

```
<input v-focus:300="shouldFocus" name="login" @blur="shouldFocus = false" />
```

The argument is available in the `binding` object, in the `arg` property. It is a string, so we have to convert it to a number in our case. So the directive can be defined like this:

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. Directive modifiers

Last but not least, a directive can have modifiers. For example, the `v-model` directive can have a `number` modifier to indicate that the value should be parsed as a number. Our custom directives can have modifiers, as you guessed.

We can imagine a `v-focus` directive that would have a `seconds` modifier to indicate that the focus delay is in seconds (and not in milliseconds) for example. It can be used like this:

Login.vue

```
<input v-focus.seconds:2="shouldFocus" name="login" @blur="shouldFocus = false" />
```

The modifiers are available in the `binding` object, in the `modifiers` property. If a modifier is present, then the value of the property is `true`.

So the directive can be defined like this:

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);
  }
};
```

We have a nice use case in the Pro Pack, where you build a custom directive to add classes on form inputs depending on their validation. Try our exercise Custom directive 🦄 to learn how to build it.

Chapter 28. Internationalization

Alors comme ça, tu veux internationaliser ton application?

OK, don't worry if you didn't understand anything of this French introduction. Your role as a developer, fortunately, is not to translate your application into French, Spanish, or whatever other language. What you can do, though, is to allow this to happen. This chapter explains how to achieve that.

Vue itself doesn't do much for the internationalization part. But a plugin of the ecosystem is here to the rescue: vue-i18n.

vue-i18n helps on two topics:

- translating text
- formatting numbers and dates

28.1. vue-i18n setup

vue-i18n is a plugin that you can instantiate with `createI18n()`. The plugin allows defining the locale of the application, and the translations you want to use.

These translations are usually defined in a dedicated file per language. vue-i18n supports several formats for these files, like YAML or JSON. The messages can also be declared or overridden in a component, either in TypeScript, or in a dedicated `<i18n></i18n>` section of a SFC.

Even if it's not the nicest format, JSON has a nice advantage: it can be understood by TypeScript and provides type-safety.

src/en.json

```
{
  "chart": {
    "title": "Score history",
    "legend": "Legend",
  }
}
```

This is tremendously helpful, as it provides auto-completion in your IDE when working with the message keys (`chart.t` gets autocompleted into `chart.title`). TypeScript will also fail the build if a language file is missing one or several keys!

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
  }
});
```

You can then use the plugin in your application:

main.ts

```
createApp(App)
  .use(i18nPlugin)
  .mount('#app');
```

We're now ready to translate our texts!

28.2. Translating text

vue-i18n follows the pattern we've seen many times: it offers a composable, called `useI18n()` that we can use in our components. The composable lets us grab the `t()` function and use it in our code or template to translate text.

For example, we can use the `chart.title` key we defined in the messages to display the title:

ScoreHistory.vue

```
const { t } = useI18n<{ message: Message }, 'en' | 'fr'>();
const title = ref(t('chart.title'));
```

You will more often use `t` directly in the templates:

ScoreHistory.vue

```
<div id="legend">{{ t('chart.legend') }}</div>
```

The function will get the text related to the key from the messages of the current locale and display it.

28.3. Message parameters

You sometimes want to have parameters in your messages. For example, we'd like to display the login of the user in a message:

src/en.json

```
{
  "chart": {
    "title": "Score history",
    "legend": "Legend",
    "user": "Score for user { login }",
  }
}
```

`t` then accepts an object as a second parameter:

ScoreHistory.vue

```
<div id="score">{{ t('chart.user', { login: 'cedric' }) }}</div>
```

28.4. Pluralization

It is also possible to handle pluralization with `|`:

```
{
  "chart": {
    "title": "Score history",
    "legend": "Legend",
    "ponies": "no ponies | one pony | {n} ponies"
  }
}
```

You can then give the number of elements to display to `t`:

```
<div id="ponies">{{ t('chart.ponies', count) }}</div>
<!-- Displays 'no ponies' if count is 0, '3 ponies' if count is 3 -->
```

28.5. Changing the locale

When you support several languages in your application, you probably want to provide a way to switch the locale used in the application (or read the user preference on startup).

You can grab the current locale and the available ones thanks to `useI18n()`:

Navbar.vue

```
const { locale, availableLocales } = useI18n();
```

`locale` is a `ref`, so you can update its value, and Vue will refresh the templates to reflect it.

You can for example add a simple `select` to your application to let the user switch the locale:

Navbar.vue

```
<select v-model="locale">
  <option v-for="availableLocale in availableLocales" :key="availableLocale"
:value="availableLocale">
    {{ availableLocale }}
  </option>
</select>
```

Note that `t` automatically refreshes the translations in the templates (as the function re-evaluates). But not in your code: the function is executed once during the setup. If necessary, you can watch the locale to trigger a refresh of the translation though:

ScoreHistory.vue

```
watchEffect(() => (title.value = t('chart.title')));
```

28.6. Formatting

vue-i18n also helps with the formatting of dates with `d(Date|string|number)`, and numbers with `n(number)`. Both functions are based on the browser `Intl` support, and allow specifying a pattern as a second parameter if needed.

To grab these functions, you can use `useI18n` as usual:

```
const { d, n, availableLocales, locale } = useI18n();
```

and then use them in templates:

```
<div id="default-date">{{ d('2020-09-18') }}</div>
<!-- 9/18/2020 in English, 18/09/2020 in French -->
<div id="default-number">{{ n(2010.983) }}</div>
<!-- 2,010.983 in English, 2 010,983 in French -->
```

28.7. Other features (lazy-loading, Vite support and more)

vue-i18n comes with some nice additional features:

- advanced formatting options for numbers, dates and currencies
- a custom `v-t` directive that can be used instead of the `t()` function. The custom directive offers some optimizations as it allows to pre-compile the translations. But it is less flexible and necessitates a custom build.

- lazy-loading is possible to only fetch the translations when you need them
- Vite support via a plugin

You should feel ready to start internationalizing your application now!

To get started and learn a few more tricks, try our exercise Internationalization 🦄! It's part of our Pro Pack, and you'll learn how to internationalize a full application.

Chapter 29. Under the hood

I have a confession to make: I particularly enjoy learning how things *really* work. There are a lot of JS frameworks out there, but very few people know what the differences are between them under the hood.

If you're curious to learn how Vue *really* works, follow me!

29.1. Rendering changes

All JS frameworks face a very similar problem: when the state of the application changes, the framework needs to re-render the page to display the changes. It can't re-render the whole page though: it would be an incredible waste since, most of the time, just a few DOM nodes really need to be updated.

There are currently two common solutions to this problem. The popular frameworks you know use one or the other.

The first solution is to compile the HTML templates into JavaScript code that creates DOM elements, and modifies them when the state of the component changes.

In pseudo-code, that means that a template of a `Greetings` component looking like:

Greetings.vue

```
<div>Hello {{ user.name }}</div>
<input type="checkbox" :value="user.isAdmin" @change="updateAdminRights()">
```

generates something like the following code to create the component:

```
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);
}
```

and a function to update the DOM when the state of the component changes:

```
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;
  }
}
```

This is basically what you would write if I told you to create an application with *only* JavaScript and no HTML.

This is what framework like Angular or Svelte do: they compile all the components of your application to generate DOM instructions, and ship this result to the users.

Vue takes a different approach, popularized by React: the Virtual DOM.

The idea is very similar but with a twist: the framework also converts the template to JavaScript, but this time acts like the whole application is rendered every time. But it isn't of course: the changes generate a Virtual DOM, an in-memory representation of the DOM. The framework then compares it to the previously rendered DOM, and only applies the differences to the real DOM.

First, let's start by how Vue compiles our templates.

29.2. Template compilation

When Vue compiles the template of a component, it generates a `render` function for the component. This `render` function is called when Vue wants to re-generate the Virtual DOM after a state change.

So a `Greetings` component looking like:

Greetings.vue

```
<div>
   <span>Hello {{ user.name }}</span>
   <small v-if="user.isAdmin">admin</small>
</div>
<input type="checkbox" :value="user.isAdmin" @change="updateAdminRights()" />
```

would result in a `render` function added to our component.

The Vue compiler has 3 parts to achieve that:

- a parser
- several transformers
- a code generator producing the `render` function.

The parser goal is to read the template character by character to figure out what's going on. Basically, it goes like:

- `<`: Oh the start of an element! Let's parse it!
- `d`: OK the element name starts with `d`
- ...
- `>`: OK the element is a `div` and has no attributes, now let's check its children
- ...
- `{`: Is it the start of an interpolation?
- `{`: Yes! It is an interpolation! Let's parse it!
- ...
- `v`: This is the start of an attribute
- `-`: Oh this is in fact a special Vue directive!
- ...
- ':': Oh this is a Vue binding
- etc.

(Yes, in my head, a parser is very happy and always surprised)

If the parser encounters something weird, it throws an error. When it has read through the whole template, it generates an AST, an Abstract Syntax Tree, representing the 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.
  ]
```

```
};
```

When the parsing is done, the Vue compiler applies transformations on the AST. The transformers massage the AST for the codegen process, making it look like the structure of a program, with function calls and arguments. This is also the phase where each directive has their logic hooked in: the transform for `v-if` changes the AST to have an `if` condition.

When the transformations are done, we are very close to have code. The "codegen" phase then takes this tree representing a program and actually generates the code of the `render` function.

This `render` function is called every time Vue wants to re-generate the Virtual DOM representing our component.

29.3. Virtual DOM

Let's implement a Virtual DOM ourselves to get an idea of how it works.

We want to create a structure representing the DOM to create, very similar to the AST structure produced by the parser.

In our simplified version, let's say that a text node is represented by an object `{ type: 'text', text: 'Hello' }`, whereas an HTMLElement is represented by an object a bit more complex:

```
const input = {
  type: 'input',
  properties: { type: 'checkbox' },
  children: []
}
```

`children` is an array that can itself contain other elements. Both elements are called `VNode` in the following examples. We are making a distinction because when you create an element in the DOM you call `document.createElement` whereas for a text node you call `document.createTextNode` as we'll see later.

Let's start by writing two helper functions to create these 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 };
}
```

Then, using these helpers, the `render` function of our component, generated by the Vue template compiler, looks like:

```
/**
 * <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()
      },
      []
    )
  ]);
}
```

This function returns the structure representing the DOM as we want it: a Virtual DOM.

At runtime, Vue then takes this new structure, and compares it to the previously rendered one, the one representing the actually rendered DOM that it has stored, and generates the changes to really apply to the DOM.

Here is what our simple implementation does: it receives the root element of the application (where we mount the application), the previous `VNode` (the one currently rendered in the real DOM), and the new one, result of the `render` function we wrote.

```
function applyChanges(parent: ChildNode, index: number, previousNode: VNode | null,
newNode: VNode | null) {
}
```

Let's start with the first case: the new VNode is null, indicating that the corresponding DOM element must be removed. We could just remove it from the DOM, but, to make things simpler, we'll keep the same number of elements (and thus the same indices), and replace it with a comment:

```
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;
  }
}
```

On the contrary, the VNode we want to display might not exist in the previous version. In that case, the previous VNode is null, indicating that we need to create a new element and add it to the 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 = 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);
    }
  }
}
```

with `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;
}
```

The other case we need to handle is if the type of the element changed. In that case we create the new element, and replace the old one with the new one:

```
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);
  }
}
```

Then we can have a text node in the previous and current version, but with different text values. In

that case, we can just set the `textContent` to the new value:

```
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);
  }
  // text changes
  else if (isTextNode(previousNode) && isTextNode(newNode)) {
    // the previous node exists
    if (oldChild) {
      // update the text only if necessary
      if (previousNode.text !== newNode.text) {
        oldChild.textContent = newNode.text;
      }
    } else {
      // create a new text node
      const newTextNode = createChildElement(newNode);
      parent.appendChild(newTextNode);
    }
    return;
  }
}
```

We also want to handle property changes:

```
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);
  }
  // text changes
  else if (isTextNode(previousNode) && isTextNode(newNode)) {
    // the previous node exists
    if (oldChild) {
      // update the text only if necessary
      if (previousNode.text !== newNode.text) {
        oldChild.textContent = newNode.text;
      }
    } else {
      // create a new text node
      const newTextNode = createChildElement(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];
    }
  }
}
```

Last feature of our implementation: what do we do when children have changed? Well it's pretty easy: we iterate on them, and recursively call our `applyChanges` function:

```
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);
  }
  // text changes
  else if (isTextNode(previousNode) && isTextNode(newNode)) {
    // the previous node exists
    if (oldChild) {
      // update the text only if necessary
      if (previousNode.text !== newNode.text) {
        oldChild.textContent = newNode.text;
      }
    } else {
      // create a new text node
      const newTextNode = createChildElement(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]
      );
    }
  }
}
```

This is not perfect or optimized, but in ~50 lines of code we have a decent implementation of a Virtual DOM library!

Now when Vue detects that the state of the application changes, it calls `render` on our component, then call `applyChanges` with the previous `render` result and the current one. `applyChanges` updates the real DOM and Vue then stores the `render` result for next time:

```
// 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;
```

This is one of the core features of Vue. It offers:

- a template compiler (to generate the `render` function)
- a virtual DOM implementation to diff the changes

The Virtual DOM implementation was originally a fork of snabbdom but it has then been completely rewritten to fit the needs of Vue more closely.

29.3.1. The `render` function and the `h` helper

To explain how Virtual DOM works, we showcased a simplified `render` function. You can actually play with the Vue template compiler online on the Vue Template Explorer website to see what the "real" render function looks like.

The "real" render function is quite close to ours. Actually, Vue even lets you write it yourself if you

want. Vue exposes a helper function to create virtual nodes named `h` (keeping the same name that `snabbdom` was using).

In your components, instead of writing a template, you can directly add a `render` function that uses `h`. Or, if you use the Composition API, you can directly return the render function from your `setup`.

So our greetings component can be written:

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()
            }
          )
        ]
      );
  }
});
```

As you can see, this is understandable, but not really as easy to write as an HTML template. Nevertheless, it is a possible alternative, and a quite powerful one, as you can write any code you want.

And there is even an *other* alternative: JSX!

29.4. JSX

Coming soon

29.5. Reactivity

Now that we know how Vue *renders* state changes, let's see how Vue *detects* state changes.

To really understand what's going on, I'd like to write our own versions of the various reactive functions Vue offers: `reactive`, `ref`, `computed` and `watchEffect`.

Let's start with a simple example to illustrate how `watchEffect` work: we have a price and a quantity, and we want to log the total price automatically when one or the other changes.

```
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'
```

How can we rewrite `reactive` and `watchEffect` to make this example work? First, we need to know when we update the price and quantity. Do we have a way to "intercept" an update to these values in JavaScript? Yes, we have: with getters and setters!

29.5.1. getter/setter

JavaScript always had some meta-programming capabilities. The use of `get` and `set` is a good example.

```
const pony = { name: 'Rainbow Dash' };
console.log(pony.name);
// logs 'Rainbow Dash'
```

You can define a getter in your object, and do something every time the property is accessed. Even if, from the outside, it still looks like a simple property access:

```
const pony = {
  get name() {
    console.log('get name');
    return 'Rainbow Dash';
  }
};
console.log(pony.name);
// logs 'get name'
// logs 'Rainbow Dash'
```

But what if you want to do so on an existing object that you did not write yourself? Well, JavaScript offers `Object.defineProperty` since circa 2011:

```
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'
```

This is pretty cool, and lots of libraries and frameworks rely on this. Vue 2.x, for example, relied on this mechanism to trigger a re-rendering of the components displayed on a page, every time the state of one of the components is updated (this is a bit of a simplification but this is roughly the gist of it). To do so, Vue 2.x rewrote every property with a setter, that just sets the initial property, but also "warns" the framework that the property changed.

```
Object.defineProperty(component, 'user', {
  set(user) {
    this.user = user;
    heyVue2APropertyChanged(); // 🔥
  }
});
```

And it did this for every property declared when the component was initialized. This was cool, but did not cover some very simple use cases. For example, adding a property to an existing object after initialization:

```
component.newProperty = 'hello';
```

```
// won't call heyVue2APropertyChanged() 🙈
```

To support that case, Vue 2.x required the usage of `Vue.set(component.newProperty, 'hello')`, which was not really intuitive, but did the job.

`Object.defineProperty` only works with objects, as the name says, and not with other types, like arrays. So again, in Vue 2.x, you couldn't use `myArray[3] = 'hello'`, because Vue would not pick it up 🙈 (see the official documentation).

29.5.2. Proxies to the rescue

This is where proxies, a new JavaScript feature adopted in the ES2015 specification, can be handy. 🙌

Proxies are a general computer-science concept that describes an intermediary between a caller and a callee. In ES2015, a proxy can target objects, but also arrays, functions, or… other proxies (but not other built-in types like `Date`).

```
const user: UserModel = { name: 'Cédric' };
const handler: ProxyHandler<UserModel> = {};
const uselessProxy: UserModel = new Proxy(user, handler);
```

The handler can do some cool things, like *trapping* properties:

```
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'
```

There are a lot of different *traps* available: `get` and `set` of course, but also `has`, `apply`, `construct`, `defineProperty`, `deleteProperty`, etc.

For Vue 3.x, this represents a very interesting opportunity, as proxies solve the issue of dynamically added properties:

```
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'
```

That's why Vue 3 uses Proxies instead of `Object.defineProperty` and does not need `Vue.set`. And as mentioned, it also works for arrays, so `myArray[3] = 'hello'` will also be picked up by a proxy (and Vue 3).

Note that proxies come with a performance cost. It's always a bit hard to measure and compare, but it is slower than `defineProperty` for setting a property for example. Vue does its best to only use them when needed though.

Let's go back to the matter at hand, and implement the reactive functions with our new knowledge of proxies.

29.5.3. Re-implement basic `reactive` and `watchEffect` functions

To recap: we want to re-evaluate the following effect every time the reactive value is updated.

```
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'
```

We can set up a very basic mechanism that stores all the watchers, and re-evaluates them every time a reactive value is updated.

Let's start by implementing the `watchEffect` function. It just evaluates the given effect, and stores it in a global array:

```
type Effect = () => void;
const effects: Array<Effect> = [];
export function watchEffect(effect: Effect): void {
  effects.push(effect);
  effect();
}
```

Now we want to re-evaluate all the effects when a reactive value is updated. So let's implement `reactive` as a function that returns a proxy wrapping the given object. The proxy *traps* updates, and triggers the re-evaluation:

```
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];
    }
  });
}
```

And our simple example works!

```
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'
```

To be closer to reality, we'd like to handle updates to nested objects, as Vue does:

```
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'
```

To do so, we need to recursively transform nested objects to reactive ones (if not already transformed) when they are accessed:

```
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 that we execute the effect right away every time a value changes. Vue does not: it pushes the executions into a job queue, and then only executes them on the next "cycle" to avoid calling the same effect multiple times. That's why we needed the `await nextTick()` at the end of my first example, to flush the job queue.

We can implement that easily enough, by adding an array of jobs to execute, and only push the effect if it is not already there.

```
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;
}
```

Then we need to drain the queue job, by implementing a `nextTick` function:

```
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();
}
```

And here we go, it works!

```
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'!
```

There is still a pretty big limitation in our implementation: we trigger the evaluation of *all* the effects, every time we update *any* value, even if does not impact our effects:

```
// update unrelated reactive value
(state as any).discount = 0.2;
await nextTick();
// logs '27' again!
```

If Vue was doing the same, a complete application would quickly crumble under its own weight! We can probably do way better: we just need to trigger the evaluation of the effects impacted by the updated value. To do so, we are going to "track" the dependencies of the effect.

29.5.4. Dependency tracking

The plan is to register the effects that need to run when a dependency is updated. For example, our effect depends on `price` and `quantity`. So we need to remember that every time we update one or the other, then we need to re-run this effect.

When `watchEffect` is called, we need to:

- store the effect as the current effect
- run the effect, to track its dependencies

```
let currentEffect: Effect | null = null;
export function watchEffect(effect: Effect): void {
  currentEffect = effect;
  effect();
  currentEffect = null;
}
```

We now want to add the current effect in the array of effects to run, for each dependency it has. When the effect runs for the first time, the reactive objects that it depends on are accessed. We thus have a very similar `reactive` function than we had, but with a twist in the `get` function of the proxy to track the dependency:

```
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;
}
```

The `set` part of the reactive proxy is similar to what we had, but we now only run the necessary effects:

```
// 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);
  }
});
```

And everything works!

29.5.5. Re-implement basic `ref` and `computed` functions

Our little example works, but we are doomed if we want to use a primitive value in an effect: if we later update the primitive value, the watcher won't run the effect again, as it only tracks the reactive objects.

That's why `ref` exists: by wrapping a primitive, it transforms it into a reactive object, that Vue can track.

Re-implementing `ref` is thus straightforward:

```
export interface Ref<T> {
  value: T;
}

export function ref<T>(value: T): Ref<T> {
  return reactive({ value });
}
```

As you can see, a `ref` is nothing more than a `reactive` object, with a `value` property.

What about `computed`? `computed` is like a "smart" (and read-only) ref: we need to re-run the getter function every time we want to read its value. So we just create an object with a value getter that re-computes the value:

```
export function computed<T>(getter: () => T): Ref<T> {
  return {
    get value() {
      return getter();
    }
  };
}
```

And we can now do pretty much everything that Vue does! 

```
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'
```

Of course, Vue does much more (to handle edge-cases, for performances reasons, etc.) but you now

know how it does it!

Chapter 30. Performances

Be careful with premature optimization. Always measure before and after. Beware of the benchmarks you find on the internets: it's pretty easy to make them say what the authors want.

Performances can mean a lot of things: speed, CPU usage (battery consumption), memory pressure…

Everything is not important for everybody: you have different needs if you are programming for a mobile website, an e-commerce platform, or a classic CRUD application.

Performances can also be split into different categories, that, once more, won't all matter to you: first load, reload, and runtime performances.

First load is when you open an application for the first time. Reload is when you come back to that application. Runtime performances is what happens when the application is running. Some of the following recommendations are very generic, and could be applied to any framework. We wrote them down because we think it's worth knowing. And because when you talk about performances, the framework is sometimes the bottleneck, but really (really) often not.

30.1. First load

When you load a modern Web application in your browser, a few things happen. First, the `index.html` is loaded and parsed by the browser. Then the JS scripts and other assets referenced are fetched. When one of the assets is received, the browser parses it, and executes it if it is a JS file.

30.2. Asset sizes

So the first tip is very obvious: be careful with your asset sizes!

The assets loading phase depends on how many assets you want to load. A lot will be slow. Big ones will be slow. Especially if the network is not that good, which happens more often than you think: you might test your application on an optical fiber connection, but some of your actual users might be in the middle of nowhere, using slow 3G. Here is what you can do.

30.3. Bundle your application

When you write your Vue application, you have imports all over the place, and your code is split across hundreds of files. But you don't want your users to load hundreds of files! So before shipping your application, you want to make a "bundle": group all the JavaScript files into one file.

Rollup (for Vite) or Webpack (for the CLI) jobs are to take all your JavaScript files (and CSS, and template HTML files) and build bundles.

They are not easy tools to master, but Vite and the Vue CLI do a pretty good job at hiding their complexity. If you don't use Vite or the CLI, you can build your application with Rollup or Webpack

directly, or you can pick another tool that may produce even better results. But be warned that this requires quite a lot of expertise (and work) to not mess things up, just to save a few extra kilobytes. I would recommend staying with Vite (ideally) or the CLI.

The team working on these tools are doing a very good job keeping up with the latest Vue, TypeScript and bundler releases.

30.4. Tree-shaking

Rollup and Webpack (or other tools you use) start from the entry point of your application (the `main.ts` file generated for you), then resolves all the imports tree, and outputs the bundle. This is cool because the bundle will only contain the files from your codebase and your third party libraries that have been imported. The rest is not embedded. So even if you have a dependency in your `package.json` that you don't use anymore (so you don't import it anymore), it will not end up in the bundle.

It's even a bit smarter than that. If you have a file `utils` exporting two functions, let's say `start()` and `stop()`, and then only import `start()` into the rest of the application, but never `stop()`, then the bundler only puts `start()` in the final bundle, and drops `stop()`. This process is called **tree-shaking**. And every framework and library in the JavaScript ecosystem is fighting hard to be tree-shakable! In theory, it means that your final bundle contains only what is really needed! But in practice, bundlers are a bit conservative, and can't figure out some stuff. For example, if you have a class `Pony` with two methods `eat` and `run`, but you only use `run`, the code of the `eat` method will be in the final bundle. So it's not perfect, but it does a good job.

30.5. Minification and dead code elimination

When your bundle has been built, the code is usually minified and dead code will be eliminated. That means all variables, method names, class names… are renamed to use a one or two characters name through the entire codebase. This is a bit scary and sounds like it could break things, but ESBuild or Terser have been doing a great job. They will also eliminate dead code that they can find.

30.6. Other assets

While the above sections were about JS specifically, your application also contains other assets, like styles, images, fonts… You should have the same concerns about them, and do your best to keep them at a reasonable size. Applying all kinds of crazy techniques to optimize your JS bundle sizes, but loading several MBs of images, wouldn't have a big impact on your page loading time and your bandwidth! As this is not really the scope of this ebook, I won't dig into this topic, but let me point out a great online resource by Addy Osmani about image optimization: Essential Image Optimization.

30.7. Compression

All the modern browsers accept a compressed version of an asset when they ask the server for it. That means you can serve a compressed version to your users, and the browser will unzip it before

parsing it. This is a must-do because it will save you tons of bandwidth and loading time!

Every server on the market gives the option of activating the compression of assets. Generally the first user to request an asset will pay the cost of the compression on the fly, and then the following ones will receive the compressed asset directly.

The most common compression algorithm used is GZIP, but some others like Brotli are also popular.

30.8. Lazy-loading

Sometimes, despite doing your best to keep your JS bundle small, you end up with a big file because your app has grown to several dozens of components, using various third party libraries. And not only will this big bundle increase the time needed to fetch the JavaScript, it will also increase the time needed to parse it and execute it.

One common solution to this problem is to use lazy-loading. It means that instead of having a big bundle of JavaScript, you split your application into several parts and tell the bundler to split it in several bundles.

The good news is that Vue (and its router) makes this task relatively easy to achieve. You can read our chapter about lazy-loading if you want to learn more.

Lazy-loading can vastly improve the loading time, as you can make the first bundle really small, with only what's needed to display the home page, and let Vue load the rest on demand when your user navigates to another part.

30.9. Server side rendering

I'd like to start by saying that this technique is for 0.0001% of you. Server side rendering (or universal rendering) is the technique that consists of pre-rendering the application on the server before serving it to the users. With this, when a user asks for `/dashboard`, she will receive a pre-rendered version of the dashboard, instead of receiving `index.html` and then letting the router do its job after Vue has finished to start.

It can lead to vast improvements in perceived startup time. Vue offers a built-in support that allows you to run the application not in a browser but on a server. You can then pre-render the pages and serve them to your users. The page will display very fast and then Vue will start its job and run as usual.

It's also a big win if you want your web site to be crawlable by search engines which don't execute JavaScript, since you can serve them pre-rendered pages, instead of a blank page.

It's also a way to display previews of your website on social networks like Twitter or Facebook. These sites will try to screenshot the shared URL, but since they don't execute JavaScript, they won't see anything of your dynamically generated page, unless you serve them a page generated on the server. So if you want to be sure that the preview is perfect, like if you are running a news site, or an e-commerce site, you need to add server-side rendering.

The bad news is that it can increase the complexity of your application. You need to setup your

server and think about the strategy you want to adopt. Do you want to pre-render all pages or just a few? Do you want to pre-render the whole page, with the data fetching and authorization check it will need, or just some critical parts of the page? Do you want to pre-render them on build, or to pre-render them on demand and cache them? Do you want to do this for all the possible profiles and languages or just some? All these questions depend on the type of application you are building, and the effort can vary greatly depending on your goal.

So, again, I would advise you to use server side rendering only if it is critical for your application, and not based on the hype…

30.10. Caching for reloads

Once your user has opened the application once, it's possible to speed up the subsequent visits

You should always cache the assets of your application (images, styles, JS bundles…). This is done by configuring your server and leveraging the `Cache-Control` and `ETag` headers. All the servers on the market allow you to do so. You can also use a CDN for this purpose, which will additionally allow the users to download the assets from a server close to their location. If you do so, the next time your users open the application, the browser won't have to send a request to fetch the assets because it will have them already!

But a cache is always tricky: you need to have a way to tell the browser "hey, I deployed a new version in production, please fetch the new assets!".

The easiest way to do this is to have a different name for the asset you updated. That means instead of deploying an asset named `main.js`, you deploy `main.xxxx.js` where `xxxx` is a unique identifier. This technique is called cache busting. And, again, Vite and the CLI are there for you: in production mode, they will name all your assets with a unique hash, derived from the content of the file. They also automatically update the sources of the scripts in `index.html` to reflect the unique names, the sources of the fonts, the sources of the stylesheets, etc.

If you use Vite or the CLI, you can safely deploy a new version and cache everything, except the `index.html` (as this will contain the links to the fresh assets deployed)!

30.11. Runtime performances

Vue's magic relies on its reactivity system and VDOM rendering: the framework automatically detects changes in the state of the application and updates the DOM accordingly. So, as a general rule of thumb, you'll want to help Vue and limit the change detection triggering and the amount of DOM to update/create/delete.

To be honest, most applications will be fine, even under heavy load. Read the chapter about Vue under the hood if you want to get a grasp on how Vue works.

But some of us will have to recode Excel in the browser for their enterprise, or will have a component with a tree displaying 10,000 customers, or another unreasonable thing to do in a browser. These things are tricky, whatever framework you use. They tend to update a lot of DOM, and have to check a lot of components. A few of the following tricks can help.

30.12. `key` in `v-for`

This is a simple tip that can really speed things up on `v-for`: add a `key`. To understand why, let me explain how modern JS frameworks (at least all major ones) handle collections. When you have a collection of 3 races and want to display them in a list, you'll write something like:

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

When you add a new race, Vue will add a DOM node in the proper position. If you update the name of one of the ponies, Vue will change just the text content of the right `li`.

How does it do that? By keeping track of which DOM node references which object reference. Vue will have an internal representation looking like:

```
node li 1 -> race #e435 // { id: 3, name: London }
node li 2 -> race #8fa4 // { id: 4, name: Paris }
```

It works great, and if you change an object for another one, Vue will destroy the node and build another one.

```
node li 1 (recreated) -> race #c1ea // { id: 1, name: Berlin }
node li 2 -> race #8fa4 // { id: 4, name: Paris }
```

If the whole collection is updated with new objects, the complete DOM list will be destroyed and recreated. Which is fine, except when you just refresh a list with almost the same content: in that case, Vue destroys the complete node list and recreates it, even if there is no need to. For example, when you fetch the same results from the server, you will have the same content, but different references as your collection will have been recreated.

The solution for this use-case is to help Vue track the objects, not by their references, but by something that you know will identify the object, typically an ID.

For this, we add a `key`:

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

With this `key`, Vue will only recreate a DOM node if the id of the race changes. On a very big list which doesn't change much, it can save a ton of DOM deletions/creations. Anyway, it's quite cheap to implement and doesn't have cons, so don't hesitate to use it. It's also a requirement if you want to use animations. If a DOM element's style is supposed to be animated (by transitioning smoothly from the previous value to the new one), and the list of races is replaced by a new one when

refreshed, then `key` is a must: without it, the animation will never happen, because the style of the element never changes. Instead, it's the element itself which is being replaced by Vue.

30.13. v-memo

Vue 3.2 introduced another trick to help with performances: the `v-memo` directive, which allows you to aggressively optimize templates in some edge cases. You can think of `v-memo` as an equivalent of `shouldComponentUpdate` in React, but available for elements or components in Vue.

Let's say we have a template like this:

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

Without `v-memo`, a change in a property of one of the races, even a non-displayed one like its country, results in the creation of a new virtual `li` that will be compared by the virtual DOM algorithm to the previous one (check the chapter about how Vue works under the hood if you want to learn more about this).

With `v-memo`, the virtual element is not recreated, and the previous one is re-used, except when the conditions of `v-memo` (here, the name of the race) change. This may look like a small improvement, but it's actually a huge improvement in performance if you render a large list of elements. The popular (although debatable) js frameworks performance benchmark has a benchmark testing this particular use-case, and Vue v3.2 outperforms most frameworks with this new feature, without the need to create a dedicated component for the row and add tricks to improve the performances (like `shouldComponentUpdate` in React).

You can see that `v-memo` accepts an array of conditions, so you can write something like:

```
<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>
```

Then the `li` will be updated if either the race's name changes or if the selected race check has a different result.

Note that if your list displays a property not listed in the conditions of `v-memo`, the list won't be updated when this property changes:

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

```
</ul>
```

If the country of a race changes, the list won't update. If, later, the name of the race changes, then the updated list will display the new name and the new country.

This trick is really only useful for very large list, and you should not need it for more typical use-cases.

30.14. Conclusion

This chapter hopefully taught you some techniques which can help solve performance problems. But remember the golden rules of performance optimization:

- don't
- don't… yet
- profile before optimizing.

As a famous computer scientist said:

> premature optimization is the root of all evil.
>
> — Donald Knuth

So strive to make the code as simple and correct and readable as possible, and only start thinking about profiling, then optimizing, if you have a proven performance problem.

Chapter 31. This is the end

Thanks for reading!

We will add some new chapters in the future releases, covering more advanced stuff and some other goodies. They all need a little more polish, but I'm sure you'll enjoy them. And of course, we'll keep up with the new releases of the framework and libraries, so you won't miss the new shiny features that will come out. All these future updates of the book and online course will be available for free, of course!

If you liked what you read, tell your friends about it!

And if you don't already own it, you should know that there is also an online course (the Pro Pack) available with this ebook. This course gives access to a whole set of exercises to build a real application, step by step, starting from scratch. For each step we provide unit tests and e2e tests covering 100% of your code, detailed instructions (which are not a basic copy-paste, but will push you to understand what you are doing), and a solution if you need it (which should be elegant and respect the best practices). A home-brewed tool analyzes your code and computes a score for each exercise, and your progress is visible on a dashboard. If you're looking for actual code samples, always up-to-date, which might save you hours of work, our Pro Pack is waiting for you! You can even try the first exercises for free. And as you are already the proud owner of this ebook, we want to thank you for your historic support with a generous discount that you can grab here!

We have tried to give you all the keys, but Web Development looks an awful lot like:

How to draw a horse. Credit to Van Oktop.

So we also provide training, mainly in France and Europe, but all over the world really (and remote training is always a possibility). We can also do some consulting work to help your team, or work with you to help you build your product. Just shoot us an email at hello@ninja-squad.com, and we'll talk about it!

Overall, I would love hearing from you and find out what you liked, loved and hated in this ebook - whether you are writing to signal a small typo, a big mistake, or just to tell us that this book helped you find your dream job (well, you never know…).

I can't finish without thanking a few people. My girlfriend, first, who has been an incredible support, even when I was rewriting something for the tenth time, in a dreadful mood on a Sunday. My colleagues, for their tireless work and feedback, their kindness for encouraging me and giving

me the time to do this crazy thing. And my friends and family, for the little words that kept me going.

And you, for buying this and reading it to the last sentence ❤.

Stay tuned.

Appendix A: Changelog

Here are all the major changes since the first version. It should help you to see what changed since your last read!

By buying this ebook, you'll get all the following updates for free. Go to https://books.ninja-squad.com/claim to obtain the latest version of this ebook.

Current versions:

- Vue: `3.4.34`

A.1. Changes since last release - 2024-07-26

The templating syntax

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

Slots

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

A.2. v3.4.0 - 2023-12-29

How to build components

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

Forms

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

A.3. 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.4. 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.5. 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.6. 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.7. v3.2.26 - 2021-12-17

The templating syntax

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

Script setup

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

Composition API

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

State Management

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

Router

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

A.8. 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.9. 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.10. 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.11. v3.0.11 - 2021-04-02

A.12. 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.13. 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.14. 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.15. 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.16. 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.17. v3.0.0-beta.10 - 2020-05-11

Global

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