Become a

# NINJA

with

Vue.js

ninja squad

# Become a ninja with Vue (free sample)

Ninja Squad

# Table of Contents

# Chapter 1. Free sample

What you're going to read is a free sample of our Vue ebook: the starting part of the full book, which explains its purpose and contents, gives an overview of ECMAScript 2015+, TypeScript, and Web Components, describes Vue philosophy, and finally lets you write your first application.

This sample extract does not require any preliminary knowledge.

# Chapter 2. 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 3.5.13 for the examples.

# Chapter 3. 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!

## 3.1. 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!

## 3.2. 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!

## 3.3. 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
```

## 3.4. 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!');
    }
  };
}
```

## 3.5. 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.

## 3.6. 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
```

## 3.7. 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);
```

## 3.8. 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
```

## 3.9. 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.

## 3.10. 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!

## 3.11. 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.

## 3.12. 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.

## 3.13. 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.

## 3.14. 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.

## 3.15. 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 4. Going further than ES2015+

## 4.1. 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!

## 4.2. 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 5. 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.

## 5.1. 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
```

## 5.2. 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';
```

## 5.3. 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;
}
```

## 5.4. 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 };
```

## 5.5. 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;
}
```

## 5.6. 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);
```

## 5.7. 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!

## 5.8. 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 6. 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.

## 6.1. 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
```

## 6.2. 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.

## 6.3. 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.

### 6.3.1. 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`
```

### 6.3.2. 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
```

### 6.3.3. 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.

## 6.4. 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 7. 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.

## 7.1. 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.

## 7.2. 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.

## 7.3. 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.

## 7.4. 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);
  }

}
```

## 7.5. 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.

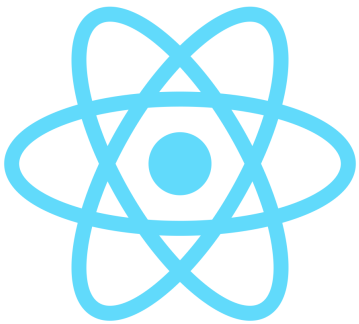
That's obviously far too short a chapter to tell you everything there is to say on Web Components, but you'll see that some concepts are going to pop out along your read. And you'll definitely see that Vue has been designed to make it easy to use Web Components with our Vue components. It is even possible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

# Chapter 8. Grasping Vue's philosophy

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

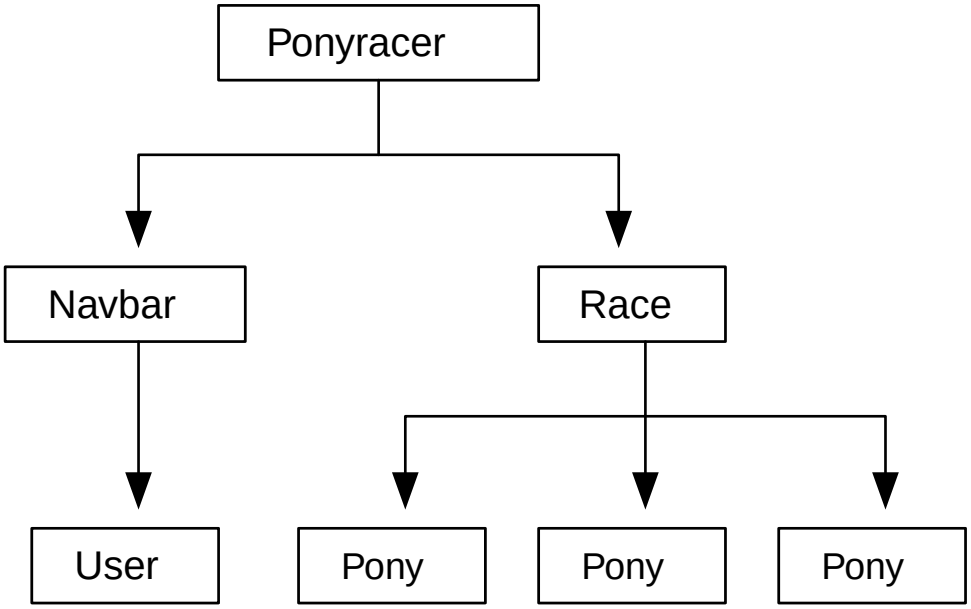
First and foremost, Vue is component-oriented. You will write tiny components and, together, they will constitute a whole application. A component is a group of HTML elements in a template, dedicated to a particular task. For this, you will usually also need to have some logic linked to that template, 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 9. 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.

## 9.1. 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@3.5.13`.

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.

## 9.2. 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.

## 9.3. 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.

## 9.4. 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.

## 9.5. 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 { 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"></script>

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

<style scoped></style>
```

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

## 9.6. 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 10. End of the free sample

That’s it! I hope that you enjoy this reading. If you want more of it (and you should), go buy it on the ebook website! :)

# 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: `3.5.13`

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

**From zero to something**

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

**Script setup**

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

**State Management**

- Remove Vuex section (2025-04-13)

**Advanced component patterns**

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

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

**The templating syntax**

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

**How to build components**

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

**Script setup**

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

**Slots**

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

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

**How to build components**

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

**Forms**

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

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

**The many ways to define components**

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

**Script setup**

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

**Forms**

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

**Slots**

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

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

**Global**

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

**Style your components**

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

**Router**

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

**Advanced component patterns**

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

**Custom directives**

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

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

**State Management**

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

**Internationalization**

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

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

**From zero to something**

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

**Style your components**

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

**Testing your app**

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

**Lazy-loading**

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

**Performances**

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

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

**The templating syntax**

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

**Script setup**

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

**Composition API**

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

**State Management**

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

**Router**

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

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

**The many ways to define components**

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

**Script setup**

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

**Suspense**

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

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

**Global**

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

**The many ways to define components**

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

**Performances**

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

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

**The templating syntax**

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

**Forms**

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

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

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

**Style your components**

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

**Provide/inject**

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

**State Management**

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

**Animations and transition effects**

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

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

**How to build components**

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

**Forms**

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

**Suspense**

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

**Router**

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

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

**Forms**

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

**Slots**

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

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

**Directives**

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

**Router**

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

**Slots**

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

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

**The wonderful world of Web Components**

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

**How to build components**

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

**Under the hood**

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

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

**Global**

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