Eric Sarrion

Vue.js Advanced Concepts

CONTENTS

WHY AN IN-DEPTH BOOK ON VUE.JS?

Lots of books deal with Vue.js, but in our opinion too superficial. Business projects are often complex projects, which require a technical environment with databases, external or internal APIs, etc.

Vue.js allows you to approach these areas, sometimes by coupling it with other libraries such as Axios to communicate in Ajax with a server, or Vuex to centralize the state of the application.

Also, in order to build more robust and maintainable projects, Vue.js can extend by creating new features such as components, directives and filters.

All these new features, and more, are studied in the following pages, certainly in depth, but above all gradually in order to allow you to carry out complete projects.

We will therefore talk about the following topics:

- The development of Vue components,
- The creation of directives and filters,

- Using Vue CLI,
- The use of Ajax,
- Using Vuex and Vue Router,
- The creation of mixins and plugins.

READING GUIDE

The order in which the chapters are written seems to us the right way to begin reading and learning this book.

Please note that Vue.js basics are not covered here. These are in particular:

- The notion of reactive variable;
- The notions of one-way and two-way binding;
- Event management;
- The use of calculated properties and watchers.

These concepts are necessary to understand the elements exposed here. If necessary, please refer to the *Vue.js Basic Concepts* book by the same author, which explains these basics.

1 – INTRODUCTION

This chapter introduces the basic tools to use Vue.js (also called Vue in a simpler way).

For this we use a basic index.html file, which will be enriched as we learn.

Basic index.html file

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="root"> Here will be displayed the HTML code generated by Vue
</div>
</body>
</html>
```

The HTML code for the Vue library is inserted using the vue.js JavaScript file located on the specified CDN server. The dist version shown here allows you to use the version of Vue for development mode, which displays error messages in the browser console if needed.

The <body> part includes a element which will

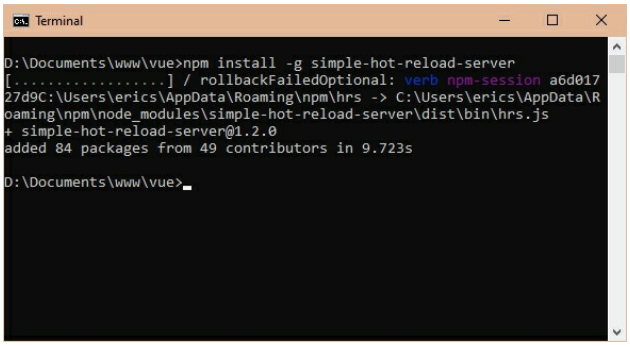
contain the HTML code generated by Vue, as we will see in the following sections.

Install a Hot Reload Server (allowing the automatic redisplay of the page)

First of all, let's install a server so that we don't have to refresh the HTML page displayed in the browser each time the page source is modified. There are many servers available on the Internet, and we will be using one of them, the *Hot Reload Server*. We install it from the following npm command:

Install the Hot Reload Server

```
npm install -g simple-hot-reload-server
```



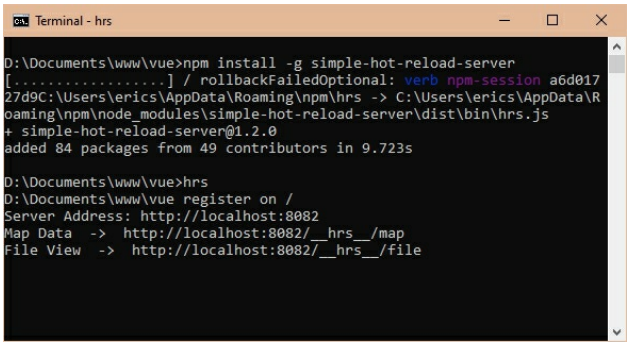
Once the server is installed, it can be started with the hrs command. The hrs -h command displays help for the command. The default port is 8082, which can be changed using the -p option in the command. The

directory used by default for the server is the one from which the command is launched, otherwise it suffices to indicate its path in the path parameter of the command (as indicated in the displayed help).

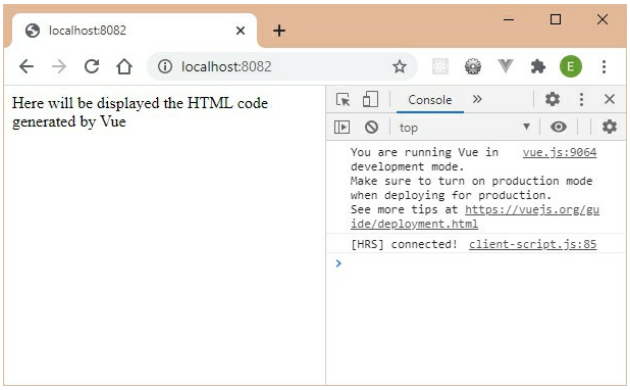
Let's start the server from the current vue directory, using the default options (port 8082).

Start the server that will use the files from the current vue directory

```
hrs
```



As noted, the server is available at the URL http://localhost:8082. Let's enter this URL in a browser, here Chrome.



Because of the use of a server allowing to

automatically reload the displayed page, any modification of the index.html file causes the page to be redisplayed in the browser without having to refresh the displayed page.

Install the Vue Devtools utility

This utility is an extension of Chrome or Firefox. Install it from your browser (here Chrome).

Pressing the F12 key displays the console and the Vue tab allowing you to debug the Vue application.

2 - CREATE COMPONENTS

Vue allows you to break down a global application into a set of components, which are then assembled to form the application. The advantage is to be able to split the application into several subsets managed independently.

The components created can also be reused in other parts of the application, or even in other applications.

We use the following structure of the HTML page, in which we create the div#root element which will contain the HTML elements of the template associated with the Vue object.

Basic index.html file

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="root"></div>
</body>
<script>
var vm = new Vue({
  el : "#root",
```

```
  template : `
    <div>
    </div>
  `
});
</script>
</html>
```

Create a Timer component with Vue.component()

A Vue component is created using the Vue.component(name, options) method:

- The name parameter is a character string corresponding to the name of the component,
- The options parameter is an object describing the options for creating the component. We will see that many of these options are the ones used when creating the Vue object (by new Vue()).

Let's create a Timer component to display the remaining time, in the form MM:SS. It is assumed that this timer is initialized at 10 seconds, that is to say 00:10.

Create a Timer component displaying 00:10

```
Vue.component("Timer", {
  data() {
    return {
        min : 0,
```

```
    sec : 10
   }
  },
  computed : {
   time() {
    var min = this.min;
    var sec = this.sec;
    if (min < 10) min = "0" + min;
    if (sec < 10) sec = "0" + sec;
    return min + ":" + sec;
   }
  },
  template : `
    <div>
      Remaining time {{time}}
   </div>
  `
});
```

The Vue.component() method uses almost the same options that were used when creating the Vue object:

- The data property is replaced by a function that returns reactive variables. This makes it possible to use the Timer component several times without these reactive variables being the same for all the components. Thanks to the data() function, each variable defined (here min and sec) will be specific to each component.
- Use the computed time property to display the time as MM:SS.
- The template property represents the template

that will be displayed when using this component.

We can therefore see that the creation of a Vue component is very similar to what we did previously (see the *Vue.js Basic Concepts* book from same author). The only difference for now is:

- The data property which is used as a function which returns the reactive variables,
- The non-use of the el property in the component. Indeed, this property is used when creating the Vue object, which we will see below.

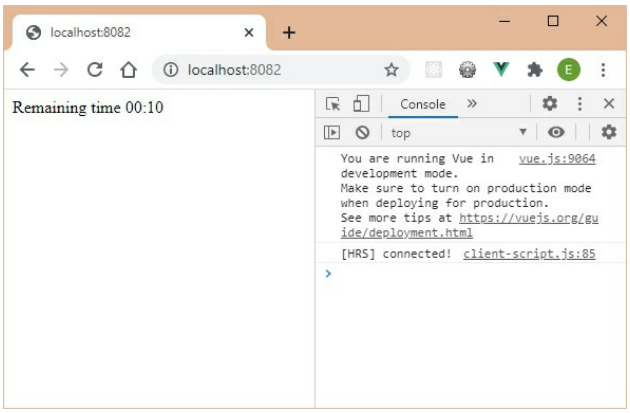
We have seen the definition of the Timer component using the Vue.component() method. It remains to use this component in an HTML page.

Use the Timer component

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="root"></div>
</body>
<script>
Vue.component("Timer", {
  data() {
    return {
        min : 0,
      sec : 10
    }
```

```
  },
  computed : {
    time() {
      var min = this.min;
      var sec = this.sec;
      if (min < 10) min = "0" + min;
      if (sec < 10) sec = "0" + sec;
      return min + ":" + sec;
    }
  },
  template : `
     <div>
        Remaining time {{time}}
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer />
    </div>
  `
});
</script>
</html>
```

The Vue object is created as before, by indicating in the el property the DOM element of the page in which we will insert the indicated template. The template integrates the component by using it in XML form, here <Timer /> or also <Timer></Timer>.



Bring the component to life

The previous Timer component remains frozen at the initial values, therefore at 00:10. The seconds and minutes should be reduced until 00:00.

To do this, all you have to do is trigger a timer with the setInterval() function of JavaScript. This function can be integrated into the created() method called during the creation of each component.

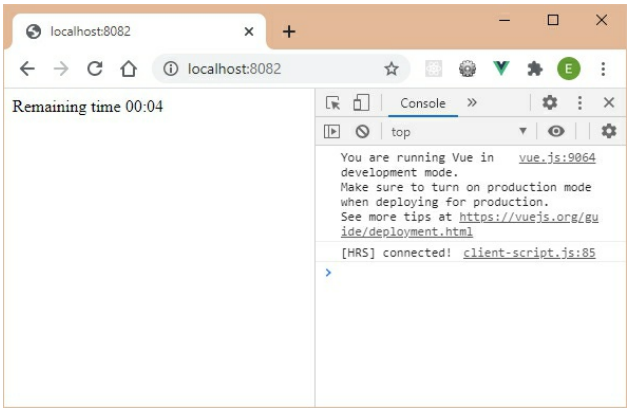
Use the created() method in the component

```
Vue.component("Timer", {
  data() {
    return {
        min : 0,
      sec : 10
    }
  },
  computed : {
    time() {
      var min = this.min;
      var sec = this.sec;
      if (min < 10) min = "0" + min;
```

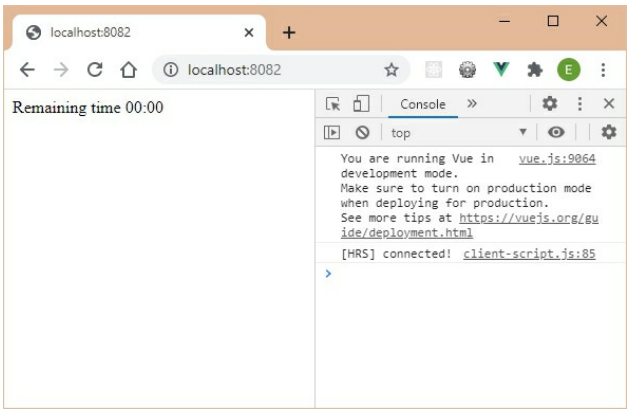
```
    if (sec < 10) sec = "0" + sec;
    return min + ":" + sec;
   }
  },
  created() {
   var timer = setInterval(() => {
    this.sec -= 1;
    if (this.sec < 0) {
     this.min -= 1;
     if (this.min < 0) {
      this.min = 0;
      this.sec = 0;
      clearInterval(timer);
     }
     else this.sec = 59;
    }
   }, 1000);
  },
  template : `
    <div>
      Remaining time {{time}}
   </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
   <div>
    <Timer />
   </div>
  `
});
```

The created() method calls the callback function specified in setInterval (callback) every second. The

processing of the callback function consists in decrementing the variable sec then, if necessary, the variable min, until reaching 0 for both. In this case, the timer is stopped by the clearInterval() function.



The timer starts and the seconds decrease until it reaches 00:00.



Pass properties to the component

The Timer component is currently initialized at 00:10. It would be good to be able to indicate in parameters the minutes and seconds that you want to count down.

The Timer component would then be used in the following form:

Use the Timer component by initializing the starting minutes and seconds

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer minutes="1" seconds="10" />
    </div>
  `
});
```

We now indicate in the attributes of the Timer component:

- The minutes attribute to indicate the initial number of minutes,
- The seconds attribute to indicate the initial number of seconds.

> Pay attention to the names of the attributes! If we use min and sec for the attribute names, Vue will confuse them with the names of the reactive min and sec variables, which will cause an error.

The attributes of a component are called the properties of the component. In short, we also say "props".

The Timer component must now use these two attributes. We indicate them in the props section of the

component. This props section is used to describe the names of the component attributes, in the form of an array of strings.

Describe the props in the component's props section

```
Vue.component("Timer", {
 data() {
  return {
      min : 0,
    sec : 10
  }
 },
 props : [
    "minutes",
  "seconds"
 ],
 computed : {
  time() {
    var min = this.min;
    var sec = this.sec;
    if (min < 10) min = "0" + min;
    if (sec < 10) sec = "0" + sec;
    return min + ":" + sec;
  }
 },
 created() {
  var timer = setInterval(() => {
    this.sec -= 1;
    if (this.sec < 0) {
      this.min -= 1;
      if (this.min < 0) {
        this.min = 0;
        this.sec = 0;
        clearInterval(timer);
```

```
        }
        else this.sec = 59;
      }
    }, 1000);
  },
  template : `
     <div>
       Remaining time {{time}}
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer minutes="1" seconds="10" />
    </div>
  `
});
```

Once the names of the props are indicated in the props section, our component should use the values that are passed to it (it currently uses the values indicated in the data section). The min and sec variables should therefore no longer be initialized with the values 0 and 10, but rather with the values of the props transmitted.

Vue allows you to know the values transmitted in the props using this.minutes and this.seconds. One thus initializes the reactive variables min and sec with these values.

Use the values of the props in the component

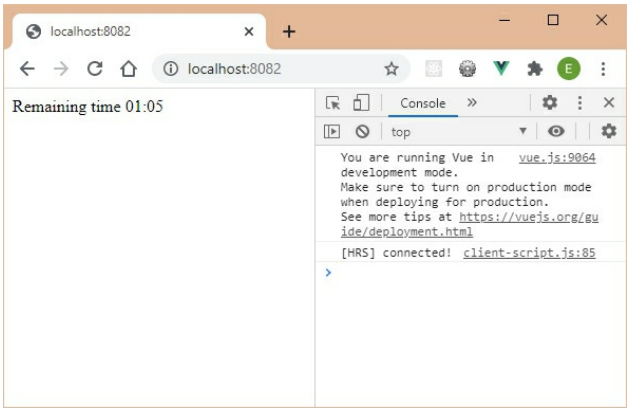
```
Vue.component("Timer", {
 data() {
  return {
    min : this.minutes,
    sec : this.seconds
  }
 },
 props : [
   "minutes",
  "seconds"
 ],
 computed : {
  time() {
    var min = this.min;
    var sec = this.sec;
    if (min < 10) min = "0" + min;
    if (sec < 10) sec = "0" + sec;
    return min + ":" + sec;
  }
 },
 created() {
  var timer = setInterval(() => {
    this.sec -= 1;
    if (this.sec < 0) {
      this.min -= 1;
      if (this.min < 0) {
        this.min = 0;
        this.sec = 0;
        clearInterval(timer);
      }
      else this.sec = 59;
    }
  }, 1000);
 },
 template : `
   <div>
```

```
      Remaining time {{time}}
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer minutes="1" seconds="10" />
    </div>
  `
});
```

The reactive variables min and sec are now initialized with the values of the transmitted props.

> We see here why the names of the props should not be the same as those of the reactive variables defined in the data section. Indeed they are both used by prefixing them with this, hence the confusion if they are the same names.

The timer is decremented from the values indicated in the props.



Display a message when the timer has reached 00:00

Rather than leaving the display at 00:00 when the timer has expired, let's display the message "End of timer" when this is the case.

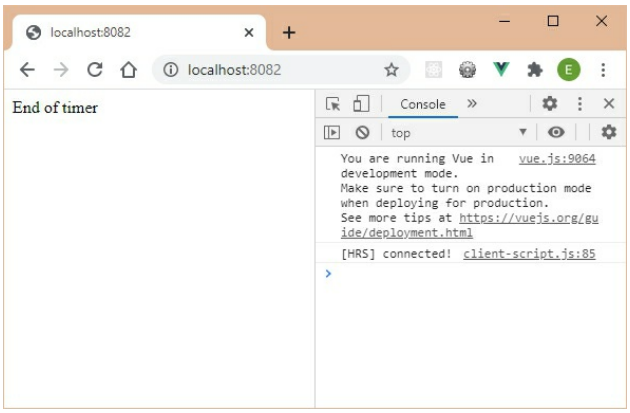
The component template is modified using the v-if and v-else directives.

Display an "End of timer" message

```
Vue.component("Timer", {
  data() {
    return {
        min : this.minutes,
      sec : this.seconds
    }
  },
  props : [
     "minutes",
    "seconds"
  ],
  computed : {
    time() {
      var min = this.min;
      var sec = this.sec;
      if (min < 10) min = "0" + min;
      if (sec < 10) sec = "0" + sec;
      return min + ":" + sec;
    }
  },
  created() {
    var timer = setInterval(() => {
```

```
      this.sec -= 1;
      if (this.sec < 0) {
        this.min -= 1;
        if (this.min < 0) {
          this.min = 0;
          this.sec = 0;
          clearInterval(timer);
        }
        else this.sec = 59;
      }
    }, 1000);
  },
  template : `
     <div>
      <div v-if="sec||min"> Remaining time {{time}} </div>
      <div v-else>End of timer</div>
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer minutes="0" seconds="5" />
    </div>
  `
});
```

When the sec and min variables are both at 0, the message is displayed, otherwise the remaining time is displayed.



Nesting components

Consider the previous Timer component. We want to associate it with a Start button to start the timer:

- The Start button turns into a Stop button as soon as the timer is started, so that it can be stopped at any time.
- If you press Stop, the button redisplays Start so that it can be restarted.

To explain the operating principle more easily, it is assumed that the timer displays for the moment "00:10", without the possibility of modifying the time remaining every second. The code of the Timer component is kept to a minimum, with only the template showing the result!

Timer component (minimized)

```
Vue.component("Timer", {
  template : `
     <div>
```

```
    00:10
   </div>
  `
});
```

The Timer component therefore displays 00:10 continuously.

Now let's write the StartStop component corresponding to the Start button becoming Stop when it is clicked (and vice versa).

StartStop component corresponding to the Start / Stop button

```
Vue.component("StartStop", {
  data() {
    return {
      stopped : false
    }
  },
  template : `
     <div>
      <button v-if="stopped" v-
on:click="stopped=!stopped">Start</button>
      <button v-else v-on:click="stopped=!stopped">Stop</button>
    </div>
  `
});
```

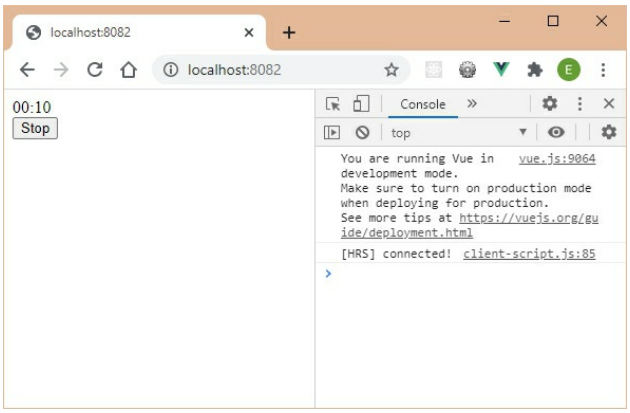
The StartStop component has a reactive stopped variable indicating whether the timer is stopped (true) or not (false). This variable is modified by clicking on the button, which thus allows the button caption to be modified.

It now remains to use the two components Timer and StartStop. We create the two components in the template of the final Vue object:

Use the Timer and StartStop components

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="root"></div>
</body>
<script>
Vue.component("StartStop", {
 data() {
   return {
     stopped : false
   }
 },
 template : `
    <div>
     <button v-if="stopped" v-on:click="stopped=!stopped">Start</button>
     <button v-else v-on:click="stopped=!stopped">Stop</button>
   </div>
 `
});
Vue.component("Timer", {
 template : `
    <div>
     00:10
   </div>
 `
});
var vm = new Vue({
```

```
  el : "#root",
  template : `
    <div>
      <Timer />
      <StartStop />
    </div>
  `
});
</script>
</html>
```



We check by clicking on the Start button that it becomes Stop, then Start, etc. Everything seems to work (except that the timer does not start, but this is normal given the minimal code we wrote).

Let's use another timer below the first one. The code of the Vue object becomes:

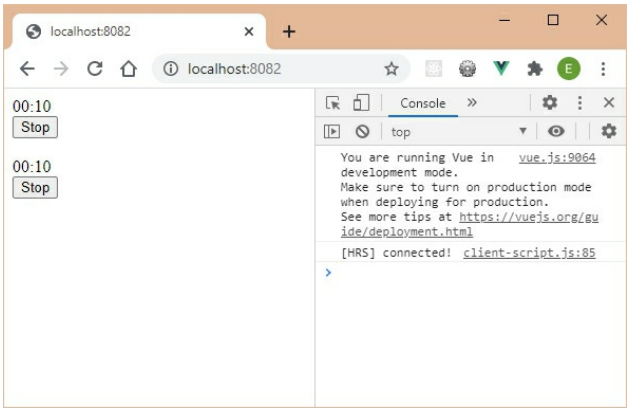
Use two timers in the page

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer />
```

```
    <StartStop />
    <br>
    <Timer />
    <StartStop />
  </div>
 `
});
```

We use two Timer components and two StartStop components, each associated with the one with which it is contiguous in the template.

We now display:



The problem, even if for the moment everything is going well, is that we have to explain the use of these two components Timer and StartStop, by explaining that the StartStop component manages the Timer component which precedes it (and especially not the one following it!).

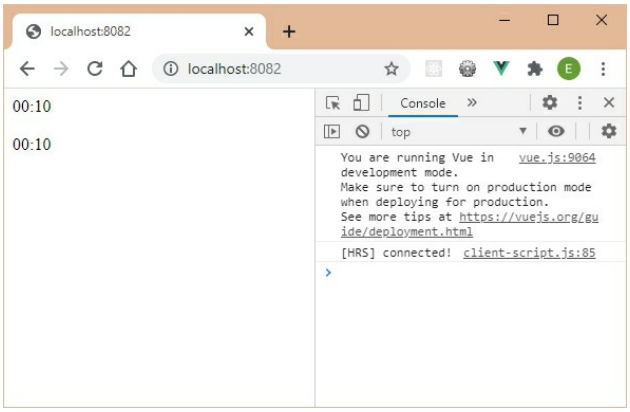
In fact it would have been better to write the final template in the following form, nesting the components:

Nesting of the Timer and StartStop components

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer>
        <StartStop />
      </Timer>
      <br>
      <Timer>
        <StartStop />
      </Timer>
    </div>
  `
});
```

Each Timer component includes a StartStop component in its body. So there is no longer any possible ambiguity. Each button manages the timer in which it is registered.

The page display is then as follows:



The Timer components are displayed, but not the buttons associated with the StartStop ...

In fact this is normal, because the Timer component is not planned (for the moment) to integrate a body inside it.

Vue proposes in this case to use an HTML element named <slot>, which we study in detail in the following section.

Use slots

A slot will allow you to create a body in a Vue component. A slot can be used in different ways to adapt to all situations:

- Using the slot minimally,
- By indicating a default content (if the component that uses it has not mentioned a body).
- You can also pass parameters to the slot.

Let us see below these different forms of use of the slot.

Minimal use of a slot

A slot used in its minimal form is used in the form <slot />, which can also be written in the form <slot> </slot>.

This HTML element (<slot>) is used in the component which is supposed to receive content, here the Timer component which will effectively contain the

<StartStop> content.

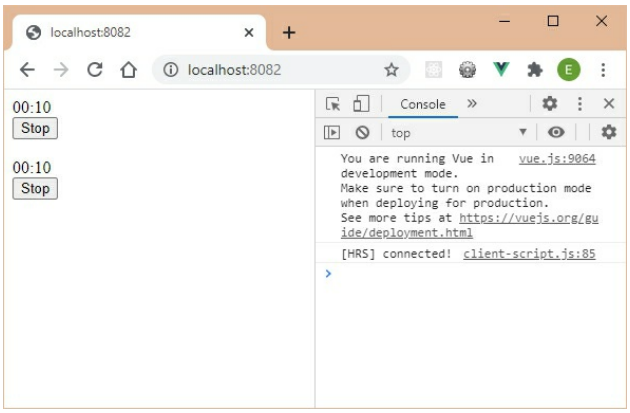
We could have done the opposite, namely that it is the StartStop component which contains the Timer component. In this case, the slot should have been nested in the StartStop component.

The Timer component is therefore modified to indicate to it that it can receive any content by means of the slot.

Indicate a slot in the Timer component

```
Vue.component("Timer", {
 template : `
    <div>
     00:10
     <slot/>
   </div>
 `
});
```

Indeed, when we display the HTML page using the same code for the Vue object template as before (two Timer components each including a StartStop component), we obtain a display that now meets our expectations:



The slot is well taken into account. Note that the function of the slot is just being replaced by the body specified in the component that uses it. We could have several different bodies for the use of the Timer component. For example :

Use different bodies for the Timer component

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer>
        Timer component using a StartStop component
        <StartStop />
      </Timer>
      <br>
      <Timer>
      </Timer>
    </div>
  `
});
```

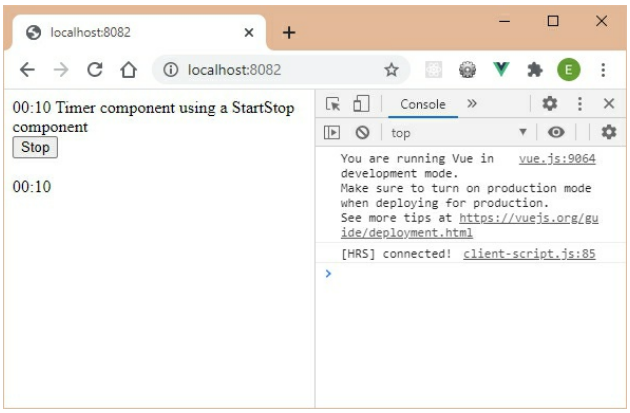
The preceding code shows that:

- The first Timer component includes a text and a

StartStop component: the text and the StartStop component will therefore be displayed after the Timer component.

- While the second Timer component does not integrate anything: only the Timer component will therefore be displayed.

The display therefore becomes:



> It is therefore clear that the principle of the slot is to integrate any content into the body of the component that uses it. This content can be empty, or different depending on the uses you want to make of the component.

Indicate default content in a slot

Since a component does not necessarily have content when it is used (for example the second use of the previous Timer component), it may be interesting to indicate one by default.

Suppose the rule here is to integrate a StartStop component into the Timer component if the content of the Timer component is empty during its use. The StartStop component therefore becomes a default content of the Timer component when the latter is used in the form <Timer /> (or <Timer></Timer>).

The code for the Timer and StartStop components, as well as their use, becomes:

Use default content in the body of the Timer component

```
Vue.component("Timer", {
 template : `
    <div>
    00:10
    <slot><StartStop/></slot>
   </div>
 `
});
Vue.component("StartStop", {
 data() {
  return {
    stopped : false
  }
 },
 template : `
    <div>
    <button v-if="stopped" v-on:click="stopped=!stopped">Start</button>
    <button v-else v-on:click="stopped=!stopped">Stop</button>
   </div>
 `
});
var vm = new Vue({
```

```
  el : "#root",
  template : `
    <div>
      <Timer>
        <StartStop />
      </Timer>
      <br>
      <Timer>
      </Timer>
    </div>
  `
});
```

The slot indicated in the Timer component is now written with content that will be displayed if the Timer component is used without a body (as is the case when it is used the second time in the example above).

The general form for writing default content in a slot is as follows:

Indicate default content in a slot

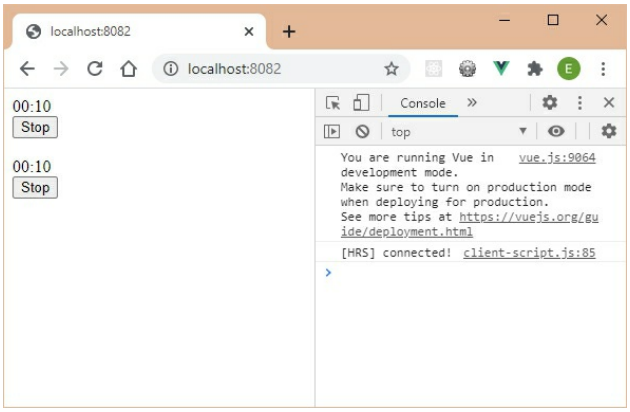
```
<slot>
  Default content that will be used if the component is used without a body
  This can be text, HTML, or even Vue components.
</slot>
```

> The default content of the slot is therefore written in the body of the <slot> tag itself.

The default content of the slot will be used if the component that integrates the slot is used without a body. If no default content was specified, the component body would be left empty if no body was

specified during use.

We now get two Timer components with associated StartStop components even if the Timer component is empty:



Transmission of information to a slot

We want the Timer component to transmit information to the StartStop component using the props (attributes) that can be specified in the slots.

For example, each Timer component has a name, and that name can then be used in the StartStop component. This would allow for example:

- In the first Timer component, we indicate the props name equal to "Timer 1". This timer is therefore called "Timer 1".
- This props is then used in the associated StartStop component by indicating on the corresponding button "Start Timer 1" or "Stop Timer 1" (instead of "Start" / "Stop" as

previously).

- Of course, for the second timer we would have the props name which would be equal to "Timer 2", with the corresponding display on the StartStop button ("Start Timer 2" or "Stop Timer 2").

This therefore makes it possible to show how to exchange information between a component and the slot included in it.

To achieve this, we will modify:

- The Timer component, so that it now includes the props name which must be used in the slot in order to be used in the StartStop component.
- The StartStop component, which now receives a new props associated with the name property passed from the slot.
- Finally the template for using the Timer and StartStop components in the Vue object, in order to indicate the values of the name property for each of the timers. We will see that the way of using the components is a little different from that used previously.

Here are the changes made in each section. Let's start with the Timer component.

Edit the Timer component

```
Vue.component("Timer", {
 props : [
   "name"
 ],
 template : `
    <div>
     <b>{{name}}</b> : 00:10
     <slot v-bind:timerSlot="name"></slot>
   </div>
 `
});
```

We have seen that the Timer component will be used in the form <Timer name="Timer 1">. This means that we pass it in the props the name property, which can be used in the template in the form {{name}}, or indicated as a parameter in the slot in the form <slot v-bind:timerSlot="name "></slot>. This form of slot writing means that we create a timerSlot props which will have the value of the name props transmitted to the Timer component.

The name timerSlot is used here so that we understand where the exchanged parameters go. But it would have been possible to call this new props name instead of timerSlot.

The StartStop component is also modified, in order to integrate the new timerSlot props received from the slot.

Modify the StartStop component

```
Vue.component("StartStop", {
 data() {
  return {
   stopped : true
  }
 },
 props : [
  "timerStartStop"
 ],
 methods : {
  start_stop() {
   this.stopped = !this.stopped;
  }
 },
 template : `
   <div>
    <button v-if="stopped" v-on:click="start_stop">
     Start {{timerStartStop}}
    </button>
    <button v-else v-on:click="start_stop">
     Stop {{timerStartStop}}
    </button>
  </div>
 `
});
```

The StartStop component receives the timerStartStop props which is then used in the component's template. This prop corresponds to the name of the timer that is indicated in the slot (the timerSlot prop).

We therefore see that we will have to make the correspondence between the name of the props in the

slot (which is timerSlot) and the name used in the StartStop component (which is timerStartStop). This correspondence occurs during the final use of the components in the Vue object.

Use components in the Vue object

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer name="Timer 1">
        <template slot-scope="{timerSlot}">
          <StartStop v-bind:timerStartStop="timerSlot"/>
        </template>
      </Timer>
      <br>
      <Timer name="Timer 2">
        <template slot-scope="{timerSlot}">
          <StartStop v-bind:timerStartStop="timerSlot"/>
        </template>
      </Timer>
    </div>
  `
});
```

Let's analyze the elements of the previous template:

- The Timer component is used in the form <Timer name="Timer 1">, which is what we wanted.
- The StartStop component is used in the form <StartStop v-bind:timerStartStop="timerSlot" />. , which means passing a prop named

timerStartStop which will have the value of the timerSlot props. The timerStartStop prop is indeed the one used in the StartStop component, which we wrote above.

On the other hand, the timerSlot property seems to come out of nowhere! So that Vue knows how to link these two properties, it uses the <template> tag with the slot-scope attribute of value "{timerSlot}":

- The <template> element is an HTML element that allows it not to be integrated into the DOM tree, but still allows to encapsulate other HTML elements. This allows you to specify the slot-scope attribute here. The <template> element could here be replaced by a element for example, but in this case this element would be integrated in the DOM, unlike the <template> element which is not. .
- The slot-scope attribute indicated in the form "{timerSlot}" makes it possible here to retrieve the value of the timerSlot property (hence the braces), which is the one indicated in the <slot>, namely the value of the props name (so the strings "Timer 1" or "Timer 2" depending on the Timer component used). If we had simply indicated "timerSlot" (without the braces), we would have

obtained the string "{timerSlot:'Timer 1'}" for the first timer, hence the use of braces to access the value of the timerSlot property.

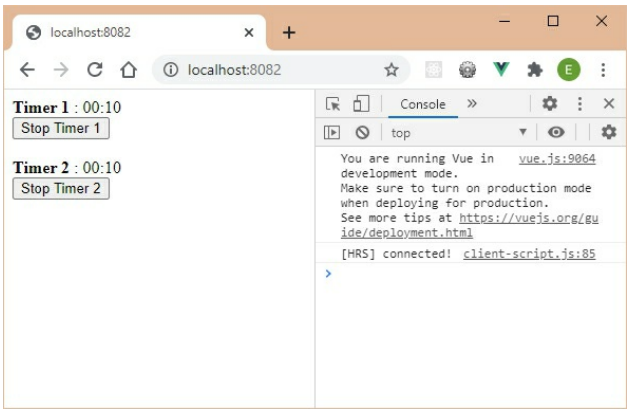
The general program that encompasses these components and the Vue object is therefore the following (it is the simple assembly of the three pieces of previous programs):

Timer and StartStop components and use in the Vue object

```
Vue.component("Timer", {
 props : [
  "name"
 ],
 template : `
    <div>
     <b>{{name}}</b> : 00:10
     <slot v-bind:timerSlot="name"></slot>
   </div>
 `
});
Vue.component("StartStop", {
 data() {
  return {
    stopped : true
  }
 },
 props : [
  "timerStartStop"
 ],
 methods : {
  start_stop() {
    this.stopped = !this.stopped;
```

```
    }
  },
  template : `
     <div>
      <button v-if="stopped" v-on:click="start_stop">
          Start {{timerStartStop}}
      </button>
      <button v-else v-on:click="start_stop">
          Stop {{timerStartStop}}
      </button>
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer name="Timer 1">
        <template slot-scope="{timerSlot}">
          <StartStop v-bind:timerStartStop="timerSlot"/>
        </template>
      </Timer>
      <br>
      <Timer name="Timer 2">
        <template slot-scope="{timerSlot}">
          <StartStop v-bind:timerStartStop="timerSlot"/>
        </template>
      </Timer>
    </div>
  `
});
```

Let's see if we get the expected result in the browser:



The name of the timer is now integrated into the label of the button (which is itself integrated into the StartStop component).

Writing simpler for the previous program

When writing the previous program, we wanted to show the use of each of the variables used (for this reason, none of the variables or props has a name that we could find elsewhere):

- The name props is used to retrieve the name of the timer in the Timer component,
- The timerSlot props is used to transfer the name of the timer to a child component, and thus to make the link between parent and child,
- The timerStartStop props are used to retrieve the name of the timer in the StartStop component.

That's a lot of different property names that ultimately

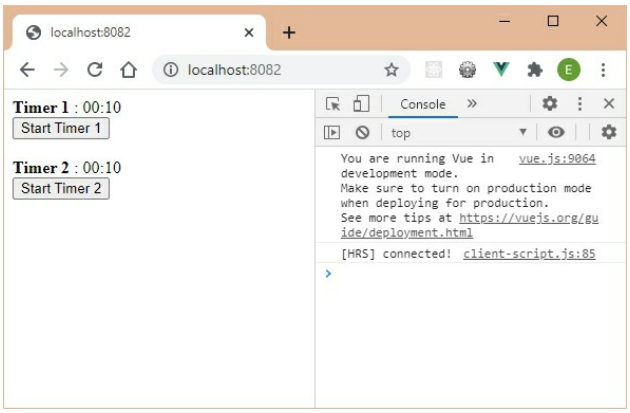
serve to convey the name of the timer. Would it be possible to simplify all this by keeping, for example, the name timername wherever possible?

Replace the name, timerSlot and timerStartStop props with timername

```
Vue.component("Timer", {
 props : [
   "timername"
 ],
 template : `
    <div>
    <b>{{timername}}</b> : 00:10
    <slot v-bind:timername="timername"></slot>
  </div>
 `
});
Vue.component("StartStop", {
 data() {
   return {
     stopped : true
   }
 },
 props : [
   "timername"
 ],
 methods : {
   start_stop() {
     this.stopped = !this.stopped;
   }
 },
 template : `
   <div>
    <button v-if="stopped" v-on:click="start_stop">
```

```
        Start {{timername}}
      </button>
      <button v-else v-on:click="start_stop">
        Stop {{timername}}
      </button>
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer timername="Timer 1">
        <template slot-scope="{timername}">
          <StartStop v-bind:timername="timername"/>
        </template>
      </Timer>
      <br>
      <Timer timername="Timer 2">
        <template slot-scope="{timername}">
          <StartStop v-bind:timername="timername"/>
        </template>
      </Timer>
    </div>
  `
});
```

Let's check that the operation is identical:



Communication from the child component to its parent

We saw previously that the communication from the parent component (here Timer) to the child component (here StartStop) was carried out via the props. We now want to show how a child component can communicate with its parent.

Let's improve our program so that the click on the Start / Stop button is taken into account. For the moment, only the text on the button is modified when the button is clicked (thanks to the reactive stopped variable updated in the StartStop component).

We therefore want the Timer component to react when clicking on the button, by displaying (for the moment) in the console the name of the timer clicked and its state (stopped or in progress).

The interest here will therefore be to show how an

internal component (here the StartStop component) can communicate with a parent component (here the Timer component). Communication from the child to the parent will take place through events.

Two steps are necessary to perform this treatment:

- We must first modify the StartStop component (the child, which manages the click on the button) by sending an event (here called startstop) to the Timer component (the parent).
- We must then retrieve in the Timer component (the parent) the event transmitted by the StartStop component (the child) in order to process it (by displaying the button state here: stopped or in progress).

Now let's see these two steps.

StartStop child component that sends a startstop message to the Timer parent component

```
Vue.component("StartStop", {
  data() {
    return {
      stopped : true
    }
  },
  props : [
    "timername"
  ],
  methods : {
```

```
    start_stop() {
      this.stopped = !this.stopped;
      this.$parent.$emit("startstop", this.stopped);
    }
  },
  template : `
     <div>
      <button v-if="stopped" v-on:click="start_stop">
          Start {{timername}}
      </button>
      <button v-else v-on:click="start_stop">
          Stop {{timername}}
      </button>
    </div>
  `
});
```

Only the start_stop() method is affected here. The this object here symbolizes the StartStop component in which the methods are written. Vue allows you to access the parent component using this.$parent (so here the Timer component). Once you have access to the parent component, all you have to do is send it an event using the $emit() method proposed by Vue on each component.

The $emit() method is used with at least one parameter which is the name of the event transmitted, and any parameters which correspond to the arguments that we wish to transmit. Here we only transmit the state of the button which is stored in the reactive variable stopped.

For the moment, the transmission of the event is of

little use if this event is not listened to by the receiver, here the Timer component. It is therefore necessary to modify the Timer component to make it process the reception of this event. For the moment, the state of the button will be displayed ("stopped" or "in progress").

Parent component Timer that receives the startstop event from the child StartStop component

```
Vue.component("Timer", {
 props : [
  "timername"
 ],
 created() {
  this.$on("startstop", function(stop) {
   console.log(this.timername + " " + (stop ? "stopped" : "in
progress"));
  });
 },
 template : `
   <div>
    <b>{{timername}}</b> : 00:10
    <slot v-bind:timername="timername"></slot>
  </div>
 `
});
```

The Timer component must listen to the startstop event as soon as it is created. For this, we use the created() method called when creating the component. The created() method here calls the $on() method:

- The $on() method defined by Vue on a component is used to process the received event,

indicated as the first parameter of the method (here "startstop").

- The second parameter of the $on() method is a callback function called to process the receipt of the indicated event. This callback function takes as parameters the arguments indicated during the emission of the event by $emit() (in the StartStop component, see above). Here it will be the stop argument (or any other name) which represents the state of the button (if true: the timer is stopped, if false: the timer is in progress). If no argument passed is used, it suffices not to specify any parameters in the callback function.

The complete program including all modified components is as follows:

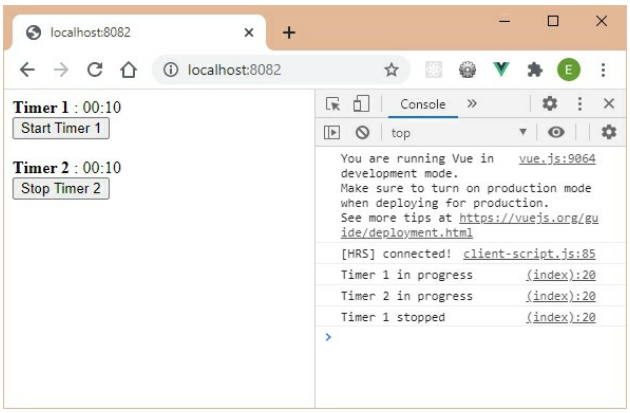
Full program

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="root"></div>
</body>
<script>
Vue.component("Timer", {
  props : [
    "timername"
  ],
```

```
  created() {
    this.$on("startstop", function(stop) {
      console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
    });
  },
  template : `
     <div>
      <b>{{timername}}</b> : 00:10
      <slot v-bind:timername="timername"></slot>
    </div>
  `
});
Vue.component("StartStop", {
  data() {
    return {
      stopped : true
    }
  },
  props : [
    "timername"
  ],
  methods : {
    start_stop() {
      this.stopped = !this.stopped;
      this.$parent.$emit("startstop", this.stopped);
    }
  },
  template : `
     <div>
      <button v-if="stopped" v-on:click="start_stop">
          Start {{timername}}
      </button>
      <button v-else v-on:click="start_stop">
          Stop {{timername}}
      </button>
    </div>
```

```
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer timername="Timer 1">
        <template slot-scope="{timername}">
          <StartStop v-bind:timername="timername"/>
        </template>
      </Timer>
      <br>
      <Timer timername="Timer 2">
        <template slot-scope="{timername}">
          <StartStop v-bind:timername="timername"/>
        </template>
      </Timer>
    </div>
  `
});
</script>
</html>
```

Let's display the corresponding page and click on the two buttons to check the texts displayed in the console.



We thus verify that a child component can

communicate with its parent by using the $emit() and $on() methods provided by Vue in the components.

Integrate the Timer component and the StartStop component

Now that we have seen the use of slots and their usefulness in our case, let's continue the integration of the two components Timer and StartStop written previously. In fact, for simplicity, we have removed the time reduction aspect by leaving the timer immobilized on a fixed time which is 00:10.

This is the moment when we can finally integrate the variation of the time, as well as the click on the Start / Stop buttons to start or stop the countdown. We take part of the code that we had voluntarily deleted to focus only on the new concepts studied.

Timer managed by the Start / Stop button

```
Vue.component("Timer", {
 data() {
  return {
      min : this.minutes,
    sec : this.seconds
  }
 },
 props : [
   "timername",
    "minutes",
```

```
    "seconds"
  ],
  computed : {
    time() {
      var min = this.min;
      var sec = this.sec;
      if (min < 10) min = "0" + min;
      if (sec < 10) sec = "0" + sec;
      return min + ":" + sec;
    }
  },
  created() {
    var timer;
    this.$on("startstop", function(stop) {
      if (stop) {
        if (timer) clearInterval(timer);
      }
      else {
        timer = setInterval(() => {
          this.sec -= 1;
          if (this.sec < 0) {
            this.min -= 1;
            if (this.min < 0) {
              this.min = 0;
              this.sec = 0;
              clearInterval(timer);
            }
            else this.sec = 59;
          }
        }, 1000);
      }
      console.log(this.timername + " " + (stop ? "stopped" : "in
progress"));
    });
  },
  template : `
```

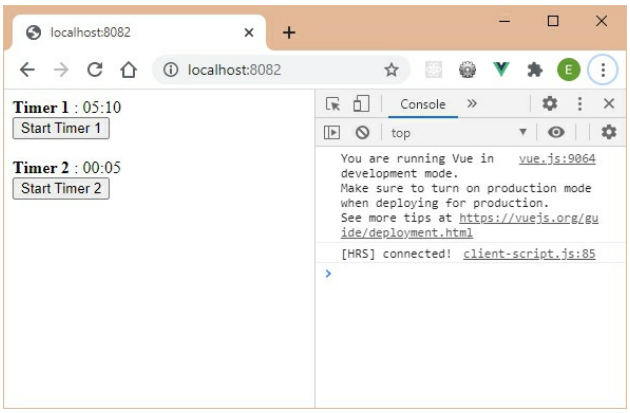
```
    <div>
      <b>{{timername}}</b> :
      <span v-if="sec||min"> {{time}} </span>
      <span v-else>End of timer</span>
      <slot v-bind:timername="timername"></slot>
    </div>
  `
});
Vue.component("StartStop", {
  data() {
    return {
      stopped : true
    }
  },
  props : [
    "timername"
  ],
  methods : {
    start_stop() {
      this.stopped = !this.stopped;
      this.$parent.$emit("startstop", this.stopped);
    }
  },
  template : `
     <div>
      <button v-if="stopped" v-on:click="start_stop">
          Start {{timername}}
      </button>
      <button v-else v-on:click="start_stop">
          Stop {{timername}}
      </button>
    </div>
  `
});
var vm = new Vue({
  el : "#root",
```

```
  template : `
    <div>
      <Timer timername="Timer 1" minutes="5" seconds="10">
        <template slot-scope="{timername}">
          <StartStop v-bind:timername="timername"/>
        </template>
      </Timer>
      <br>
      <Timer timername="Timer 2" minutes="0" seconds="5">
        <template slot-scope="{timername}">
          <StartStop v-bind:timername="timername"/>
        </template>
      </Timer>
    </div>
  `
});
```

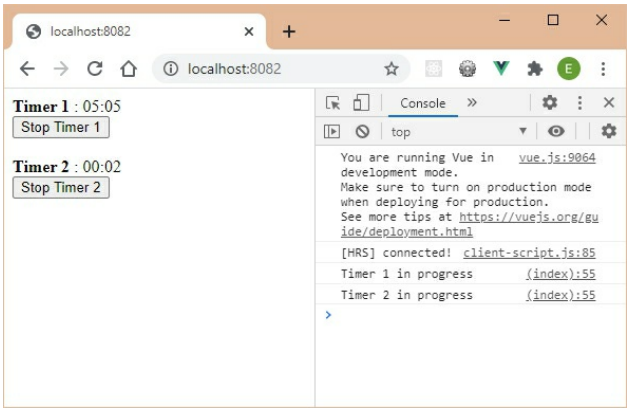
The code is similar to what we have already written:

- We simply added the minutes and seconds props in the Timer component, as well as the created() method which allows to start or stop the timer.
- The reactive min and sec variables of the Timer component are retrieved from the minutes and seconds props transmitted in the Timer component.

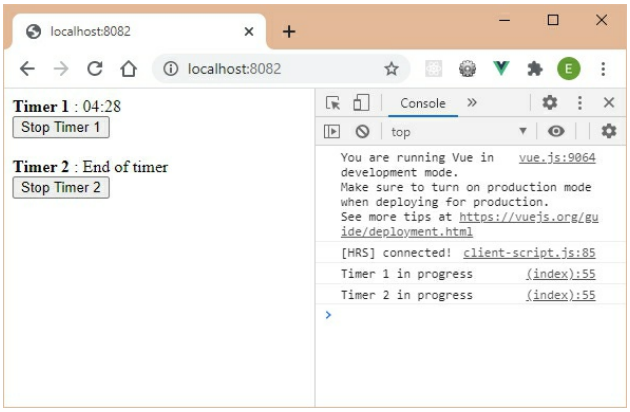
Let's check that everything is working correctly: when the program is launched, none of the timers are active:



Let's click on the two buttons successively: the timers start:



When timer 2 expires, the message End of timer is displayed:



We can see some possible areas for improvement:

- Be able to start a timer (as soon as it is displayed) if the associated button is set to Stop when it is initialized (so the timer can be stopped by clicking on the button),
- When a timer is at 00:00, you must be able to restart it and change its label from Stop to Start.
- If a Timer component does not have StartStop content, start the timer automatically as soon as the timer is created.

These improvements will be those made in the following sections.

Start a timer if the associated button is set to Stop when it is initialized

The Start / Stop button of each timer is set to Start because the reactive variable stopped of the StartStop component is set to true. Setting this variable to false in the data section displays the Stop button but the timer does not start. This shows that improvements are desirable ...

Here we want to be able to manage the state of each button independently, and not by directly initializing the reactive variable stopped to true or false (because

the configuration chosen would in this case be the same for all the timers).

For this we introduce a new props in the StartStop component, namely the stop props allowing to indicate if the timer is stopped (stop="1") or started (stop="0").

The StartStop component would then be used as follows:

Start the timer as soon as it is created

```
<StartStop v-bind:timername="timername" stop="0" />
```

Stop the timer as soon as it is created

```
<StartStop v-bind:timername="timername" stop="1" />
```

> Positioning the props stop to 0 or 1 therefore allows the StartStop component to know whether the timer should start or not.

The StartStop component must be modified to handle this new props. When creating the component, we see if the timer is on (stop="0"), and if so we must start the timer.

StartStop component including the stop props

```
Vue.component("StartStop", {
 data() {
  return {
   stopped : parseInt(this.stop)
  }
 },
 props : [
```

```
    "timername",
   "stop"
 ],
 created() {
  if (!this.stopped) {
    this.stopped = !this.stopped;
    this.start_stop();
  }
 },
 methods : {
  start_stop() {
    this.stopped = !this.stopped;
    this.$parent.$emit("startstop", this.stopped);
  }
 },
 template : `
    <div>
    <button v-if="stopped" v-on:click="start_stop">
        Start {{timername}}
    </button>
    <button v-else v-on:click="start_stop">
        Stop {{timername}}
    </button>
  </div>
 `
});
```

The reactive variable stopped is now initialized from the stop props. The parseInt() instruction used allows you to retrieve the integer value of the props, because it is indicated in the form of a character string when it is used.

The stop props is also added in the list of component

props (in addition to the timername props), while the created() method allows processing (starting or not the timer by simulating a click on the button).

The following program takes all the code by setting the props stop to "1" for the first timer and to "0" for the second.

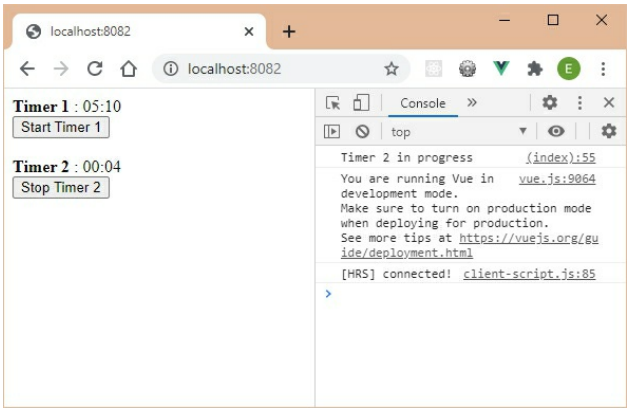
Use the stop props to start or not the timer

```
Vue.component("Timer", {
  data() {
    return {
        min : this.minutes,
      sec : this.seconds
    }
  },
  props : [
    "timername",
     "minutes",
    "seconds"
  ],
  computed : {
    time() {
      var min = this.min;
      var sec = this.sec;
      if (min < 10) min = "0" + min;
      if (sec < 10) sec = "0" + sec;
      return min + ":" + sec;
    }
  },
  created() {
    var timer;
    this.$on("startstop", function(stop) {
      if (stop) {
```

```
      if (timer) clearInterval(timer);
    }
    else {
      timer = setInterval(() => {
        this.sec -= 1;
        if (this.sec < 0) {
          this.min -= 1;
          if (this.min < 0) {
            this.min = 0;
            this.sec = 0;
            clearInterval(timer);
          }
          else this.sec = 59;
        }
      }, 1000);
    }
    console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
  });
 },
 template : `
    <div>
    <b>{{timername}}</b> :
    <span v-if="sec||min"> {{time}} </span>
    <span v-else>End of timer</span>
    <slot v-bind:timername="timername"></slot>
  </div>
 `
});
Vue.component("StartStop", {
 data() {
  return {
    stopped : parseInt(this.stop)
  }
 },
 props : [
   "timername",
```

```
    "stop"
  ],
  created() {
    if (!this.stopped) {
      this.stopped = !this.stopped;
      this.start_stop();
    }
  },
  methods : {
    start_stop() {
      this.stopped = !this.stopped;
      this.$parent.$emit("startstop", this.stopped);
    }
  },
  template : `
     <div>
      <button v-if="stopped" v-on:click="start_stop">
          Start {{timername}}
      </button>
      <button v-else v-on:click="start_stop">
          Stop {{timername}}
      </button>
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer timername="Timer 1" minutes="5" seconds="10">
        <template slot-scope="{timername}">
          <StartStop v-bind:timername="timername" stop="1" />
        </template>
      </Timer>
      <br>
      <Timer timername="Timer 2" minutes="0" seconds="5">
```

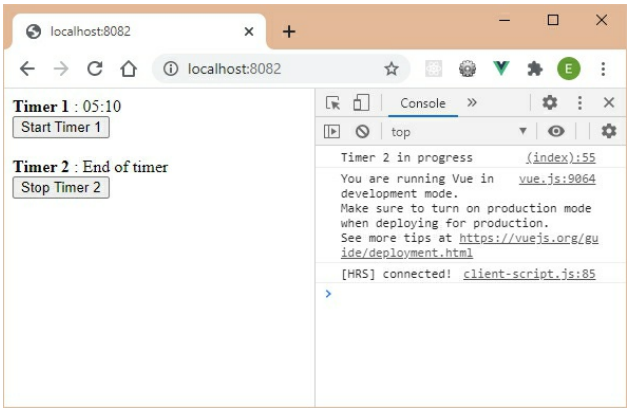
```
      <template slot-scope="{timername}">
        <StartStop v-bind:timername="timername" stop="0" />
      </template>
    </Timer>
  </div>
  `
});
```



The second timer is started directly, unlike the first for which you must click on the button.

Restart a timer that has expired

When timer 2 has expired, the button text is not refreshed:



Timer 2 has expired and yet the associated button text is still Stop. It should be set to Start and also reset the initial values of the timer.

To achieve this:

- The Timer component, which knows that the timer has expired, should be able to access the start_stop() method of the StartStop child component. Calling this method will refresh the button text automatically.
- Updating the displayed minutes and seconds can be done directly in the Timer component by resetting them with the values transmitted in the props (this.minutes and this.seconds).

Use the created() method or the mounted() method?

A parent component (here the Timer component) can have access to its child components (here the StartStop component) by means of the this.$children instruction which is an array accessible in each component. This $children array is however accessible only when the component is in the mounted state, and not just created: this means that you have to wait until the component is registered in the DOM to be able to have access to its children.

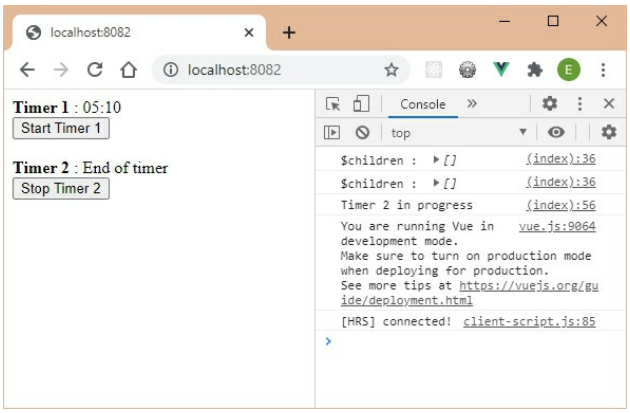
You can easily check this by displaying in the console the content of the this.$children object in the created() method of the Timer component:

Display this.$children in the created method of the Timer component

```
Vue.component("Timer", {
 data() {
  return {
      min : this.minutes,
    sec : this.seconds
  }
 },
 props : [
  "timername",
   "minutes",
  "seconds"
 ],
 computed : {
  time() {
    var min = this.min;
    var sec = this.sec;
    if (min < 10) min = "0" + min;
    if (sec < 10) sec = "0" + sec;
    return min + ":" + sec;
  }
 },
 created() {
  console.log("$children : ", this.$children);
  var timer;
  this.$on("startstop", function(stop) {
    if (stop) {
      if (timer) clearInterval(timer);
    }
```

```
    else {
      timer = setInterval(() => {
        this.sec -= 1;
        if (this.sec < 0) {
          this.min -= 1;
          if (this.min < 0) {
            this.min = 0;
            this.sec = 0;
            clearInterval(timer);
          }
          else this.sec = 59;
        }
      }, 1000);
    }
    console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
  });
 },
 template : `
   <div>
    <b>{{timername}}</b> :
    <span v-if="sec||min"> {{time}} </span>
    <span v-else>End of timer</span>
    <slot v-bind:timername="timername"></slot>
  </div>
 `
});
```

Launching the program displays the contents of the $children array of each Timer component in the console. We can see that this content is an empty array [] for each of the timers. The reason is that we are accessing the $children array while the component is only in the created state, and not yet mounted.



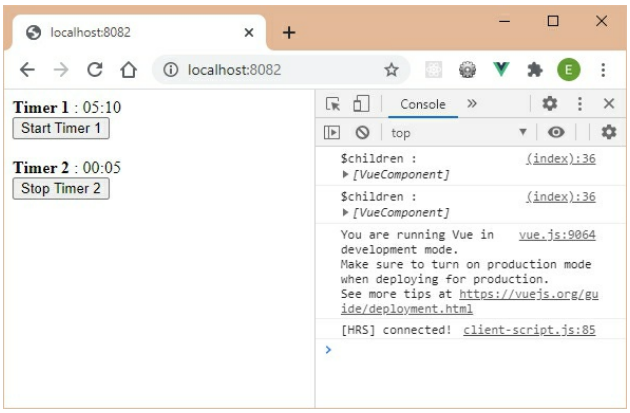
All you have to do is change the name of the method in the Timer component: created() becomes mounted().

Replace created() with mounted() in the Timer component

```
Vue.component("Timer", {
  data() {
    return {
        min : this.minutes,
      sec : this.seconds
    }
  },
  props : [
    "timername",
     "minutes",
    "seconds"
  ],
  computed : {
    time() {
      var min = this.min;
      var sec = this.sec;
      if (min < 10) min = "0" + min;
      if (sec < 10) sec = "0" + sec;
      return min + ":" + sec;
    }
  },
  mounted() {
```

```
  console.log("$children : ", this.$children);
  var timer;
  this.$on("startstop", function(stop) {
    if (stop) {
      if (timer) clearInterval(timer);
    }
    else {
      timer = setInterval(() => {
        this.sec -= 1;
        if (this.sec < 0) {
          this.min -= 1;
          if (this.min < 0) {
            this.min = 0;
            this.sec = 0;
            clearInterval(timer);
          }
          else this.sec = 59;
        }
      }, 1000);
    }
    console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
  });
 },
 template : `
    <div>
    <b>{{timername}}</b> :
    <span v-if="sec||min"> {{time}} </span>
    <span v-else>End of timer</span>
    <slot v-bind:timername="timername"></slot>
  </div>
 `
});
```

Once done, we get:



The components included in the $children array are now accessible.

However, if you look closely you will see that the second timer did not start automatically as it was before ... Starting the timer in the mounted() method instead of created() messed up the operation ... Here is the reason:

The created() method of the StartStop component executes before the mounted() method of the Timer component is executed, so executing the start_stop() method in the StartStop component does indeed send a startstop message to the Timer component , but this one is not yet ready to process it. The StartStop component should wait until its parent component is ready before sending it the startstop message. For that, we will use the $nextTick() method.

Use the $nextTick() method

Vue has foreseen these extreme cases. If you read the

Vue API https://vuejs.org/v2/api/#mounted you see that Vue says to wait for the "next tick" so that the parent component and the child are both in the mounted state. For this, we use, as indicated, the method this.$nextTick(callback) in which the callback function is executed when both components are in the mounted state.

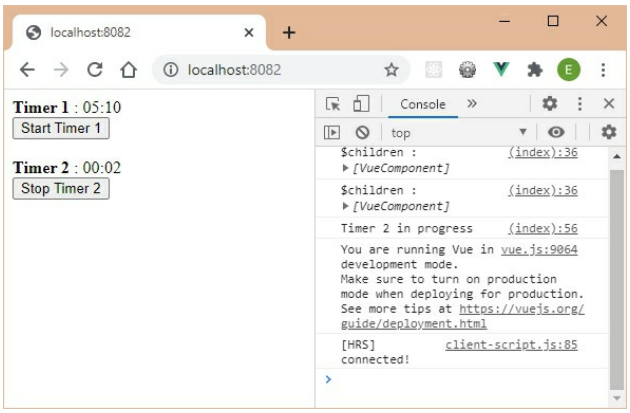
Let's use the $nextTick() method in the start_stop() method of the StartStop component. The startstop message will only be issued when both components are in the mounted state.

Use the $nextTick() method in the StartStop component

```
Vue.component("StartStop", {
 data() {
  return {
   stopped : parseInt(this.stop)
  }
 },
 props : [
   "timername",
  "stop"
 ],
 created() {
  if (!this.stopped) {
   this.stopped = !this.stopped;
   this.start_stop();
  }
 },
 methods : {
  start_stop() {
```

```
    this.$nextTick(function() {
      this.stopped = !this.stopped;
      this.$parent.$emit("startstop", this.stopped);
    });
   }
 },
 template : `
    <div>
     <button v-if="stopped" v-on:click="start_stop">
         Start {{timername}}
     </button>
     <button v-else v-on:click="start_stop">
         Stop {{timername}}
     </button>
   </div>
 `
});
```

Now let's check that the second timer starts correctly when the page loads.



Access the $children array

Each $children array contains a VueComponent class component that actually corresponds to the StartStop component:

- Access to this component is via this.$children[0],
- Then we access the start_stop() method of the component by means of the this.$children[0].start_stop() instruction.

Let's modify the program to incorporate these changes. The Timer and StartStop components are impacted.

Modification of components

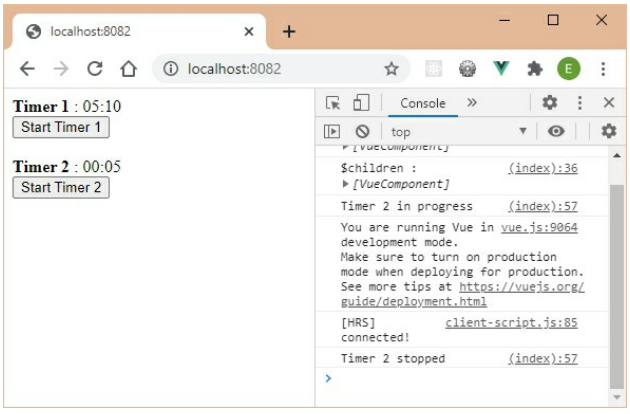
```
Vue.component("Timer", {
 data() {
  return {
      min : this.minutes,
    sec : this.seconds
  }
 },
 props : [
  "timername",
   "minutes",
  "seconds"
 ],
 computed : {
  time() {
    var min = this.min;
    var sec = this.sec;
    if (min < 10) min = "0" + min;
    if (sec < 10) sec = "0" + sec;
    return min + ":" + sec;
  }
 },
 mounted() {
  console.log("$children : ", this.$children);
```

```
  var timer;
  this.$on("startstop", function(stop) {
    if (stop) {
      if (timer) clearInterval(timer);
    }
    else {
      timer = setInterval(() => {
        this.sec -= 1;
        if (this.sec < 0) {
          this.min -= 1;
          if (this.min < 0) {
            this.min = this.minutes;
            this.sec = this.seconds;
            this.$children[0].start_stop();
            clearInterval(timer);
          }
          else this.sec = 59;
        }
      }, 1000);
    }
    console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
  });
 },
 template : `
    <div>
     <b>{{timername}}</b> :
     <span v-if="sec||min"> {{time}} </span>
     <span v-else>End of timer</span>
     <slot v-bind:timername="timername"></slot>
   </div>
 `
});
Vue.component("StartStop", {
 data() {
   return {
     stopped : parseInt(this.stop)
```

```
    }
  },
  props : [
     "timername",
    "stop"
  ],
  created() {
    if (!this.stopped) {
      this.stopped = !this.stopped;
      this.start_stop();
    }
  },
  methods : {
    start_stop() {
      this.$nextTick(function() {
        this.stopped = !this.stopped;
        this.$parent.$emit("startstop", this.stopped);
      });
    }
  },
  template : `
     <div>
      <button v-if="stopped" v-on:click="start_stop">
          Start {{timername}}
      </button>
      <button v-else v-on:click="start_stop">
          Stop {{timername}}
      </button>
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer timername="Timer 1" minutes="5" seconds="10">
```

```
      <template slot-scope="{timername}">
        <StartStop v-bind:timername="timername" stop="1" />
      </template>
    </Timer>
    <br>
    <Timer timername="Timer 2" minutes="0" seconds="5">
      <template slot-scope="{timername}">
        <StartStop v-bind:timername="timername" stop="0" />
      </template>
    </Timer>
  </div>
 `
});
```

Let's start the program and wait for the end of the second timer: once the end of it, the second timer is repositioned on Start and the remaining time is displayed again.



Start the timer automatically if the StartStop component is missing

We want to be able to use the Timer component without necessarily inserting a StartStop component in

its body. In this case, the timer must start automatically and restart at each end of the expiry date (perpetual timer).

The Timer component must test for the presence of a child in its $children array:

- If a child is present, we assume that it is the StartStop component, and in this case we use the start_stop() method on this child (as before).
- Otherwise, the StartStop component is absent, and in this case, the startstop message must be sent in the Timer component to start it directly (you cannot use the start_stop() method because the StartStop component is absent).

Let's write the corresponding program.

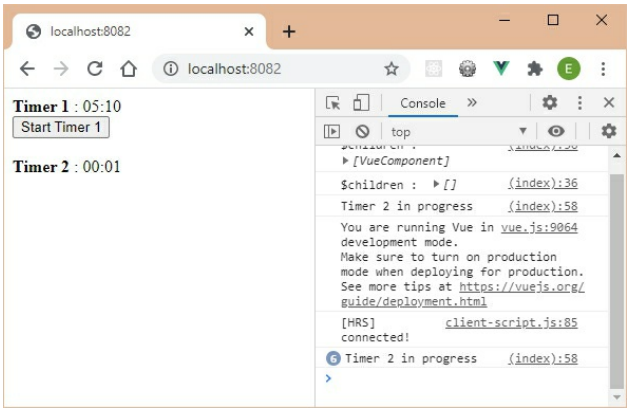
Start the timer if the StartStop component is missing

```
Vue.component("Timer", {
 data() {
  return {
      min : this.minutes,
    sec : this.seconds
  }
 },
 props : [
  "timername",
   "minutes",
  "seconds"
 ],
 computed : {
```

```
  time() {
    var min = this.min;
    var sec = this.sec;
    if (min < 10) min = "0" + min;
    if (sec < 10) sec = "0" + sec;
    return min + ":" + sec;
  }
},
mounted() {
  console.log("$children : ", this.$children);
  var timer;
  this.$on("startstop", function(stop) {
    if (stop) {
      if (timer) clearInterval(timer);
    }
    else {
      timer = setInterval(() => {
        this.sec -= 1;
        if (this.sec < 0) {
          this.min -= 1;
          if (this.min < 0) {
            this.min = this.minutes;
            this.sec = this.seconds;
            if (this.$children.length) this.$children[0].start_stop();
            else this.$emit("startstop", 0);
            clearInterval(timer);
          }
          else this.sec = 59;
        }
      }, 1000);
    }
    console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
  });
  if (!this.$children.length) this.$emit("startstop", 0);
},
template : `
```

```
    <div>
     <b>{{timername}}</b> :
     <span v-if="sec||min"> {{time}} </span>
     <span v-else>End of timer</span>
     <slot v-bind:timername="timername"></slot>
    </div>
  `
});
Vue.component("StartStop", {
  data() {
    return {
      stopped : parseInt(this.stop)
    }
  },
  props : [
     "timername",
    "stop"
  ],
  created() {
    if (!this.stopped) {
      this.stopped = !this.stopped;
      this.start_stop();
    }
  },
  methods : {
    start_stop() {
      this.$nextTick(function() {
        this.stopped = !this.stopped;
        this.$parent.$emit("startstop", this.stopped);
      });
    }
  },
  template : `
     <div>
     <button v-if="stopped" v-on:click="start_stop">
          Start {{timername}}
```

```
    </button>
    <button v-else v-on:click="start_stop">
        Stop {{timername}}
    </button>
  </div>
 `
});
var vm = new Vue({
 el : "#root",
 template : `
  <div>
    <Timer timername="Timer 1" minutes="5" seconds="10">
      <template slot-scope="{timername}">
        <StartStop v-bind:timername="timername" stop="1" />
      </template>
    </Timer>
    <br>
    <Timer timername="Timer 2" minutes="0" seconds="5">
    </Timer>
  </div>
 `
});
```



The second timer is now started at each end of the cycle (visible here in the display of the console, here 6 times).

Create new components from existing ones

Vue allows you to create new components from those already created. For example :

- The TimerStartStop component would combine the Timer component and the StartStop component into a single component,
- The TimerStartCount component would do the same, but additionally displaying the number of times the timer has been started.

Let's take a look at each of these new components.

TimerStartStop component

Let's start by creating a TimerStartStop component that would combine the Timer and StartStop components into one.

Its use would also be simplified, because it would no longer be necessary to surround the StartStop component with the <template> indicating the slot-scope attribute. We would use it as follows:

Using the new TimerStartStop component

```
<TimerStartStop timername="Timer 3" minutes="1" seconds="5">
</TimerStartStop>
```

Or also, since the TimerStartStop component has no

body:

Using the new TimerStartStop component

```
<TimerStartStop timername="Timer 3" minutes="1" seconds="5" />
```

The props are the same as those used in the Timer component.

The TimerStartStop component is created as follows:

TimerStartStop component grouping the Timer and StartStop components

```
Vue.component("TimerStartStop", {
 props : [
   "timername",
    "minutes",
   "seconds"
 ],
 template : `
   <Timer v-bind:timername="timername"
            v-bind:minutes="minutes"
         v-bind:seconds="seconds">
     <StartStop v-bind:timername="timername" v-bind:stop="1" />
   </Timer>
 `
});
```

The timer is supposed to be stopped on launch, because the stop props is set to "1" in the StartStop component. If you set it to "0", the timer starts immediately.

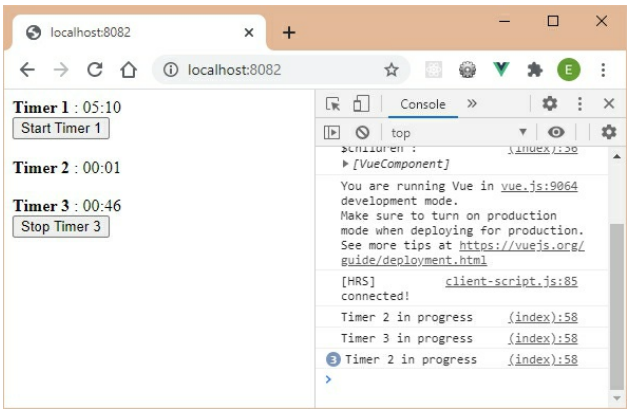
The code of the other components is not changed. The Vue object can use the different components created in the different forms.

Vue object using the created components

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <Timer timername="Timer 1" minutes="5" seconds="10">
        <template slot-scope="{timername}">
          <StartStop v-bind:timername="timername" stop="1" />
        </template>
      </Timer>
      <br>
      <Timer timername="Timer 2" minutes="0" seconds="5">
      </Timer>
      <br>
      <TimerStartStop timername="Timer 3" minutes="1" seconds="5">
      </TimerStartStop>
    </div>
  `
});
```

The Timer 3 timer is the one created from the new TimerStartStop component.

Let's check that these timers are working:



TimerStartCount component

This new component uses the same principle as the previous TimerStartStop. From the two components Timer and StartStop, we create this new component which also allows to display the number of times that we clicked on each button to start a timer.

You must therefore be warned that the timer has been started. The Timer component knows this because it receives the startstop event, with the stop argument as a parameter, which is 1 if the timer is stopped, 0 otherwise.

You just need to use the v-on:startstop directive on the Timer component to perform processing when this event is received.

TimerStartCount component that displays the number of timer starts

```
Vue.component("Timer", {
 data() {
  return {
      min : this.minutes,
    sec : this.seconds
  }
 },
 props : [
  "timername",
   "minutes",
  "seconds"
 ],
 computed : {
  time() {
```

```
    var min = this.min;
    var sec = this.sec;
    if (min < 10) min = "0" + min;
    if (sec < 10) sec = "0" + sec;
    return min + ":" + sec;
  }
},
mounted() {
  var timer;
  this.$on("startstop", function(stop) {
    if (stop) {
      if (timer) clearInterval(timer);
    }
    else {
      timer = setInterval(() => {
        this.sec -= 1;
        if (this.sec < 0) {
          this.min -= 1;
          if (this.min < 0) {
            this.min = this.minutes;
            this.sec = this.seconds;
            if (this.$children.length) this.$children[0].start_stop();
            else this.$emit("startstop", 0);
            clearInterval(timer);
          }
          else this.sec = 59;
        }
      }, 1000);
    }
    console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
  });
  if (!this.$children.length) this.$emit("startstop", 0);
},
template : `
   <div>
    <b>{{timername}}</b> :
```

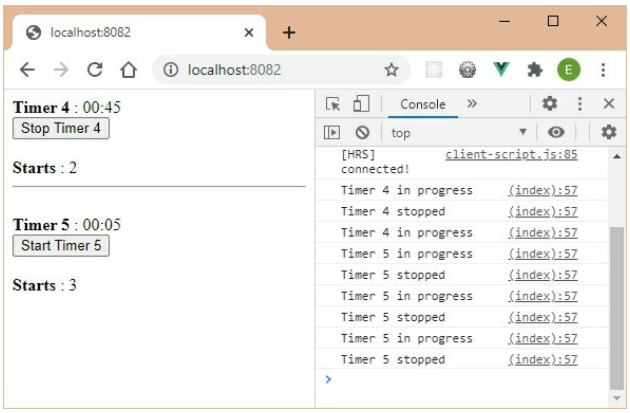
```
    <span v-if="sec||min"> {{time}} </span>
    <span v-else>End of timer</span>
    <slot v-bind:timername="timername"></slot>
  </div>
 `
});
Vue.component("StartStop", {
 data() {
  return {
   stopped : parseInt(this.stop)
  }
 },
 props : [
   "timername",
  "stop"
 ],
 created() {
  if (!this.stopped) {
   this.stopped = !this.stopped;
   this.start_stop();
  }
 },
 methods : {
  start_stop() {
   this.$nextTick(function() {
    this.stopped = !this.stopped;
    this.$parent.$emit("startstop", this.stopped);
   });
  }
 },
 template : `
   <div>
   <button v-if="stopped" v-on:click="start_stop">
       Start {{timername}}
   </button>
   <button v-else v-on:click="start_stop">
```

```
        Stop {{timername}}
     </button>
   </div>
 `
});
Vue.component("TimerStartStop", {
 props : [
   "timername",
    "minutes",
   "seconds"
 ],
 template : `
    <Timer v-bind:timername="timername"
           v-bind:minutes="minutes"
         v-bind:seconds="seconds">
     <StartStop v-bind:timername="timername" v-bind:stop="1" />
   </Timer>
 `
});
Vue.component("TimerStartCount", {
 props : [
   "timername",
    "minutes",
   "seconds"
 ],
 data() {
   return {
     count : 0
   }
 },
 methods : {
   startstop(stop) {
     if (!stop) this.count++;
   }
 },
 template : `
```

```
    <div>
     <Timer v-bind:timername="timername"
            v-bind:minutes="minutes"
          v-bind:seconds="seconds" v-on:startstop="startstop">
      <StartStop v-bind:timername="timername" v-bind:stop="1" />
    </Timer>
    <br>
    <b>Starts</b> : {{count}}
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <TimerStartCount timername="Timer 4" minutes="1"
seconds="15">
      </TimerStartCount>
      <hr><br>
      <TimerStartCount timername="Timer 5" minutes="0" seconds="5">
      </TimerStartCount>
    </div>
  `
});
```

When the startstop event is received by the Timer component, the startstop(stop) method is activated in which the stop parameter indicates whether the timer is stopped or not. If the timer is running, the reactive count variable is displayed which takes account of all the starts.

Here we use two timers in the page in order to count separately the number of starts of the two timers.



Count the number of starts of all timers

In addition to counting the number of individual starts of each timer, it is also possible to count the total number of starts.

It suffices that each TimerStartStop component of the page listens for the startstop event, and increments a reactive variable during each start click on one of the timers of the page. It's the same principle as the previous component but using this principle in the Vue object directly.

Only one problem will be encountered: the TimerStartStop component does not receive the startstop event, because it is only the Timer component (included in the TimerStartStop component) that receives it. The Timer component would therefore have to send this event back to its parent to prevent it, which is done by the this.$parent.$emit("startstop", stop) statement performed when this event is received in the

Timer component.

> An event received by a component does not go up by itself in the hierarchy of Vue components, we must send it ourselves to the parent so that it processes it if necessary.

Display the total number of starts of all timers

```
Vue.component("Timer", {
  data() {
    return {
        min : this.minutes,
      sec : this.seconds
    }
  },
  props : [
    "timername",
     "minutes",
    "seconds"
  ],
  computed : {
    time() {
      var min = this.min;
      var sec = this.sec;
      if (min < 10) min = "0" + min;
      if (sec < 10) sec = "0" + sec;
      return min + ":" + sec;
    }
  },
  mounted() {
    var timer;
    this.$on("startstop", function(stop) {
      if (stop) {
        if (timer) clearInterval(timer);
```

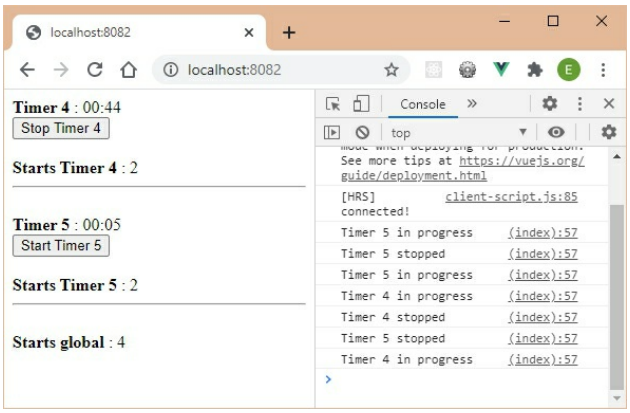
```
    }
    else {
      timer = setInterval(() => {
        this.sec -= 1;
        if (this.sec < 0) {
          this.min -= 1;
          if (this.min < 0) {
            this.min = this.minutes;
            this.sec = this.seconds;
            if (this.$children.length) this.$children[0].start_stop();
            else this.$emit("startstop", 0);
            clearInterval(timer);
          }
          else this.sec = 59;
        }
      }, 1000);
    }
    console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
    this.$parent.$emit("startstop", stop);
  });
  if (!this.$children.length) this.$emit("startstop", 0);
 },
 template : `
    <div>
     <b>{{timername}}</b> :
     <span v-if="sec||min"> {{time}} </span>
     <span v-else>End timer</span>
     <slot v-bind:timername="timername"></slot>
   </div>
 `
});
Vue.component("StartStop", {
 data() {
   return {
     stopped : parseInt(this.stop)
   }
```

```
  },
  props : [
     "timername",
    "stop"
  ],
  created() {
    if (!this.stopped) {
      this.stopped = !this.stopped;
      this.start_stop();
    }
  },
  methods : {
    start_stop() {
      this.$nextTick(function() {
        this.stopped = !this.stopped;
        this.$parent.$emit("startstop", this.stopped);
      });
    }
  },
  template : `
     <div>
      <button v-if="stopped" v-on:click="start_stop">
          Start {{timername}}
      </button>
      <button v-else v-on:click="start_stop">
          Stop {{timername}}
      </button>
    </div>
  `
});
Vue.component("TimerStartStop", {
  props : [
    "timername",
     "minutes",
    "seconds"
  ],
```

```
 template : `
    <Timer v-bind:timername="timername"
           v-bind:minutes="minutes"
         v-bind:seconds="seconds">
     <StartStop v-bind:timername="timername" v-bind:stop="0" />
   </Timer>
 `
});
Vue.component("TimerStartCount", {
 props : [
   "timername",
    "minutes",
   "seconds"
 ],
 data() {
   return {
     count : 0
   }
 },
 methods : {
   startstop(stop) {
     if (!stop) this.count++;
   }
 },
 template : `
   <div>
    <Timer v-bind:timername="timername"
           v-bind:minutes="minutes"
         v-bind:seconds="seconds" v-on:startstop="startstop">
     <StartStop v-bind:timername="timername" v-bind:stop="1" />
   </Timer>
   <br>
   <b>Starts {{timername}}</b> : {{count}}
   </div>
 `
});
```

```
var vm = new Vue({
  el : "#root",
  data() {
    return {
      count : 0
    }
  },
  methods : {
    startstop(stop) {
      if (!stop) this.count++;
    }
  },
  template : `
    <div>
      <TimerStartCount timername="Timer 4" minutes="1" seconds="15"
                       v-on:startstop="startstop" >
      </TimerStartCount>
      <hr><br>
      <TimerStartCount timername="Timer 5" minutes="0" seconds="5"
                       v-on:startstop="startstop">
      </TimerStartCount>
      <hr><br>
      <b>Starts global</b> : {{count}}
    </div>
  `
});
```

The global number of starts is updated in the reactive variable count of the Vue object, displayed in the page, when each startstop event is received (if the stop parameter is equal to 0).



TimerStartCounts component

The TimerStartCounts component is a generalization of the previous program. This component allows to encapsulate several TimerStartCount components and to display below the total number of starts of all timers.

In order to be able to count the total number of starts, we allow each TimerStartCount component to send a startstop message to its parent component, here the TimerStartCounts component. The startstop message therefore moves up the component hierarchy from the StartStop component.

TimerStartCounts component

```
Vue.component("Timer", {
  data() {
    return {
        min : this.minutes,
      sec : this.seconds
    }
  },
  props : [
    "timername",
```

```
    "minutes",
   "seconds"
 ],
 computed : {
   time() {
     var min = this.min;
     var sec = this.sec;
     if (min < 10) min = "0" + min;
     if (sec < 10) sec = "0" + sec;
     return min + ":" + sec;
   }
 },
 mounted() {
   var timer;
   this.$on("startstop", function(stop) {
     if (stop) {
       if (timer) clearInterval(timer);
     }
     else {
       timer = setInterval(() => {
         this.sec -= 1;
         if (this.sec < 0) {
           this.min -= 1;
           if (this.min < 0) {
             this.min = this.minutes;
             this.sec = this.seconds;
             if (this.$children.length) this.$children[0].start_stop();
             else this.$emit("startstop", 0);
             clearInterval(timer);
           }
           else this.sec = 59;
         }
       }, 1000);
     }
     console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
     this.$parent.$emit("startstop", stop);
```

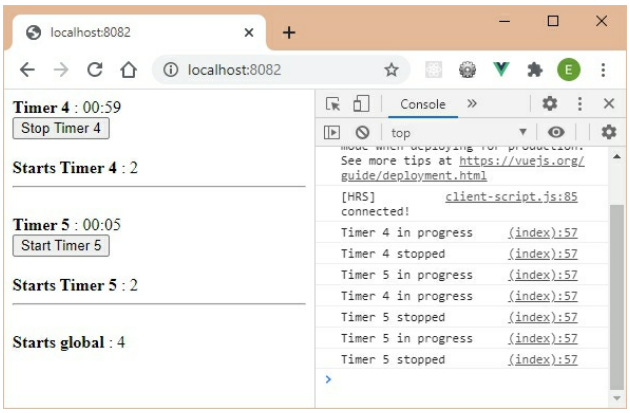
```
    });
    if (!this.$children.length) this.$emit("startstop", 0);
  },
  template : `
     <div>
      <b>{{timername}}</b> :
      <span v-if="sec||min"> {{time}} </span>
      <span v-else>End of timer</span>
      <slot v-bind:timername="timername"></slot>
    </div>
  `
});
Vue.component("StartStop", {
  data() {
    return {
      stopped : parseInt(this.stop)
    }
  },
  props : [
     "timername",
    "stop"
  ],
  created() {
    if (!this.stopped) {
      this.stopped = !this.stopped;
      this.start_stop();
    }
  },
  methods : {
    start_stop() {
      this.$nextTick(function() {
        this.stopped = !this.stopped;
        this.$parent.$emit("startstop", this.stopped);
      });
    }
  },
```

```
  template : `
     <div>
     <button v-if="stopped" v-on:click="start_stop">
         Start {{timername}}
     </button>
     <button v-else v-on:click="start_stop">
         Stop {{timername}}
     </button>
    </div>
  `
});
Vue.component("TimerStartStop", {
  props : [
    "timername",
     "minutes",
    "seconds"
  ],
  template : `
     <Timer v-bind:timername="timername"
            v-bind:minutes="minutes"
         v-bind:seconds="seconds">
     <StartStop v-bind:timername="timername" v-bind:stop="0" />
    </Timer>
  `
});
Vue.component("TimerStartCount", {
  props : [
    "timername",
     "minutes",
    "seconds"
  ],
  data() {
    return {
      count : 0
    }
  },
```

```
  methods : {
    startstop(stop) {
      if (!stop) this.count++;
      this.$parent.$emit("startstop", stop);
    }
  },
  template : `
    <div>
     <Timer v-bind:timername="timername"
              v-bind:minutes="minutes"
           v-bind:seconds="seconds" v-on:startstop="startstop">
      <StartStop v-bind:timername="timername" v-bind:stop="1" />
    </Timer>
    <br>
    <b>Starts {{timername}}</b> : {{count}}
    </div>
  `
});
Vue.component("TimerStartCounts", {
  data() {
    return {
      count : 0
    }
  },
  created() {
    this.$on("startstop", function(stop) {
      if (!stop) this.count++;
    });
  },
  template : `
    <div>
      <slot></slot>
      <hr><br>
      <b>Starts global</b> : {{count}}
    </div>
  `
```

```
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <TimerStartCounts>
        <TimerStartCount timername="Timer 4" minutes="1"
seconds="15" />
        <hr><br>
        <TimerStartCount timername="Timer 5" minutes="0" seconds="5"
/>
      </TimerStartCounts>
    </div>
  `
});
```

We use the slot in the template of the TimerStartCounts component to allow the integration of any content into the TimerStartCounts component.



Props in components

Props are used to pass values into components. They are used like traditional attributes, but unlike HTML attributes, they can be of any type (character string,

integer, boolean, array, object, or even function).

We have seen that components get and use props, which must be specified in the component's props section. Their names are listed as strings in a table.

However, Vue also offers to check that the type of props transmitted to the component is the expected one. If indeed the component expects a Boolean value for a prop (so true or false) and something else is transmitted, the program will not work.

Use the Elements component to display a list of elements

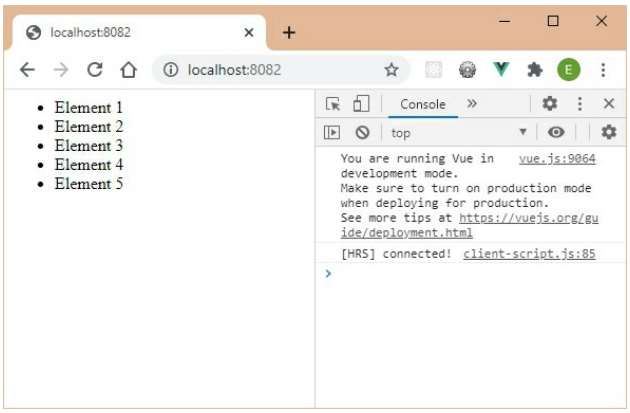
Let's write the Elements component which allows to display a list of elements which is passed to it in the elements props represented by an array of strings.

Display a list of items

```
Vue.component("Elements", {
 props : [
   "elements"
 ],
 template : `
   <div>
     <ul>
       <li v-for="element in elements">
         {{element}}
       </li>
     </ul>
   </div>
```

```
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  },
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
```

The reactive variable elements is passed to the component as an array. It is received in the Elements component via the elements props, to be then traversed as an array by the v-for directive.



What if the elements props passed to the component

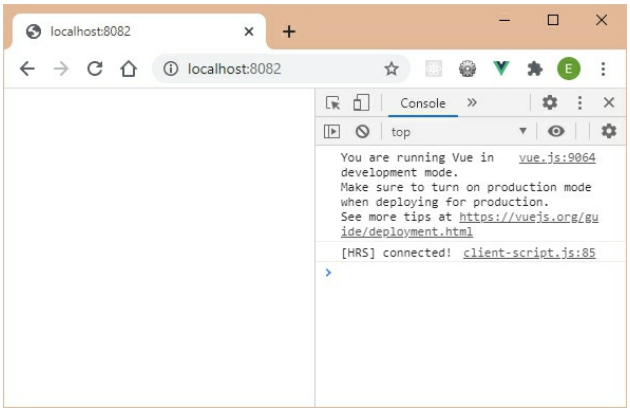
was not an array like above, but something else, for example a boolean value true or false?

Let's modify the reactive elements variable to set it to true.

Set the reactive variable elements to true

```
var vm = new Vue({
  el : "#root",
  data : {
    elements : true
  },
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
```

The result is the following:



Instead of the item list, we have a blank page, and no error message displayed.

To avoid this, and at least get an error message, Vue allows you to check the type of props that are used in

components.

Check the type of props used

Vue allows you to indicate, for each props, its type. The most used are:

- String, for character strings,
- Number, for numeric values,
- Boolean, for Boolean values,
- Array, for arrays,
- Object, for objects,
- Function, for functions,
- Date, for dates.

Here we want to check that the elements props is of type Array. We therefore write in the definition of the Elements component which uses the elements props:

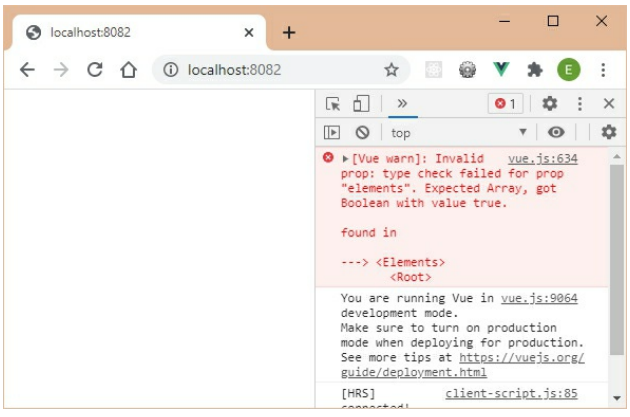
Elements component that checks the type of the elements props

```
Vue.component("Elements", {
 props : {
   "elements" : Array
 },
 template : `
   <div>
     <ul>
       <li v-for="element in elements">
         {{element}}
       </li>
     </ul>
   </div>
```

```
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : true
  },
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
```

The component's props section is now defined as objects (and no longer as an array as before). Each property of the object is a props name, for which the value is the expected type (here the Array type).

Once this change has been made, let's check if it is taken into account:



Vue automatically displays a message in the console indicating that the props type is not the expected one.

Using multiple types for a props

There are cases where a props can have multiple types, which allows more flexibility in using the component.

Suppose the elements props could be of type Array or Boolean, and use each of these possibilities in the Elements component. We then indicate elements: [Array, Boolean] to specify the possible types for this props.

Using an Array or Boolean props

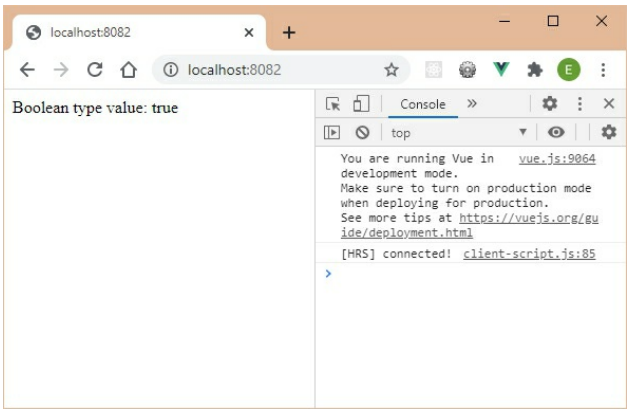
```
Vue.component("Elements", {
 props : {
  "elements" : [Array, Boolean]
 },
 template : `
  <div>
   <ul v-if="elements instanceof Array">
    <li v-for="element in elements">
     {{element}}
    </li>
   </ul>
   <div v-else-if="typeof elements == 'boolean'">
    Boolean type value: {{elements}}</div>
   <div v-else>Value of unknown type: {{elements}}</div>
  </div>
 `
});
var vm = new Vue({
 el : "#root",
 data : {
  elements : true
 },
```

```
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
```

The Elements component template tests the type of the elements props:

- If it's an Array, it displays the list of elements,
- If it is a Boolean, it displays the Boolean value transmitted,
- Otherwise, it indicates an error.

By passing a boolean (true) in the elements prop:



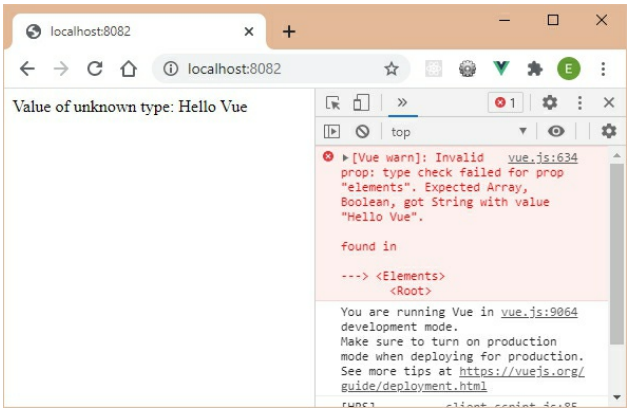
Whereas if we indicate an array of elements, the array is displayed. And if we indicate something other than an array or a Boolean value, for example a String "Hello Vue":

Pass a "Hello Vue" String in the elements props

```
var vm = new Vue({
```

```
  el : "#root",
  data : {
    elements : "Hello Vue"
  },
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
```

Since the String type is not accepted in the component, an error message (warning) is displayed in the console.



The program detected the error (thanks to the v-else directive) and Vue also displays an error message in the console because the type transmitted (here String) is not one of the authorized types.

Another form of writing to define props in components

The previous program works, but does not allow to indicate several criteria for the same props. The value

thus assigned to the props is its type, or an array of possible types.

Vue provides, as we will use in the next section, that other criteria are possible for a props (in addition to its type), for example: its default value, whether it is mandatory or not, etc.

For that, another form of writing to define a props in a component is necessary. We then write the program as follows:

Another form of writing to define a props

```
Vue.component("Elements", {
 props : {
   "elements" : {
    type : Array
  }
 },
 template : `
   <div>
     <ul>
       <li v-for="element in elements">
         {{element}}
       </li>
     </ul>
   </div>
 `
});
var vm = new Vue({
 el : "#root",
 data : {
   elements : true
 },
```

```
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
```

The props elements is now worth an object that contains several properties, including the type property in order to be able to indicate the expected type for the props (which can also be worth an array of types).

We will subsequently enrich this structure by adding new properties for each of the props used.

Specify default values for props

Vue also allows you to specify default values for the props, when these are not transmitted when using the component.

For this we use the default property when defining the props in the component. All you have to do is use the default property, specifying its default value.

> This rule is true only if the props is not an array or an object. In these last two cases, the indicated value must be a function which returns the default value. An example of this is discussed below.

Consider the two cases cited: the default is (or is not) an object or an array.

The default is neither an object nor an array

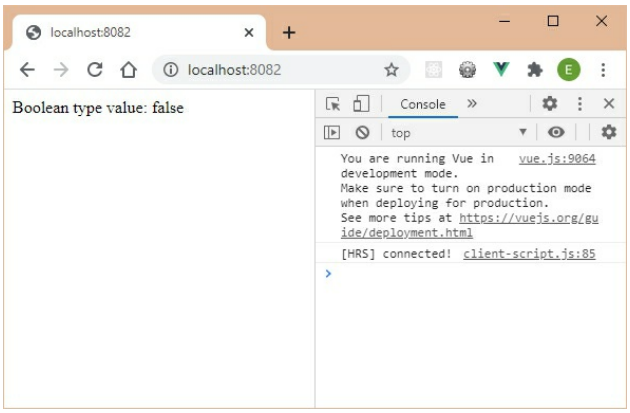
Let us indicate for example that the elements props is worth the Boolean value true by default, in the case where this props is not specified when using the component.

Indicate that the elements props is true by default

```
Vue.component("Elements", {
 props : {
  "elements" : {
   type : [Array, Boolean], default : false
  }
 },
 template : `
  <div>
   <ul v-if="elements instanceof Array">
    <li v-for="element in elements">
     {{element}}
    </li>
   </ul>
   <div v-else-if="typeof elements == 'boolean'">
     Boolean type value: {{elements}}</div>
   <div v-else>Value of unknown type: {{elements}}</div>
  </div>
 `
});
var vm = new Vue({
 el : "#root",
 data : {
 },
 template : `
  <div>
   <Elements />
```

```
  </div>
 `
});
```

The Elements component is used here without specifying the elements props. This props will take the value false in this case.



The default is an object or an array

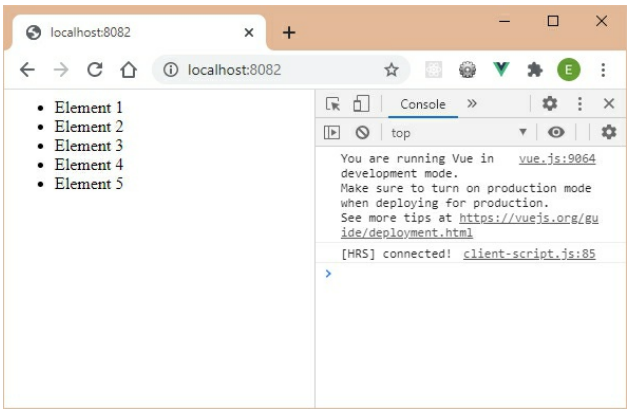
In the case where the default value is an object or an array, the value of the default property is a function that returns the object or the array (otherwise Vue produces an error indicating to do so).

Suppose by default we want to display a list of five elements located in an array. We therefore write:

Indicate that the elements props is ["Element 1",…, "Element 5"] by default

```
Vue.component("Elements", {
 props : {
   "elements" : {
      type : [Array, Boolean],
    default : function() {
```

```
      return [
          "Element 1",
          "Element 2",
          "Element 3",
          "Element 4",
        "Element 5"
        ]
    }
  }
 },
 template : `
   <div>
     <ul v-if="elements instanceof Array">
       <li v-for="element in elements">
         {{element}}
       </li>
     </ul>
     <div v-else-if="typeof elements == 'boolean'">
        Boolean type value: {{elements}}</div>
     <div v-else>Value of unknown type: {{elements}}</div>
   </div>
 `
});
var vm = new Vue({
 el : "#root",
 data : {
 },
 template : `
   <div>
     <Elements />
   </div>
 `
});
```



Indicate that a props is required

The required property set to true in a props definition is used to indicate that the props is mandatory, and it must therefore be specified when using the component.

> In fact, even if a default value is set on this property, the fact of not indicating its value when using the component causes an error displayed in the console.

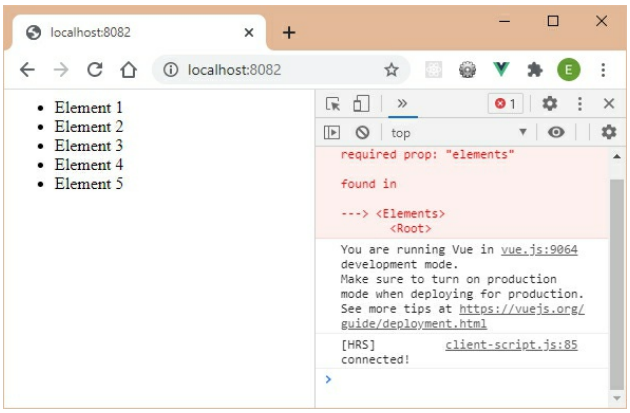
Indicate that the elements props is mandatory

```
Vue.component("Elements", {
 props : {
    "elements" : {
      type : [Array, Boolean],
    required : true,
    default : function() {
      return [
            "Element 1",
            "Element 2",
            "Element 3",
            "Element 4",
        "Element 5"
         ]
    }
```

```
    }
  },
  template : `
    <div>
      <ul v-if="elements instanceof Array">
        <li v-for="element in elements">
          {{element}}
        </li>
      </ul>
      <div v-else-if="typeof elements == 'boolean'">
         Boolean type value: {{elements}}</div>
      <div v-else>Value of unknown type: {{elements}}</div>
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : ['Element 1']
  },
  template : `
    <div>
      <Elements />
    </div>
  `
});
```

The value of the elements props is not specified, but its mention is mandatory (required is true).

The list is displayed (thanks to the default value indicated), but the console displays an error message (in fact a warning only).

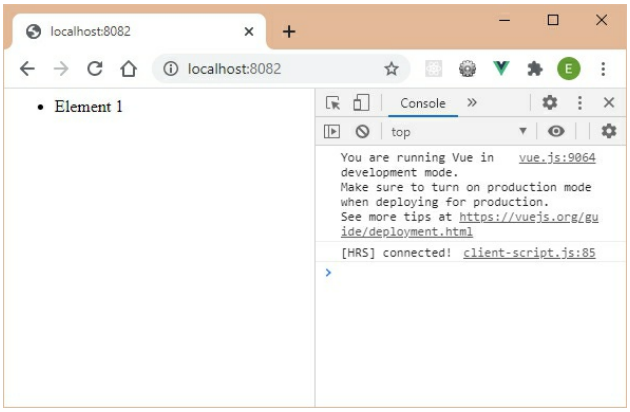


If you specify a value for the elements props while using the Elements component, the error message in the console disappears.

Use a value for the elements props

```
Vue.component("Elements", {
 props : {
    "elements" : {
       type : [Array, Boolean],
    required : true,
    default : function() {
      return [
           "Element 1",
           "Element 2",
           "Element 3",
           "Element 4",
        "Element 5"
         ]
    }
  }
 },
 template : `
  <div>
    <ul v-if="elements instanceof Array">
      <li v-for="element in elements">
        {{element}}
```

```
      </li>
    </ul>
    <div v-else-if="typeof elements == 'boolean'">
      Boolean type value: {{elements}}</div>
    <div v-else>Value of unknown type: {{elements}}</div>
  </div>
 `
});
var vm = new Vue({
 el : "#root",
 data : {
  elements : ['Element 1']
 },
 template : `
  <div>
   <Elements v-bind:elements="elements" />
  </div>
 `
});
```



The list sent as props is displayed and Vue no longer displays the error message in the console.

Perform specific validation

In addition to indicating that a props is mandatory, we

can also indicate that its value corresponds to certain criteria chosen by the programmer.

To do this, Vue allows, thanks to the validator property, to set a validation function which takes as parameter the current value of the props. This value of the props is then analyzed by the validation function which decides whether it is correct or not:

- If the props is considered correct, the validation function returns true,
- If the props is found to be incorrect, the validation function returns false and an error message is displayed in the console (in fact a warning which does not prevent the program from continuing to operate).

Let's use the validator property to indicate that the elements props must contain at least two elements in the list.

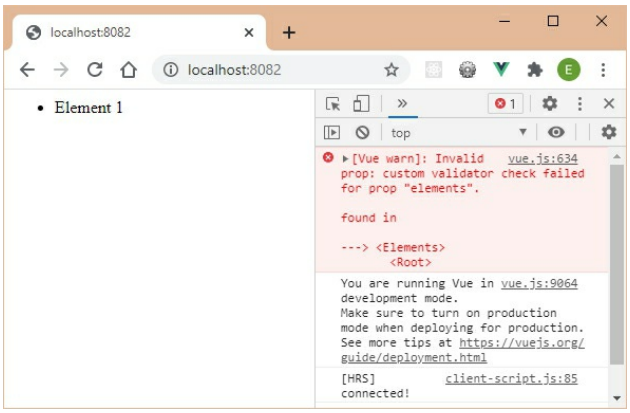
Use the validator property to verify that the elements prop contains at least two elements

```
Vue.component("Elements", {
  props : {
     "elements" : {
       type : [Array, Boolean],
     required : true,
     default : function() {
       return [
            "Element 1",
```

```
            "Element 2",
            "Element 3",
            "Element 4",
          "Element 5"
           ]
      },
      validator(value) {
        return (value.length >= 2);
      }
    }
  },
  template : `
    <div>
      <ul v-if="elements instanceof Array">
        <li v-for="element in elements">
          {{element}}
        </li>
      </ul>
      <div v-else-if="typeof elements == 'boolean'">
        Boolean type value: {{elements}}</div>
      <div v-else>Value of unknown type: {{elements}}</div>
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      "Element 1"
    ]
  },
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
```

```
});
```

Here we use the elements props with only one element in the list. An error is displayed in the console (but the list is still displayed).

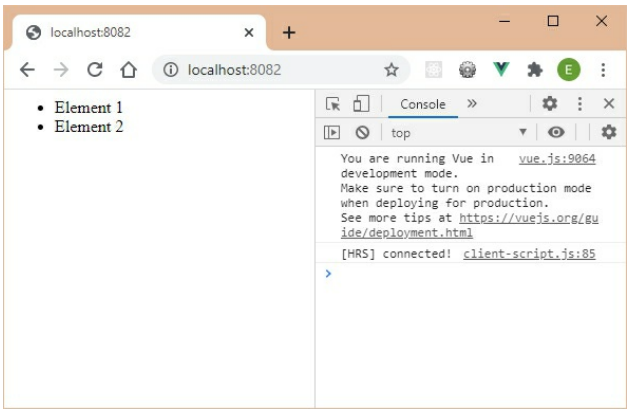


While if we add a second element in the list:

Use a list with two list items

```
var vm = new Vue({
  el : "#root",
  data : {
   elements : [
      "Element 1",
      "Element 2"
    ]
  },
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
```

The list is displayed and the error message has disappeared from the console.



Component lifecycle

We saw previously some methods of the lifecycle of a component. For that, we used the created() and mounted() methods, which we had defined in the component.

These methods make it possible to carry out particular treatments at precise moments in the life of the component. For example, we observed that as long as the component is not "mounted", it was impossible to access its parent in the tree structure.

Let's take a look at the methods that Vue calls in each component, depending on whether the component is created, updated, or destroyed.

To do this, we will create an App component in the template of the Vue object. We will enrich the component with the methods of the lifecycle as we go.

App component used to display the lifecycle

```
Vue.component("App", {
```

```
  data() {
    return {
      count : 0
    }
  },
  template : `
    <div>
      App component: count = {{count}}
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App />
    </div>
  `
});
```

The App component uses a reactive count variable that will be incremented every second, allowing you to observe the lifecycle methods that handle the component update.

Component creation

When a component is created, the following methods are called in the component:

- beforeCreate() before creating the component,
- created() after creating the component,
- beforeMount() before the component is attached in the DOM tree,

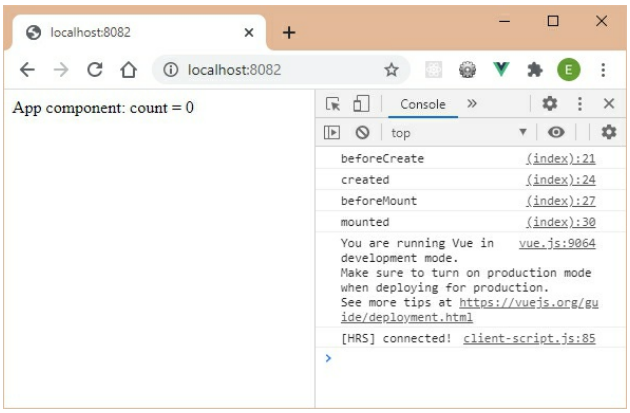
- mounted() after the component is attached in the DOM tree.

Let's integrate these methods into the App component. A message is displayed for each of them in the console.

Use lifecycle methods for component creation

```
Vue.component("App", {
 data() {
  return {
   count : 0
  }
 },
 beforeCreate() {
  console.log("beforeCreate");
 },
 created() {
  console.log("created");
 },
 beforeMount() {
  console.log("beforeMount");
 },
 mounted() {
  console.log("mounted");
 },
 template : `
  <div>
   App component: count = {{count}}
  </div>
 `
});
var vm = new Vue({
 el : "#root",
 template : `
```

```
    <div>
      <App />
    </div>
  `
});
```



The name of each of the methods is displayed in the console, showing the sequencing of their call.

Component update

When updating a component (by modifying one of its reactive variables), the following lifecycle methods are also called:

- beforeUpdate() before updating the component,
- updated() after updating the component.

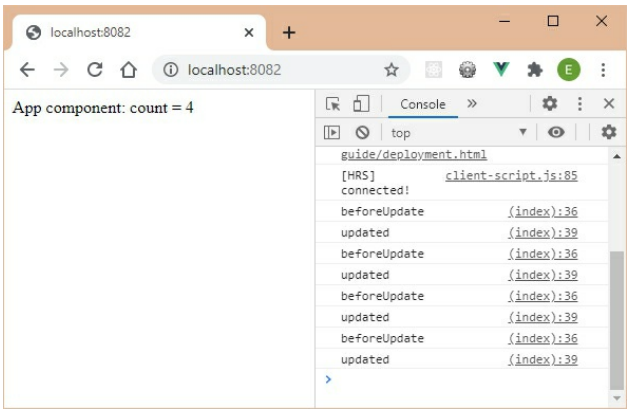
To update the App component, we modify the value of the reactive variable count. To do this, we insert in one of the methods used when creating the component, a timer which increments the count variable by 1 every second. Let's insert the management of this timer in the created() method for example.

Change the value of count every second and display lifecycle update methods

```
Vue.component("App", {
 data() {
  return {
   count : 0
  }
 },
 beforeCreate() {
  console.log("beforeCreate");
 },
 created() {
  console.log("created");
  setInterval(() => {
   this.count++;
  }, 1000);
 },
 beforeMount() {
  console.log("beforeMount");
 },
 mounted() {
  console.log("mounted");
 },
 beforeUpdate() {
  console.log("beforeUpdate");
 },
 updated() {
  console.log("updated");
 },
 template : `
  <div>
   App component: count = {{count}}
  </div>
 `
});
```

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App />
    </div>
  `
});
```

The counter is incremented from second to second, and the beforeUpdate() and updated() methods are called on each update.



Component destruction

When a component is deleted, the following methods are called:

- beforeDestroy() before the component is destroyed,
- destroy() after destroying the component.

The deletion of a component can be done directly by calling its $destroy() method defined on the component

(therefore accessible by this.$destroy()).

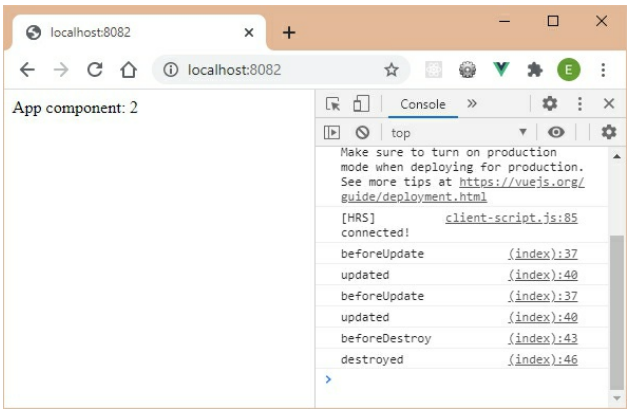
For example, let's call this method when the reactive variable count reaches the value 3. We also implement the beforeDestroy() and destroy() methods in the component.

Component destruction

```
Vue.component("App", {
  data() {
    return {
      count : 0
    }
  },
  beforeCreate() {
    console.log("beforeCreate");
  },
  created() {
    console.log("created");
    setInterval(() => {
      this.count++;
      if (this.count == 3) this.$destroy();
    }, 1000);
  },
  beforeMount() {
    console.log("beforeMount");
  },
  mounted() {
    console.log("mounted");
  },
  beforeUpdate() {
    console.log("beforeUpdate");
  },
  updated() {
```

```
    console.log("updated");
  },
  beforeDestroy() {
    console.log("beforeDestroy");
  },
  destroyed() {
    console.log("destroyed");
  },
  template : `
    <div>
      App component: {{count}}
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App />
    </div>
  `
});
```

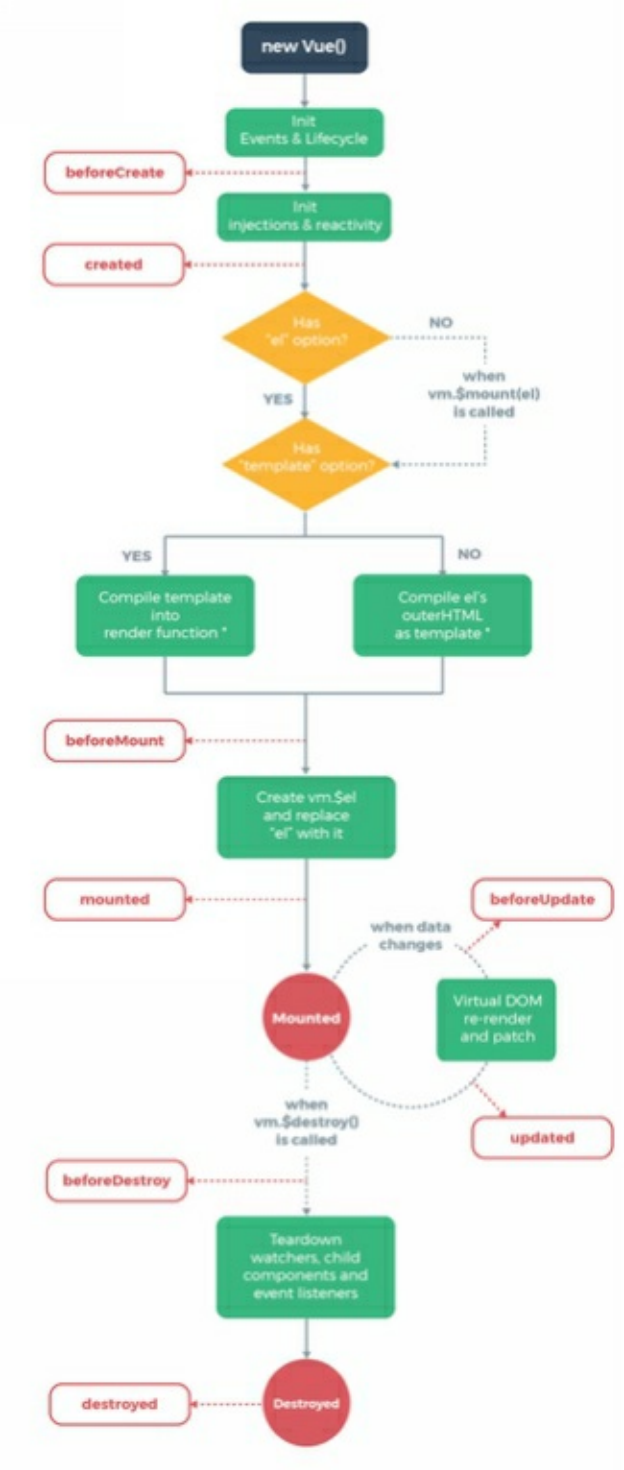
When the value of count reaches the limit, the counting stops and the component is destroyed.



Global scheme

The page https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram shows the overall diagram of the lifecycle of a component, from its creation to its destruction.

We copy here the diagram provided by the official Vue documentation.


new Vue()
Init
Events & Lifecycle
beforeCreate
Init
injections & reactivity
created
Has
"el" option?
NO
when
vm.$mount(el)
is called
YES
Has
"template" option?
YES
NO
Compile template
into
render function *
Compile el's
outerHTML
as template *
beforeMount
Create vm.$el
and replace
"el" with it
mounted
beforeUpdate
when data
changes
Mounted
Virtual DOM
re-render
and patch
when
vm.$destroy()
is called
updated
beforeDestroy
Teardown
watchers, child
components and
event listeners
destroyed
Destroyed

3 – MANAGE THE ELEMENTS OF A LIST AS COMPONENTS

Let's use the previous knowledge to manage a list of items with Vue components. This example repeats the one studied in *Vue.js Basic Concepts*, chapter 3 "*Manage the elements of a list*", but using the components.

We use the usual following structure of the HTML page, in which we create the div#root element which will contain the HTML elements of the template associated with the Vue object.

Basic index.html file

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="root"></div>
</body>
<script>
var vm = new Vue({
```

```
  el : "#root",
  template : `
    <div>
    </div>
  `
});
</script>
</html>
```

The display of this basic page corresponds to a page empty of any content, because the element associated with the template does (for the moment) contain any content.

In order to manage the list as components, we create three of them which are as follows:

- The App component, which is the entire application. It is this component that will be inserted into the div#root element of the HTML page (identified by the el option in the Vue object).
- The Elements component, which is the list of displayed elements.
- The Element component that corresponds to a list item. This list element must turn into an edit control when double-clicked, and must display a Remove button allowing it to be removed from the global list.

> It is traditional to group the entire application into a

> single component (here App) which will be the one registered in the el element managed by Vue.

Let's take a look at the different steps in managing the list:

- Display of the elements of the list,
- Deleting an item from the list,
- Modification of an element of the list,
- Focus in the control being modified,
- Adding an item to the list.

Show items in the list

To display the items in the list we are currently using the App and Elements components. Each list element will be an HTML <li> element, and will be transformed below into an Element component (so as to be progressive in the explanations).

Use the <li> tag to display each list item

Each list item is simply displayed here, without being able to be modified or deleted.

Show items in the list

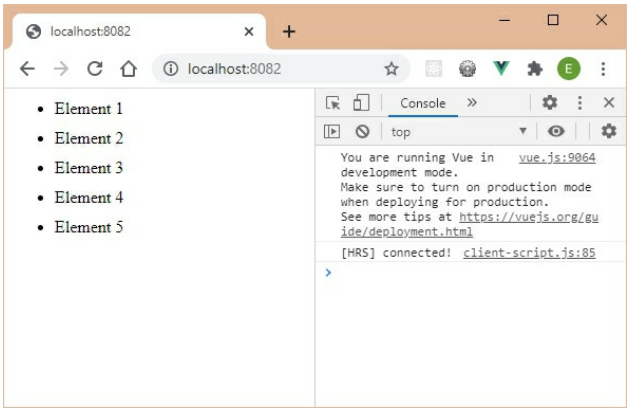
```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
```

```
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  template : `
    <ul v-if="elements.length">
      <li v-for="(element,index) in elements" style="margin-top:10px;">
        {{element.text}}
      </li>
    </ul>
    <div v-else><br>Empty List</div>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
```

The application has a reactive variable which is elements, containing the list of elements to display.

This list is passed as props to the App component, which just displays the Elements component displaying the list also passed as props.

The Elements component is used to display the list of elements transmitted to it as an HTML list. If the list is empty (elements.length == 0) we display the text Empty List, otherwise we display the list elements as <li> using the v-for directive.



Use the Element component to display each list item

Now let's integrate the Element component into our program. Each <li> element becomes an Element component, which itself represents an <li> element.

Use Element component for each list item

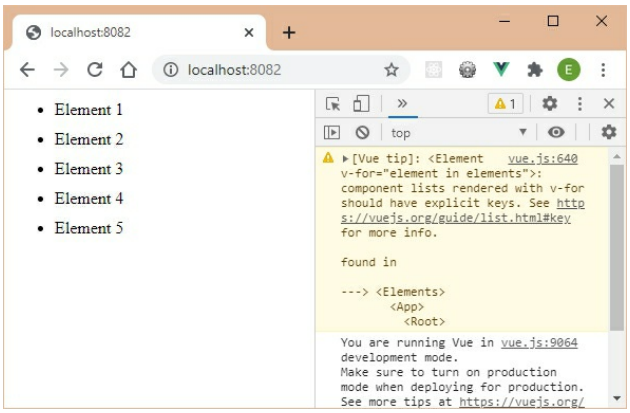
```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
```

```
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  template : `
    <ul v-if="elements.length">
      <Element v-for="(element) in elements" style="margin-top:10px;"
            v-bind:text="element.text" />
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
  props : [
    "text"
  ],
  template : `
    <li style="margin-top:10px;">
      <span>{{text}}</span>
    </li>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
```

```
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
```

The Element component uses the text props which contains the text of the element to display in the list.

Let's check that the operation is identical:



The list is displayed, but Vue displays a message (no error, only a warning) indicating that the elements displayed during the v-for directive must have a unique key identified by the key attribute. This allows Vue to make it easier to update the list in the event of deletion or addition to the list, thanks to a unique identifier on each item in the list.

This identifier can be for example the index of the element in the list. This index is known in the v-for directive using this directive as v-for="(element, index) in elements". Let's use the v-for directive in this form

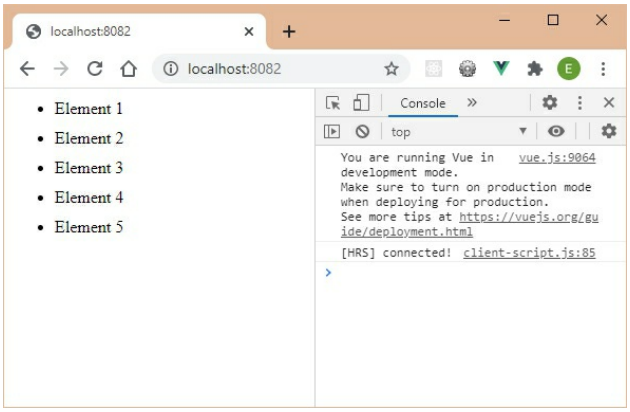
to assign a key attribute to each list item. The Elements component is modified.

Assign a key attribute to each list item

```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  template : `
    <ul v-if="elements.length">
      <Element v-for="(element, index) in elements" style="margin-top:10px;"
               v-bind:key="index"
               v-bind:text="element.text" />
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
  props : [
    "text"
  ],
  template : `
    <li style="margin-top:10px;">
      <span>{{text}}</span>
```

```
    </li>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
```

We check that the prevention message has disappeared:



Remove an item from the list

Let's start by allowing the list to be edited by displaying a Remove button after each list item.

Clicking on this button removes this item from the global list.

Add a delete button for each list item

For now, let's only display text each time we click a Delete button for a list item.

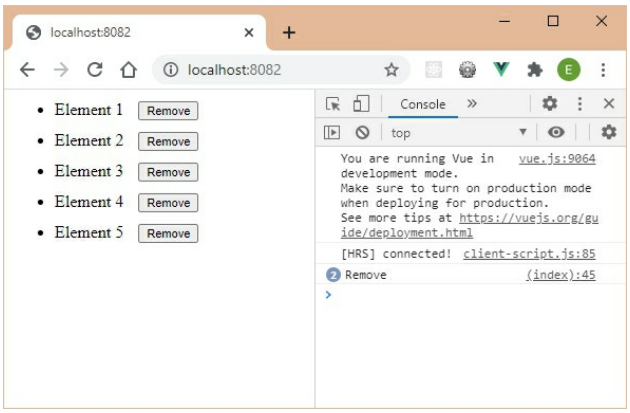
Display a delete button after each list item

```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
              v-bind:key="index"
              v-bind:text="element.text" />
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
```

```
  props : [
    "text"
  ],
  methods : {
    remove() {
      console.log("Remove");
    }
  },
  template : `
    <li style="margin-top:10px;">
      <span>{{text}}</span>
      <button style="margin-left:10px; font-size:11px;"
            v-on:click="remove">
        Remove
      </button>
    </li>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
```

Clicking on the Remove button calls the remove() method defined in the Element component. This

method currently displays the text "Remove" in the console.



Find out the index of the element to delete

To perform the deletion, you must know the element to be deleted. This element is known by its index, which is stored in the key attribute. However, if we use this key property in our programs, Vue produces an error because this attribute is reserved for its internal use.

The easiest way is to add a new props in the component which will have the value of the element's index, and will be named for example index.

Adding index props in Element component

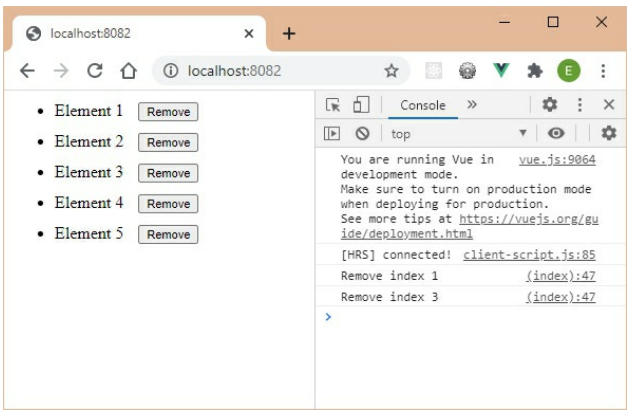
```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
```

```
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
                 v-bind:key="index"
                 v-bind:index="index"
                 v-bind:text="element.text" />
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
  props : [
    "text",
    "index"
  ],
  methods : {
    remove() {
      console.log("Remove index " + this.index);
    }
  },
  template : `
    <li style="margin-top:10px;">
      <span>{{text}}</span>
      <button style="margin-left:10px; font-size:11px;"
              v-on:click="remove">
        Remove
      </button>
    </li>
  `
```

```
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
```

The props index is passed in the props of the Element component, and is used in the remove() method by means of this.index. This therefore allows you to know the element that you want to delete.

After several clicks on the Remove buttons from the displayed list, the console displays the index of the elements clicked:



Remove item from list

The element must now be really deleted from the list. To do this, we must update the reactive list of elements that is stored in the Vue object (elements property). Removing an item from this list will display the list again because it is reactive.

We will see two ways to delete an element in the list of elements of the list stored in the Vue object:

- Using the vm.elements property directly,
- Using events.

Directly use the vm.elements property

The remove() method defined in the Element component can access the list of elements of the Vue object by means of vm.elements. The filter() method used on this array allows you to indicate the elements to keep or to delete in the result array.

Delete elements directly in vm.elements

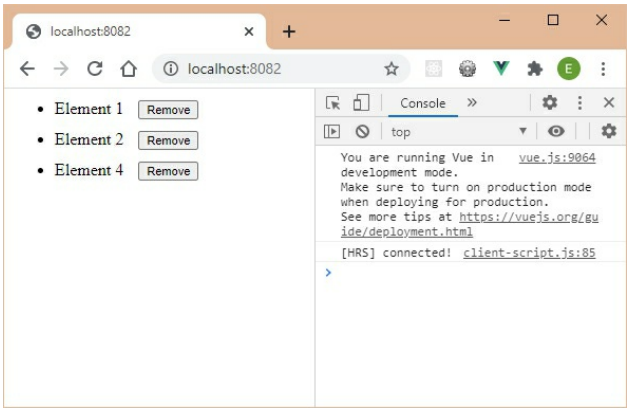
```
Vue.component("App", {
 props : [
  "elements"
 ],
 template : `
  <div>
   <Elements v-bind:elements="elements" />
  </div>
 `
```

```
});
Vue.component("Elements", {
 props : [
  "elements"
 ],
 template : `
  <ul v-if="elements.length">
     <Element v-for="(element, index) in elements" style="margin-
top:10px;"
           v-bind:key="index"
           v-bind:index="index"
           v-bind:text="element.text" />
  </ul>
  <div v-else><br>Empty List</div>
 `
});
Vue.component("Element", {
 props : [
  "text",
  "index"
 ],
 methods : {
  remove() {
   vm.elements = vm.elements.filter((element, index) => {
    if (index == this.index) return false;
    else return true;
   });
  }
 },
 template : `
  <li style="margin-top:10px;">
   <span>{{text}}</span>
   <button style="margin-left:10px; font-size:11px;"
        v-on:click="remove">
    Remove
   </button>
```

```
    </li>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
```

The callback function used in the filter() method is used here in ES6 in order to keep the value of this and allow to retrieve the value of this.index in the body of the callback function.

Let's remove Element 3 and Element 5 from the list:



Use events

Rather than directly accessing the Vue object (via the vm object) in the methods of a component, Vue recommends using the event bus, called "Event Bus".

Indeed, Vue allows you to send or receive any events, by using the methods $on() (to receive) or $emit() (to send).

Here it is necessary to have a centralized management of the events received or to be transmitted, so as to manage the list of elements in a centralized manner. We will therefore globally create an additional Vue object called eventBus, which will only be used to send or receive events to manage the vm.elements list:

- To emit an event managing the list, we will use eventBus.$emit(),
- To receive an event managing the list, we will use eventBus.$on().

Let's use the event bus to handle removing items from the list.

Use an event bus to remove an item from the list

```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
    <div>
```

```
    <Elements v-bind:elements="elements" />
   </div>
  `
});
Vue.component("Elements", {
  props : [
   "elements"
  ],
  template : `
   <ul v-if="elements.length">
      <Element v-for="(element, index) in elements" style="margin-
top:10px;"
            v-bind:key="index"
            v-bind:index="index"
            v-bind:text="element.text" />
   </ul>
   <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
  props : [
   "text",
   "index"
  ],
  methods : {
   remove() {
    eventBus.$emit("remove", this.index);
   }
  },
  template : `
   <li style="margin-top:10px;">
    <span>{{text}}</span>
    <button style="margin-left:10px; font-size:11px;"
          v-on:click="remove">
     Remove
    </button>
```

```
    </li>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
```

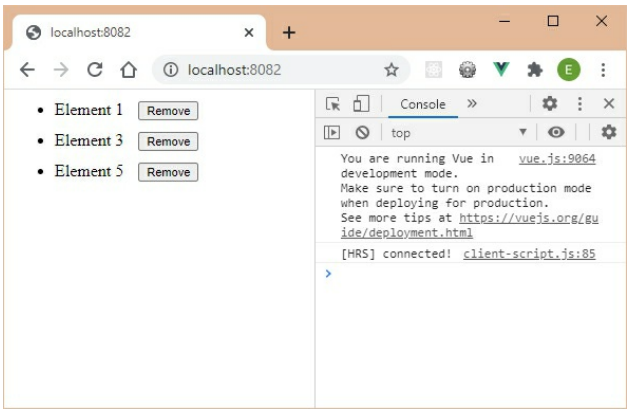
The eventBus object is global for the entire program and is used to globally manage the events received, here the remove events.

Vue allows you to specify any number of parameters when calling $emit(), these parameters being received in the processing callback function of the $on() method. The type of parameters is also arbitrary. It

suffices, during the $emit(), to indicate them in the order in which they will be processed by $on().

> The event bus is used to centralize the management of the reactive variable vm.elements only when the corresponding events are received by eventBus.$on().

Let's check by removing a few elements that the operation is identical:



Modify an item in the list

A list item is modified by double-clicking on this item, which transforms it into an edit control. Pressing Enter or exiting the entry field validates this field, and the list is redisplayed with the new element.

To do this, we introduce in each Element component a reactive modifyOn variable indicating whether the list element is being modified (modifyOn is true) or not (modifyOn is false).

> The Element component template uses this reactive

modifyOn variable to determine whether the element should be displayed as an edit field or as text.

Double click on a list item taken into account

Let's use the modifyOn variable in the Element component to manage the display of the element as an input field or as text.

Use modifyOn variable in Element component

```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
                 v-bind:key="index"
                 v-bind:index="index"
                 v-bind:text="element.text" />
    </ul>
```

```
  <div v-else><br>Empty List</div>
 `
});
Vue.component("Element", {
 props : [
  "text",
  "index"
 ],
 data() {
  return {
   modifyOn : false
  }
 },
 methods : {
  remove() {
   eventBus.$emit("remove", this.index);
  }
 },
 template : `
  <li style="margin-top:10px;">
   <input v-if="modifyOn" type="text" v-bind:value="text">
   <div v-else>
    <span v-on:dblclick="modifyOn=true">{{text}}</span>
    <button style="margin-left:10px; font-size:11px;"
          v-on:click="remove">
     Remove
    </button>
   </div>
  </li>
 `
});
var vm = new Vue({
 el : "#root",
 data : {
  elements : [
   { text : "Element 1" },
```

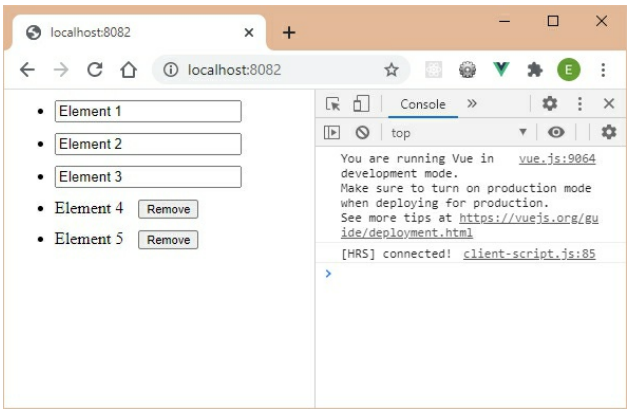
```
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
```

We add a data section in the Element component, which will contain the reactive modifyOn variable.

Depending on the value of modifyOn, the v-if directive displays either an input field <input> or text in the form of . Double clicking on the changes modifyOn to true, which redisplays the corresponding Element component in the form of an edit field (due to the reactivity).

The input field is initialized with the value of the text props passed into the component, by means of the v-bind directive.

Double click on some items in the list. They turn into input fields.



Validation of the entry by Enter or exit the field

We add the keydown.enter and blur events to the edit control to manage the output of the control. This activates the modify() method which resets the modifyOn variable to false to remove the input field.

Taking into account of the keydown.enter and blur events on the edit control

```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
```

```
 template : `
   <ul v-if="elements.length">
      <Element v-for="(element, index) in elements" style="margin-
top:10px;"
            v-bind:key="index"
            v-bind:index="index"
            v-bind:text="element.text" />
   </ul>
   <div v-else><br>Empty List</div>
 `
});
Vue.component("Element", {
 props : [
   "text",
   "index"
 ],
 data() {
   return {
     modifyOn : false
   }
 },
 methods : {
   remove() {
     eventBus.$emit("remove", this.index);
   },
   modify() {
     console.log("modify");
     this.modifyOn = false;
   }
 },
 template : `
   <li style="margin-top:10px;">
      <input v-if="modifyOn" type="text"
         v-on:keydown.enter="modify"
         v-on:blur="modify"
         v-bind:value="text">
```

```
      <div v-else>
        <span v-on:dblclick="modifyOn=true">{{text}}</span>
        <button style="margin-left:10px; font-size:11px;"
              v-on:click="remove">
          Remove
        </button>
      </div>
    </li>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
```

Editing the item in the responsive item list

It remains to modify the contents of the reactive

variable elements. We use the event bus (eventBus object) to send and receive the modify event, including in the event parameters:

- The index of the element to modify in the elements array,
- The new value of the element (the one entered by the user).

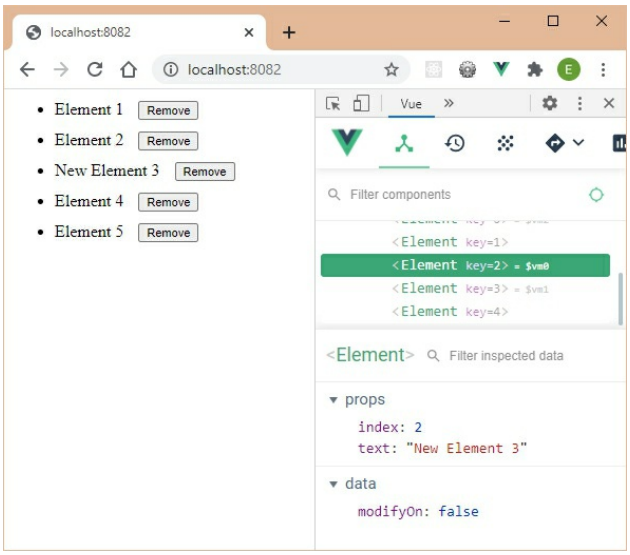
Editing the item in the list

```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
                 v-bind:key="index"
                 v-bind:index="index"
                 v-bind:text="element.text" />
    </ul>
    <div v-else><br>Empty List</div>
```

```
  `
});
Vue.component("Element", {
 props : [
  "text",
  "index"
 ],
 data() {
  return {
   modifyOn : false
  }
 },
 methods : {
  remove() {
   eventBus.$emit("remove", this.index);
  },
  modify(event) {
   this.modifyOn = false;
   var newText = event.target.value;
   eventBus.$emit("modify", this.index, newText);
  }
 },
 template : `
  <li style="margin-top:10px;">
     <input v-if="modifyOn" type="text"
        v-on:keydown.enter="modify"
        v-on:blur="modify"
        v-bind:value="text">
    <div v-else>
     <span v-on:dblclick="modifyOn=true">{{text}}</span>
     <button style="margin-left:10px; font-size:11px;"
          v-on:click="remove">
       Remove
     </button>
    </div>
  </li>
```

```
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
eventBus.$on("modify", function(index, value) {
  vm.elements[index].text = value;
});
```

The modify() method of the Element component is modified to emit the modify event (via the event bus), then this event is received by this same bus that performs the update of vm.elements.



We check in the Vue tab of the console that the elements variable is updated when modifying the text of the element.

Give focus to the element being edited

When you double-click on an element in the list, it becomes an edit control, but you still have to click in this field to be able to modify it. We would therefore like the edit control to be displayed with the focus directly, without having to click in it.

Obtaining the DOM element corresponding to the edit control

The focus() method of JavaScript allows to give the focus to a DOM element, here the input field. To do this, we must retrieve the corresponding DOM element

(<input> element), knowing that we currently have access to the Element component which contains this DOM element.

In the template of a component, Vue allows you to put references using the ref attribute. Each reference registered in the template can then be used thanks to the object this.$refs, which contains in properties the references registered in the template.

For example, if in the template we write:

Adding a reference to the input field

```
<input type="text" ref="input">
```

We can then, in the JavaScript code, access the corresponding DOM element by writing:

Access to the DOM element with the input reference

```
var input = this.$refs.input;
input.focus();
```

Calling the focus() method on the edit control

Once the DOM element has been obtained (here the previous input variable), it remains to give it the focus using the input.focus() method.

However, where in the program should you write this instruction? Indeed, the input control does not always exist, because it is only present after the double click,

when the list of elements has been refreshed and the input control displayed.

The updated() method of the Element component's lifecycle is called when the Element component is redisplayed (with or without the edit field). In the updated() method, all you have to do is test whether this.$refs.input exists (in this case the edit control is visible) and then give it the focus.

Let's use the updated() method in the Element component to give focus to the input field.

Give focus to the edit control

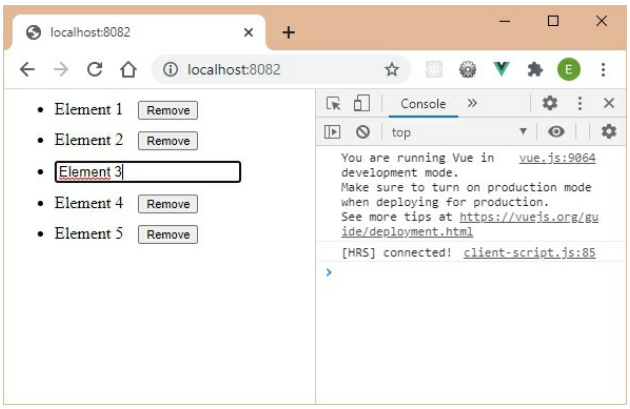
```
Vue.component("App", {
  props : [
    "elements"
  ],
  template : `
    <div>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
              v-bind:key="index"
```

```
            v-bind:index="index"
            v-bind:text="element.text" />
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
  props : [
    "text",
    "index"
  ],
  data() {
    return {
      modifyOn : false
    }
  },
  updated() {
    if (this.$refs.input) this.$refs.input.focus();
  },
  methods : {
    remove() {
      eventBus.$emit("remove", this.index);
    },
    modify(event) {
      this.modifyOn = false;
      var newText = event.target.value;
      eventBus.$emit("modify", this.index, newText);
    }
  },
  template : `
    <li style="margin-top:10px;">
        <input v-if="modifyOn" type="text"
            v-on:keydown.enter="modify"
            v-on:blur="modify"
                v-bind:value="text"
            ref="input">
```

```
      <div v-else>
        <span v-on:dblclick="modifyOn=true">{{text}}</span>
        <button style="margin-left:10px; font-size:11px;"
              v-on:click="remove">
          Remove
        </button>
      </div>
    </li>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
eventBus.$on("modify", function(index, value) {
  vm.elements[index].text = value;
});
```

The input reference is added to the template of the input field, and the updated() method tests whether this reference exists and then gives focus to the corresponding DOM element.

Double click on an item in the list: the edit control gets the focus directly.



Add item to list

Adding a new item to the list is inspired by what we have done previously. We use the event bus to produce the add event which adds the new element. The text associated with the new element corresponds to the number of elements already present in the list: if for example 8 elements are in the list, the caption of the new element will be "Element 9".

The Add Element button is positioned before the list of displayed elements. It is registered in the template of the App component before the Elements component.

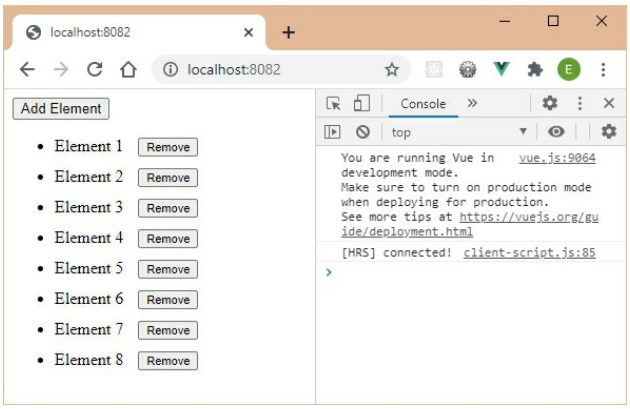
Button add a new item to the list

```
Vue.component("App", {
  props : [
    "elements"
  ],
  methods : {
    add() {
      eventBus.$emit("add");
    }
  },
  template : `
    <div>
      <button v-on:click="add">Add Element</button>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
                 v-bind:key="index"
                 v-bind:index="index"
                 v-bind:text="element.text" />
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
  props : [
    "text",
```

```
  "index"
],
data() {
  return {
    modifyOn : false
  }
},
updated() {
  if (this.$refs.input) this.$refs.input.focus();
},
methods : {
  remove() {
    eventBus.$emit("remove", this.index);
  },
  modify(event) {
    this.modifyOn = false;
    var newText = event.target.value;
    eventBus.$emit("modify", this.index, newText);
  }
},
template : `
  <li style="margin-top:10px;">
      <input v-if="modifyOn" type="text"
          v-on:keydown.enter="modify"
          v-on:blur="modify"
              v-bind:value="text"
          ref="input">
    <div v-else>
      <span v-on:dblclick="modifyOn=true">{{text}}</span>
      <button style="margin-left:10px; font-size:11px;"
            v-on:click="remove">
        Remove
      </button>
    </div>
  </li>
`
```

```
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
eventBus.$on("modify", function(index, value) {
  vm.elements[index].text = value;
});
eventBus.$on("add", function() {
  var text = "Element " + (vm.elements.length + 1);
  vm.elements.push({text});
});
```

Let's click on the add button several times:



Display a context menu to manage list items

We want to deepen our knowledge even further by allowing a context menu to be displayed when a simple click on an item in the list is made. The double click will still work by allowing to modify the double clicked element.

The context menu displays the choice between the following two options:

- Modify: allows you to modify the element,
- Remove: allows you to remove the item.

These are the same actions as those already allowed, but with a different way to access them.

To achieve this, several steps are necessary:

- Take into account the click on a list item and display the contextual menu,
- Be able to delete the menu by clicking

elsewhere in the page,

- Select a menu item and process the corresponding action,
- Style the menu by displaying in red the menu item on which the mouse is.

Let's see these different steps below.

Take into account the click on a list item and display the context menu

To represent a contextual menu, we will use a new ContextMenu component, which will contain an <ul> list with <li> elements for each menu item.

To display the contextual menu at a location on the page, the x and y coordinates of the click made on an element are transmitted to it as props. As the actions to be performed (Modify, Remove) are associated with each element of the list, we create a ContextMenu component in the Element component, following the element representing the text of the element.

Integration of the ContextMenu component in the page

```
Vue.component("App", {
 props : [
  "elements"
 ],
 methods : {
  add() {
```

```
      eventBus.$emit("add");
    }
  },
  template : `
    <div style="position:relative;">
      <button v-on:click="add">Add Element</button>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  data() {
    return {
    }
  },
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
                 v-bind:key="index"
                 v-bind:index="index"
                 v-bind:text="element.text">
      </Element>
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
  props : [
    "text",
    "index"
  ],
  data() {
```

```
  return {
    modifyOn : false,
    displayContextMenu : false,
      x : 0,
    y : 0
  }
},
updated() {
  if (this.$refs.input) this.$refs.input.focus();
},
methods : {
  remove() {
    eventBus.$emit("remove", this.index);
  },
  modify(event) {
    this.modifyOn = false;
    var newText = event.target.value;
    eventBus.$emit("modify", this.index, newText);
  },
  displayCtxMenu(event) {
    this.x = event.clientX;
    this.y = event.clientY;
    this.displayContextMenu = true;
  }
},
template : `
  <li style="margin-top:10px;">
      <input v-if="modifyOn" type="text"
          v-on:keydown.enter="modify"
          v-on:blur="modify"
              v-bind:value="text"
          ref="input">
    <div v-else>
      <span v-on:dblclick="modifyOn=true"
           v-on:click="displayCtxMenu" style="cursor:pointer;">
        {{text}}
```

```
      </span>
      <ContextMenu v-show="displayContextMenu" v-bind:x="x" v-
bind:y="y" />
      <button style="margin-left:10px; font-size:11px;"
            v-on:click="remove">
        Remove
      </button>
    </div>
  </li>
 `
});
Vue.component("ContextMenu", {
 props : [
  "x",
  "y"
 ],
 data() {
  return {
  }
 },
 methods : {
  modify() {
   console.log("Modify");
  },
  remove() {
   console.log("Remove");
  }
 },
 template : `
   <div v-bind:style="{position:'absolute', top:y + 'px', left:x + 'px',
                backgroundColor:'gainsboro'}">
     <ul style="margin-left:0px; padding:5px; list-style-type:none;
           cursor:pointer;">
     <li v-on:click="modify">Modify</li>
     <li v-on:click="remove">Remove</li>
    </ul>
```

```
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
eventBus.$on("modify", function(index, value) {
  vm.elements[index].text = value;
});
eventBus.$on("add", function() {
  var text = "Element " + (vm.elements.length + 1);
  vm.elements.push({text});
});
```

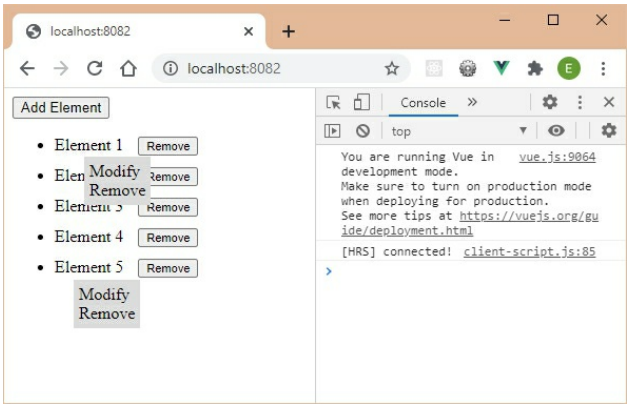
The ContextMenu component uses the x and y props to obtain the coordinates where the menu will be

displayed (in left and top coordinates, knowing that the menu is positioned in absolute.

When clicking on a list item represented by the Element component, the x and y coordinates of the mouse are retrieved using event.clientX and event.clientY. This is handled in the displayCtxMenu() method defined in the Element component.

Note that each Element component internally has a ContextMenu component attached to it. In fact, we do not have a single ContextMenu component for the whole application, even if at any time only one menu is displayed.

Let's click on the first and then the last list item.



We see that the contextual menu is displayed, but remains displayed if we click on a new element. On the other hand, if we move the click on a list element, the menu moves according to the click on this element.

The reasons for this are easy to understand. As each list

item has its context menu, each click displays the corresponding menu. And this menu moves according to the click on the element because the menu is unique for each element (it is the props x and y which are only modified which redisplays the menu in a new location).

Clicking on a new list item must therefore delete the previous menu that may have been displayed, in order to keep only one menu displayed at a time. For this, when clicking, we must explore the other Element components to possibly set their reactive variable displayContextMenu to false.

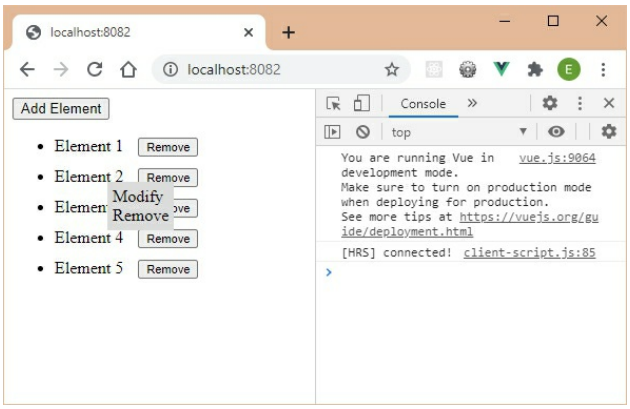
The displayCtxMenu() method which displays the contextual menu when clicked therefore becomes:

Suppress the display of context menus that do not correspond to the clicked item

```
displayCtxMenu(event) {
  this.x = event.clientX;
  this.y = event.clientY;
  this.displayContextMenu = true;
  this.$parent.$children.forEach((child) => {
     if(child.index != this.index)
      child.displayContextMenu = false;
  });
}
```

Using the ES6 notation for the callback function associated with the forEach() method keeps the value of this in the callback function.

We verify that clicks on different list items display only one context menu at a time:



Be able to delete the menu by clicking elsewhere in the page

At this moment, only one contextual menu is displayed at a time, but it should be deleted if we click on a blank place in the page (as we would do in a very natural way in any application). Indeed, for the moment the displayed contextual menu cannot be deleted but only moved when clicking on list items.

As only one contextual menu is now displayed (and it is this one that must be hidden), we memorize the contextual menu currently displayed if there is one:

- To do this, each time a list item is clicked, the ContextMenu component concerned is memorized. We generate a new event called for example displaycontextmenu, which will be retrieved by the event bus, which will be used to

store in a contextmenu variable associated with the Vue object, the ContextMenu component currently displayed. It is the same principle as that used to manage the list of displayed elements.

- When you want to hide the ContextMenu component following a click elsewhere in the page, it will suffice to emit the hidecontextmenu event which will also be processed by the event bus.

The program becomes:

Taking into account the click elsewhere in the page to hide the contextual menu if it is displayed

```
Vue.component("App", {
 props : [
  "elements"
 ],
 methods : {
  add() {
   eventBus.$emit("add");
  },
  click() {
   eventBus.$emit("hidecontextmenu");
  }
 },
 template : `
  <div style="position:relative; height:100%;" v-on:click="click">
   <button v-on:click="add">Add Element</button>
   <Elements v-bind:elements="elements" />
  </div>
 `
```

```
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  data() {
    return {
    }
  },
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
                 v-bind:key="index"
                 v-bind:index="index"
                 v-bind:text="element.text">
      </Element>
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
  props : [
    "text",
    "index"
  ],
  data() {
    return {
      modifyOn : false,
      displayContextMenu : false,
        x : 0,
      y : 0
    }
  },
  updated() {
    if (this.$refs.input) this.$refs.input.focus();
```

```
  },
  methods : {
    remove() {
      eventBus.$emit("remove", this.index);
    },
    modify(event) {
      this.modifyOn = false;
      var newText = event.target.value;
      eventBus.$emit("modify", this.index, newText);
    },
    displayCtxMenu(event) {
      this.x = event.clientX;
      this.y = event.clientY;
      this.displayContextMenu = true;
      this.$parent.$children.forEach((child) => {
          if(child.index != this.index)
         child.displayContextMenu = false;
      });
      eventBus.$emit("displaycontextmenu", this.$children[0]);
    }
  },
  template : `
    <li style="margin-top:10px;">
       <input v-if="modifyOn" type="text"
           v-on:keydown.enter="modify"
           v-on:blur="modify"
               v-bind:value="text"
           ref="input">
      <div v-else>
          <span v-on:dblclick="modifyOn=true"
            v-on:click.stop="displayCtxMenu" style="cursor:pointer;">
        {{text}}
       </span>
       <ContextMenu v-show="displayContextMenu" v-bind:x="x" v-
bind:y="y" />
       <button style="margin-left:10px; font-size:11px;"
```

```
            v-on:click="remove">
        Remove
      </button>
    </div>
  </li>
 `
});
Vue.component("ContextMenu", {
 props : [
  "x",
  "y"
 ],
 data() {
  return {
  }
 },
 methods : {
  modify() {
   console.log("Modify");
  },
  remove() {
   console.log("Remove");
  }
 },
 template : `
   <div v-bind:style="{position:'absolute', top:y + 'px', left:x + 'px',
                backgroundColor:'gainsboro'}">
    <ul style="margin-left:0px; padding:5px; list-style-type:none;
cursor:pointer;">
      <li v-on:click="modify">Modify</li>
      <li v-on:click="remove">Remove</li>
    </ul>
  </div>
 `
});
var vm = new Vue({
```

```
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
eventBus.$on("modify", function(index, value) {
  vm.elements[index].text = value;
});
eventBus.$on("add", function() {
  var text = "Element " + (vm.elements.length + 1);
  vm.elements.push({text});
});
eventBus.$on("displaycontextmenu", function(contextmenu) {
  vm.contextmenu = contextmenu;
});
eventBus.$on("hidecontextmenu", function() {
   if (vm.contextmenu)
    vm.contextmenu.$parent.displayContextMenu = false;
});
```

Some remarks are necessary:

- In order for the click to be taken into account on the entire visible page, it suffices for the enclosing the application to be set with the CSS property height:100% (which is found in the App component).
- We use the stop modifier when clicking on a list element (Element component). Indeed, the propagation of the click event must be prevented in this case, otherwise the menu would never be displayed (it would be displayed and then immediately hidden by the hidecontextmenu event).

After clicking on different items in the list, the menu hides by clicking on a blank place on the page.

Select a menu item and process the corresponding action

You must now process the action when you select an item from the context menu (Modify or Remove). For now, we simply display the caption of the action in the console, using the modify() and remove() methods defined in the ContextMenu component.

It suffices that in these methods, a message is sent to the parent component (here Element), so that it

performs the corresponding actions (modify or delete the list element). The Element component will therefore have to listen to the two events (here called modify and remove) sent from the ContextMenu component.

Generate the modify and remove events and take them into account

```
Vue.component("App", {
  props : [
    "elements"
  ],
  methods : {
    add() {
      eventBus.$emit("add");
    },
    click() {
      eventBus.$emit("hidecontextmenu");
    }
  },
  template : `
    <div style="position:relative; height:100%;" v-on:click="click">
      <button v-on:click="add">Add Element</button>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  data() {
    return {
```

```
    }
  },
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
                 v-bind:key="index"
                 v-bind:index="index"
                 v-bind:text="element.text">
      </Element>
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
  props : [
    "text",
    "index"
  ],
  data() {
    return {
      modifyOn : false,
      displayContextMenu : false,
        x : 0,
      y : 0
    }
  },
  created() {
    this.$on("modify", function() {
      this.modifyOn = true;
      this.displayContextMenu = false;
    });
    this.$on("remove", function() {
      this.remove();
      this.displayContextMenu = false;
    });
```

```
},
updated() {
  if (this.$refs.input) this.$refs.input.focus();
},
methods : {
  remove() {
    eventBus.$emit("remove", this.index);
  },
  modify(event) {
    this.modifyOn = false;
    var newText = event.target.value;
    eventBus.$emit("modify", this.index, newText);
  },
  displayCtxMenu(event) {
    this.x = event.clientX;
    this.y = event.clientY;
    this.displayContextMenu = true;
    this.$parent.$children.forEach((child) => {
        if(child.index != this.index)
       child.displayContextMenu = false;
    });
    eventBus.$emit("displaycontextmenu", this.$children[0]);
  }
},
template : `
  <li style="margin-top:10px;">
      <input v-if="modifyOn" type="text"
          v-on:keydown.enter="modify"
          v-on:blur="modify"
              v-bind:value="text"
          ref="input">
    <div v-else>
        <span v-on:dblclick="modifyOn=true"
          v-on:click.stop="displayCtxMenu" style="cursor:pointer;">
       {{text}}
      </span>
```

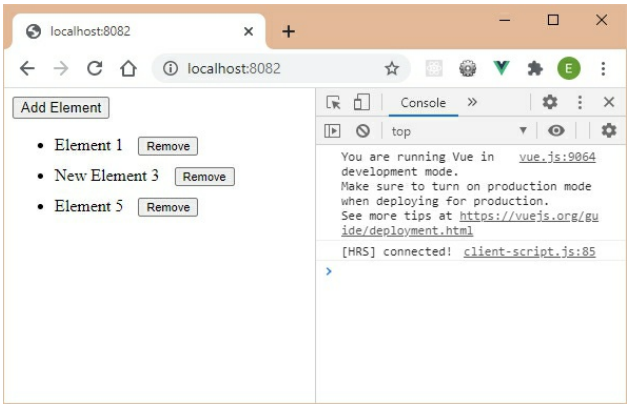
```
      <ContextMenu v-show="displayContextMenu" v-bind:x="x" v-
bind:y="y" />
      <button style="margin-left:10px; font-size:11px;"
            v-on:click="remove">
        Remove
      </button>
    </div>
  </li>
 `
});
Vue.component("ContextMenu", {
 props : [
  "x",
  "y"
 ],
 data() {
  return {
  }
 },
 methods : {
  modify() {
   this.$parent.$emit("modify");
  },
  remove() {
   this.$parent.$emit("remove");
  }
 },
 template : `
   <div v-bind:style="{position:'absolute', top:y + 'px', left:x + 'px',
                backgroundColor:'gainsboro'}">
    <ul style="margin-left:0px; padding:5px; list-style-type:none;
cursor:pointer;">
      <li v-on:click="modify">Modify</li>
      <li v-on:click="remove">Remove</li>
    </ul>
  </div>
```

```
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
eventBus.$on("modify", function(index, value) {
  vm.elements[index].text = value;
});
eventBus.$on("add", function() {
  var text = "Element " + (vm.elements.length + 1);
  vm.elements.push({text});
});
eventBus.$on("displaycontextmenu", function(contextmenu) {
  vm.contextmenu = contextmenu;
});
eventBus.$on("hidecontextmenu", function() {
   if (vm.contextmenu)
```

```
    vm.contextmenu.$parent.displayContextMenu = false;
});
```

The taking into account of the modify and remove events in the Element component is done in the created() method of the Element component.

We can see that these actions have modified the list of elements:



Style the menu by displaying in red the menu item on which the mouse is

A possible improvement consists in styling the contextual menu so as to display the chosen item in red. To do this, we create a new component named ItemMenu which will be used to display each menu item.

Use an ItemMenu component to style the context menu

```
Vue.component("App", {
  props : [
    "elements"
  ],
```

```
  methods : {
    add() {
      eventBus.$emit("add");
    },
    click() {
      eventBus.$emit("hidecontextmenu");
    }
  },
  template : `
    <div style="position:relative; height:100%;" v-on:click="click">
      <button v-on:click="add">Add Element</button>
      <Elements v-bind:elements="elements" />
    </div>
  `
});
Vue.component("Elements", {
  props : [
    "elements"
  ],
  data() {
    return {
    }
  },
  template : `
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
                 v-bind:key="index"
                 v-bind:index="index"
                 v-bind:text="element.text">
      </Element>
    </ul>
    <div v-else><br>Empty List</div>
  `
});
Vue.component("Element", {
```

```
props : [
  "text",
  "index"
],
data() {
  return {
    modifyOn : false,
    displayContextMenu : false,
      x : 0,
    y : 0
  }
},
created() {
  this.$on("modify", function() {
    this.modifyOn = true;
    this.displayContextMenu = false;
  });
  this.$on("remove", function() {
    this.remove();
    this.displayContextMenu = false;
  });
},
updated() {
  if (this.$refs.input) this.$refs.input.focus();
},
methods : {
  remove() {
    eventBus.$emit("remove", this.index);
  },
  modify(event) {
    this.modifyOn = false;
    var newText = event.target.value;
    eventBus.$emit("modify", this.index, newText);
  },
  displayCtxMenu(event) {
    this.x = event.clientX;
```

```
    this.y = event.clientY;
    this.displayContextMenu = true;
    this.$parent.$children.forEach((child) => {
        if(child.index != this.index)
       child.displayContextMenu = false;
    });
    eventBus.$emit("displaycontextmenu", this.$children[0]);
   }
 },
 template : `
   <li style="margin-top:10px;">
      <input v-if="modifyOn" type="text"
          v-on:keydown.enter="modify"
          v-on:blur="modify"
              v-bind:value="text"
          ref="input">
    <div v-else>
        <span v-on:dblclick="modifyOn=true"
          v-on:click.stop="displayCtxMenu" style="cursor:pointer;">
       {{text}}
      </span>
      <ContextMenu v-show="displayContextMenu" v-bind:x="x" v-
bind:y="y" />
      <button style="margin-left:10px; font-size:11px;"
            v-on:click="remove">
       Remove
      </button>
    </div>
   </li>
 `
});
Vue.component("ContextMenu", {
 props : [
   "x",
   "y"
 ],
```

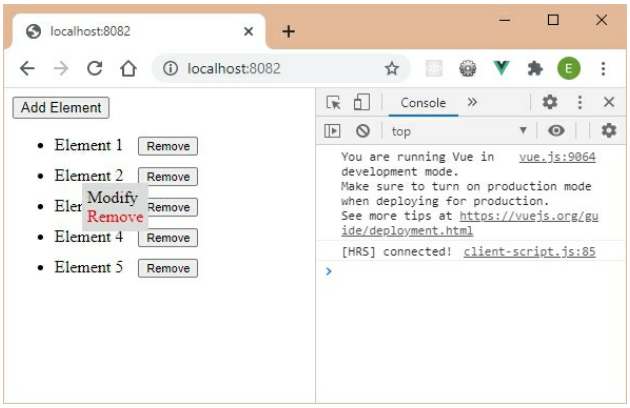
```
  data() {
    return {
    }
  },
  methods : {
    modify() {
      console.log("modify");
      this.$parent.$emit("modify");
    },
    remove() {
      this.$parent.$emit("remove");
    }
  },
  template : `
     <div v-bind:style="{position:'absolute', top:y + 'px', left:x + 'px',
                         backgroundColor:'gainsboro'}">
      <ul style="margin-left:0px; padding:5px; list-style-type:none;
cursor:pointer;">
        <ItemMenu v-on:click="modify">Modify</ItemMenu>
        <ItemMenu v-on:click="remove">Remove</ItemMenu>
      </ul>
    </div>
  `
});
Vue.component("ItemMenu", {
  data() {
    return {
      color:""
    }
  },
  methods : {
    mouseover() {
      this.color = "red";
    },
    mouseout() {
      this.color = "";
```

```
    }
  },
  template : `
     <li v-on:click="$emit('click')"
          v-on:mouseover="mouseover" v-on:mouseout="mouseout"
        v-bind:style="{color:color}">
      <slot/>
    </li>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
eventBus.$on("modify", function(index, value) {
  vm.elements[index].text = value;
});
eventBus.$on("add", function() {
```

```
  var text = "Element " + (vm.elements.length + 1);
  vm.elements.push({text});
});
eventBus.$on("displaycontextmenu", function(contextmenu) {
  vm.contextmenu = contextmenu;
});
eventBus.$on("hidecontextmenu", function() {
   if (vm.contextmenu)
    vm.contextmenu.$parent.displayContextMenu = false;
});
```

The ItemMenu component handles mouseover and mouseout events to display the style of items. But the click on an item must be transmitted to the ItemMenu component so that it processes the click on the item concerned (hence the directive v-on:click="$emit('click')" on the element < li>).

By displaying the contextual menu then by hovering over the Remove item with the mouse, this item turns red:



Dynamic creation of the context

menu

The previous context menu is displayed by means of templates, which is the traditional way of building a Vue application.

Another way (less traditional and less recommended) would be to write JavaScript code that creates the ContextMenu component dynamically. The advantage of doing this is that you only have one ContextMenu component that would be displayed at the desired location on the page, rather than having one ContextMenu component per list item.

> Even if this way of proceeding is less recommended, it is nevertheless interesting because it allows you to see how to use Vue components with JavaScript code.

We use here the same code of the ContextMenu component as the one written previously, namely:

ContextMenu component

```
var ContextMenu = Vue.component("ContextMenu", {
  props : [
    "x",
    "y"
  ],
  data() {
    return {
    }
  },
  methods : {
```

```
    modify() {
      console.log("Modify");
    },
    remove() {
      console.log("Remove");
    }
  },
  template : `
     <div v-bind:style="{position:'absolute', top:y + 'px', left:x + 'px',
                         backgroundColor:'gainsboro'}">
       <ul style="margin-left:0px; padding:5px; list-style-type:none;
                  cursor:pointer;">
       <li v-on:click="modify">Modify</li>
       <li v-on:click="remove">Remove</li>
     </ul>
    </div>
  `
});
```

Notice that we store the return of the Vue.component() method call in a ContextMenu variable. Indeed, the return of Vue.component() is a JavaScript function which is used to construct the objects associated with this component, hence the importance of giving a name to this class of objects in order to then be able to instantiate them by new.

The context menu is created in the displayCtxMenu() method called in the Element component when clicking on a list item. The menu variable represents an instance of the context menu and is managed by a global variable so that it is unique for the whole page.

Create the context menu when clicking on a list item

```
displayCtxMenu(event) {
  var x = event.clientX;
  var y = event.clientY;
  if (menu) {
    menu.$destroy();
    menu.$el.parentElement.removeChild(menu.$el);
  }
  menu = new ContextMenu({
     propsData : {
        x : x,
        y : y
    }
  }).$mount();
  this.$el.appendChild(menu.$el);
}
```

If the menu is already present (menu variable initialized), this menu is destroyed:

- We first destroy the Vue component associated with the menu (by the menu.$destroy() instruction),
- Then we remove the menu from the HTML page using removeChild().

Then the menu is then created (in all cases):

- We instantiate a ContextMenu object by passing it the corresponding props in the propsData property,
- Then we transform the created object into a Vue component using $mount(),

- Then we insert the corresponding DOM element (this.$el) into the DOM tree using appendChild(). This last statement would have been optional if we had specified the el property when creating the ContextMenu object, but it would have replaced all the contents of the DOM element associated with el (whereas appendChild() only inserts without replace).

The complete code for the page using the ContextMenu component is as follows:

Use a ContextMenu component dynamically

```
Vue.component("App", {
 props : [
  "elements"
 ],
 methods : {
  add() {
   eventBus.$emit("add");
  },
  click() {
   if (menu) {
    menu.$destroy();
    menu.$el.parentElement.removeChild(menu.$el);
    menu = null;
   }
  }
 },
 template : `
  <div style="position:relative; height:100%;" v-on:click="click">
   <button v-on:click="add">Add Element</button>
```

```
    <Elements v-bind:elements="elements" />
   </div>
  `
});
Vue.component("Elements", {
  props : [
   "elements"
  ],
  data() {
   return {
   }
  },
  template : `
   <ul v-if="elements.length">
      <Element v-for="(element, index) in elements" style="margin-
top:10px;"
            v-bind:key="index"
            v-bind:index="index"
            v-bind:text="element.text">
     </Element>
   </ul>
   <div v-else><br>Empty List</div>
  `
});
var menu;
Vue.component("Element", {
  props : [
   "text",
   "index"
  ],
  data() {
   return {
     modifyOn : false
   }
  },
  updated() {
```

```
  if (this.$refs.input) this.$refs.input.focus();
 },
methods : {
  remove() {
    eventBus.$emit("remove", this.index);
  },
  modify(event) {
    this.modifyOn = false;
    var newText = event.target.value;
    eventBus.$emit("modify", this.index, newText);
  },
  displayCtxMenu(event) {
    var x = event.clientX;
    var y = event.clientY;
    if (menu) {
      menu.$destroy();
      menu.$el.parentElement.removeChild(menu.$el);
    }
    menu = new ContextMenu({
        propsData : {
          x : x,
          y : y
      }
    }).$mount();
    this.$el.appendChild(menu.$el);
  }
},
template : `
  <li style="margin-top:10px;">
      <input v-if="modifyOn" type="text"
          v-on:keydown.enter="modify"
          v-on:blur="modify"
              v-bind:value="text"
          ref="input">
    <div v-else>
        <span v-on:dblclick="modifyOn=true"
```

```
          v-on:click.stop="displayCtxMenu" style="cursor:pointer;">
        {{text}}
      </span>
      <button style="margin-left:10px; font-size:11px;"
            v-on:click="remove">
        Remove
      </button>
    </div>
  </li>
 `
});
var ContextMenu = Vue.component("ContextMenu", {
 props : [
  "x",
  "y"
 ],
 data() {
  return {
  }
 },
 methods : {
  modify() {
   console.log("Modify");
  },
  remove() {
   console.log("Remove");
  }
 },
 template : `
   <div v-bind:style="{position:'absolute', top:y + 'px', left:x + 'px',
                backgroundColor:'gainsboro'}">
     <ul style="margin-left:0px; padding:5px; list-style-type:none;
           cursor:pointer;">
     <li v-on:click="modify">Modify</li>
     <li v-on:click="remove">Remove</li>
    </ul>
```

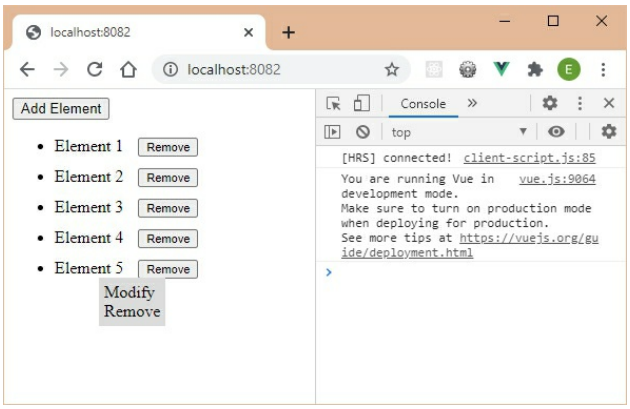
```
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  template : `
    <App v-bind:elements="elements" />
  `
});
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
eventBus.$on("modify", function(index, value) {
  vm.elements[index].text = value;
});
eventBus.$on("add", function() {
  var text = "Element " + (vm.elements.length + 1);
  vm.elements.push({text});
});
```

The Element component template no longer includes the ContextMenu component, as the latter is created

dynamically. Management of the Element component is therefore simpler.

Note the use of the modifier stop when clicking on a list item, so that the menu can be displayed (as seen previously).

Clicking on an item in the list displays the context menu, which then disappears when clicking outside the list.



4 – CREATE DIRECTIVES

Directives are special attributes positioned on HTML elements or components created by Vue. A directive is used to modify the default behavior of the HTML element or of the component on which it is used.

A directive always begins with v-, regardless of the directive. There are however two exceptions to this rule, for the v-on and v-bind directives which are among the most used:

- The v-on directive can be replaced by @, followed by the name of the event to handle,
- The v-bind directive can be replaced by :, followed by the name of the attribute associated with the one-way binding.

> The v-on and v-bind directives can be used in this form, but also in their shortened form @ and :.

Vue also allows you to create your own directives, which can be used in HTML elements or in components.

As seen in the previous chapter, we use the following

structure of the HTML page, in which we create the div#root element which will contain the HTML elements of the template associated with the Vue object.

Basic index.html file

```
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="root"></div>
</body>
<script>
var vm = new Vue({
  el : "#root",
  template : `
    <div>
    </div>
  `
});
</script>
</html>
```

Create a directive with Vue.directive()

A directive is created using the Vue.directive(name, callback) method:

- The name parameter is a string corresponding to the name of the directive (without indicating v-,

which will only be added when using the directive),

- The callback parameter is a callback function of the form callback(el, binding, vnode).

> This form of using the Vue.directive() method is the most common. We will see at the end of the chapter another form of use which makes it possible to manage the directive according to its life cycle.

The callback function callback(el, binding, vnode) has the three parameters el, binding and vnode.

The el parameter represents a DOM element, but its meaning is different depending on whether the directive is applied to an HTML element or to a Vue component:

- If the directive is applied to an HTML element, it represents this DOM element (which can have children in its descendants),
- If the directive is applied to a component, el represents the DOM element at the root of the component (the one indicated as the root element in the template).

The binding parameter is an object indicating specifics about the directive. Let us take as an example the directive with the v-dir name (which does not exist but

which we use as an example):

- binding.name represents the name of the directive (without the v- prefix), so here binding.name will be "dir",
- binding.rawName represents the full expression of the directive (with the v- prefix, up to the end of the expression or the = sign). If for example we write v-dir:arg1.stop.lazy, binding.rawName will be equal to this expression.
- binding.value represents the value specified in the directive (the value is optional), after the = sign, and having evaluated the expression. If for example we write v-dir="1+1", binding.value will be equal to 2.
- binding.expression also represents the value specified in the directive (the value is optional), after the = sign, but without evaluating the expression. If for example we write v-dir="1+1", binding.expression will be equal to "1+1" (while binding.value will be equal to 2).
- binding.arg represents an argument specified in the directive (the argument is optional), after the sign :. If for example we write v-dir:arg1, binding.arg will be equal to "arg1".
- binding.modifiers represents an array of objects

containing the modifiers used in the directive (modifiers are optional), after the sign .. If for example we write v-dir.stop.lazy, binding.modifiers will be worth an object {stop: true, lazy: true}.

We can therefore see that the binding parameter represents an object that Vue provides in order to help us perform processing according to the values indicated in the directive.

The vnode parameter is an object associated with the virtual DOM created by Vue. This parameter can be useful when the directive is used on a Vue component (instead of a simple HTML element).

To write our first directives, let's take two simple examples:

- The v-log directive, when present, displays information in the console,
- The v-hide directive, when present, hides the element to which it applies (unconditionally).

v-log directive to display information in the console

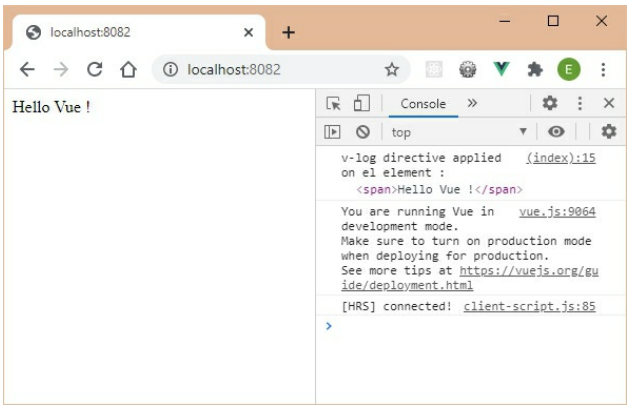
Let's use the v-log directive to display, in the console, information about the HTML element to which it is

applied.

v-log directive

```
Vue.directive("log", function(el) {
  console.log("v-log directive applied on el element : ", el);
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <span v-log>Hello Vue !</span>
    </div>
  `
});
```

The v-log directive is used here in a element. When this element is displayed in the page, the directive will be applied and the message in the console will be displayed.



The v-log directive does its job correctly. However, we can already see a limitation if we try for example to display in the console, information on the parent of the el element (therefore on the element which

contains the element).

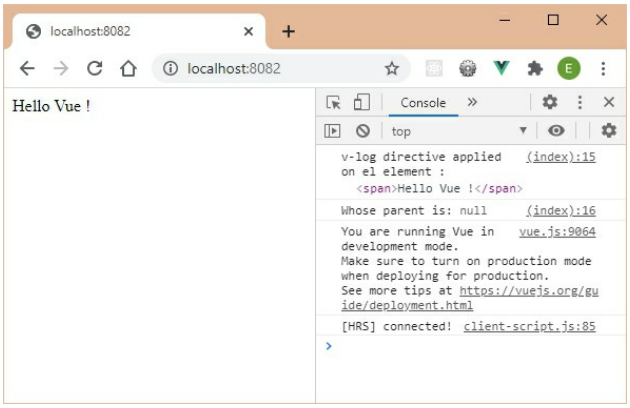
Let's modify the directive to display additional information about the element represented by el.parentElement.

View information about el.parentElement

```
Vue.directive("log", function(el) {
  console.log("v-log directive applied on el element : ", el);
  console.log("Whose parent is:", el.parentElement);
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <span v-log>Hello Vue !</span>
    </div>
  `
});
```

The only change is the addition of a new console.log() statement in the directive.

The display in the console is then the following:



When displayed in the directive, the parent of the

 element does not exist (it is set to null). In fact it exists (it is the element used in the template of the Vue object), but it is not yet accessible when the directive is used. Hence the importance of knowing the life cycle of a directive, which we will see at the end of this chapter.

v-hide directive to hide an element on the page

The previous v-log directive was used to display in the console information about the el element to which it applies. Let's write the v-hide directive which allows to hide this element in the page.

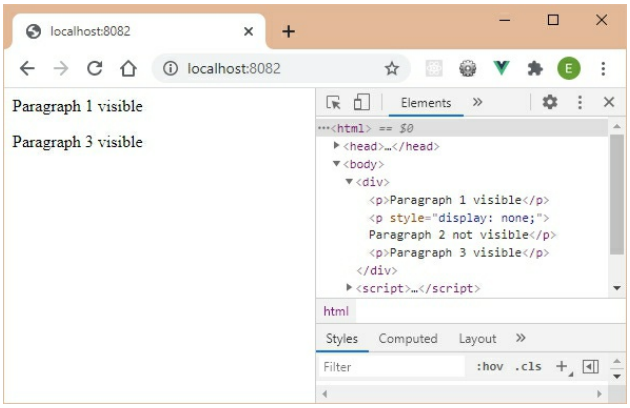
v-hide directive

```
Vue.directive("hide", function(el) {
  el.style.display = "none";
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <p>Paragraph 1 visible</p>
      <p v-hide>Paragraph 2 not visible</p>
      <p>Paragraph 3 visible</p>
    </div>
  `
});
```

The el element on which the v-hide directive is applied (here the second paragraph) is hidden by setting its

display property to "none" (in the directive).

Only the first and third paragraphs are visible on the page:



The second paragraph is well registered in the DOM, but with the CSS display property set to "none".

Use the value of a directive

The previously created v-log and v-hide directives are used without specifying a value. Thus the v-hide directive always hides the element on which it applies.

Let's modify the v-hide directive so that it behaves like the v-show directive defined in Vue. We indicate as the value of the directive a boolean value:

- If v-hide is true, the element is hidden,
- If v-hide is false, the item is displayed.

In this case, needing to know the value associated with the directive, we indicate the binding parameter in the

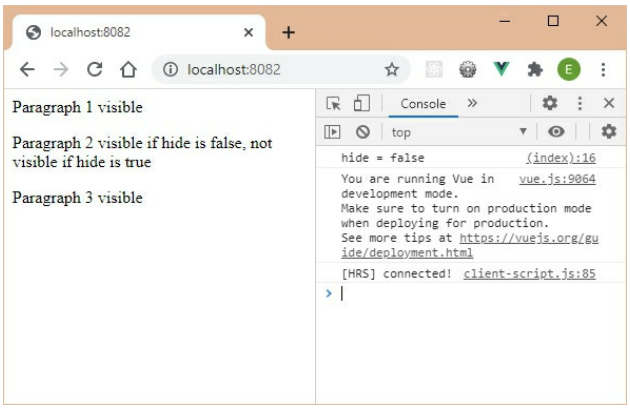
callback function.

v-hide directive with a boolean value

```
Vue.directive("hide", function(el, binding) {
  var hide = binding.value;
  console.log("hide = " + hide);
  if (hide) el.style.display = "none";
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <p>Paragraph 1 visible</p>
      <p v-hide="false">Paragraph 2 visible if hide is false,
                    not visible if hide is true</p>
      <p>Paragraph 3 visible</p>
    </div>
  `
});
```

The computed value of the directive is set in binding.value. If this value is true, the element is hidden otherwise it is displayed.

The second paragraph is therefore displayed because v-hide is set to false.



The value retrieved in binding.value is a calculated value, which means that it can be the result of a JavaScript expression, for example the value of a reactive variable.

Let's create the hide reactive variable in the data section of the Vue object, and assign this variable as the value of the v-hide directive.

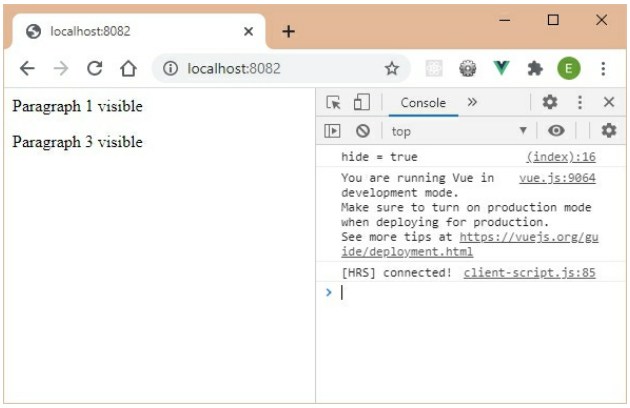
Assign the hide variable as the value of the v-hide directive

```
Vue.directive("hide", function(el, binding) {
  var hide = binding.value;
  console.log("hide = " + hide);
  if (hide) el.style.display = "none";
});
var vm = new Vue({
  el : "#root",
  data : {
    hide : true
  },
  template : `
    <div>
      <p>Paragraph 1 visible</p>
      <p v-hide="hide">Paragraph 2 visible if hide is false,
                       not visible if hide is true</p>
```

```
    <p>Paragraph 3 visible</p>
  </div>
`
});
```

The principle is the same as for the v-show directive. The value of the directive is sought among the reactive variables (here the variable hide), and the value of the variable (here true) is transmitted in the directive.

The second paragraph is hidden because the reactive variable hide is set to true.



Use a directive on a component

Previous directives were used directly on HTML elements, but can be used on Vue components.

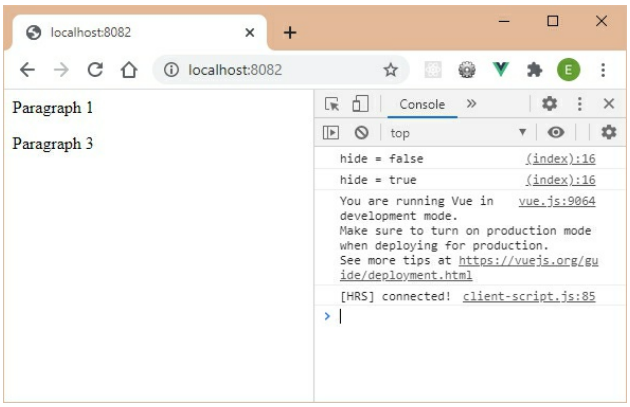
Let's create the App component that just displays a paragraph. The text of the paragraph is passed in the text props of the App component.

We use the v-hide directive on the three App components used here.

Use the v-hide directive on App components

```
Vue.directive("hide", function(el, binding) {
  var hide = binding.value;
  console.log("hide = " + hide);
  if (hide) el.style.display = "none";
});
Vue.component("App", {
  props : [
    "text"
  ],
  template : `
    <p>{{text}}</p>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" v-hide="false" />
      <App text="Paragraph 2" v-hide="true" />
      <App text="Paragraph 3" />
    </div>
  `
});
```

Only the second App component is hidden here. Since the v-hide directive is registered on two App components, it is therefore executed twice (hence the two messages displayed in the console).



Use the vnode parameter in the callback function

The vnode parameter is the third parameter that can be used in the callback function of the Vue.directive(name, callback) method.

It allows access to the virtual DOM created by Vue and therefore to the component to which the directive applies. Access to the component also allows access to reactive variables defined on this component.

Let's create a v-upper directive to indicate that we want to display a text in upper case. We use this directive on the previous App component, which allows to display in uppercase the text transmitted in the props.

v-upper directive

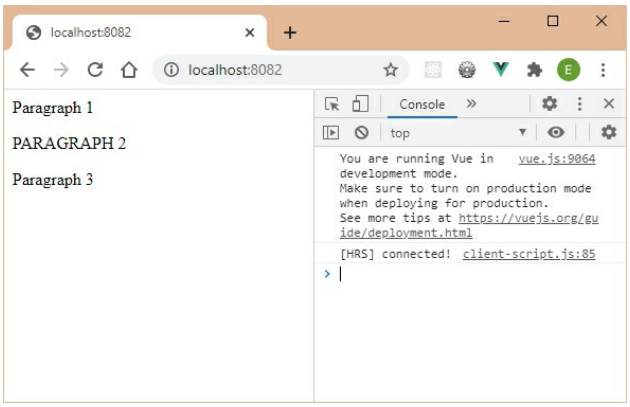
```
Vue.directive("upper", function(el, binding, vnode) {
  vnode.componentInstance.text2 =
vnode.componentInstance.text2.toUpperCase();
});
```

```
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <p>{{text2}}</p>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" v-upper />
      <App text="Paragraph 3" />
    </div>
  `
});
```

The vnode parameter allows access to vnode.componentInstance which is the corresponding App component instance. You can then modify the reactive variable text2 of the component by displaying the text in uppercase. Thanks to the responsiveness, the modified text is displayed again on the screen.

> Without the vnode parameter, it is not possible to access the App component, nor the text2 reactive variable associated with this component.

Let's check that the second paragraph is written in upper case:



Use an argument in the directive

The v-upper directive makes the specified text uppercase. You can add an argument to the directive that allows you to specify additional information. For example, also put the text in bold, in italics ...

Use the bold argument to make bold

Let's start with the possibility of making the text bold using the v-upper directive. We would therefore write in the template, using the bold argument:

Use the bold argument in the v-upper directive

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph upper + bold" v-upper:bold />
      <App text="Paragraph upper" v-upper />
```

```
    <App text="Paragraph normal" />
  </div>
 `
});
```

The "bold" string will be passed in the binding.arg parameter of the callback function. The writing of the v-upper directive and the App component become:

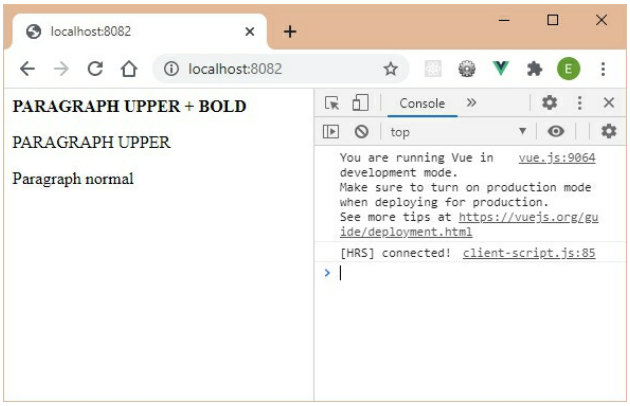
v-upper directive using the bold argument

```
Vue.directive("upper", function(el, binding, vnode) {
 var arg = binding.arg;
 if (arg == "bold") vnode.componentInstance.bold = true;
 vnode.componentInstance.text2 =
vnode.componentInstance.text2.toUpperCase();
});
Vue.component("App", {
 data() {
  return {
   text2 : this.text,
   bold : false
  }
 },
 props : [
  "text"
 ],
 template : `
  <p>
   <span v-if="bold">
    <b>{{text2}}</b>
   </span>
   <span v-else>
    {{text2}}
   </span>
  </p>
```

```
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph upper + bold" v-upper:bold />
      <App text="Paragraph upper" v-upper />
      <App text="Paragraph normal" />
    </div>
  `
});
```

The App component now has a new responsive variable named bold, which allows text to be bold if it is set to true.

The v-upper directive analyzes the presence of this argument in the directive, and if present, sets the corresponding bold reactive variable to true in the App component instance. This therefore makes it possible to display the text in upper case, but also in bold if the bold argument is present in the directive.



Use bold and italic arguments simultaneously

Vue allows you to indicate only one argument after the directive. But this argument being a character string, we can indicate several names separated for example by a comma (but not to use a point because it is the sign of the presence of a modifier, see next section).

We would therefore use a template of the following form, in order to indicate texts with the bold and / or italic arguments.

Use the bold and italic arguments in the v-upper directive

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph upper + bold" v-upper:bold />
      <App text="Paragraph upper + italic" v-upper:italic />
      <App text="Paragraph  upper + italic + bold" v-upper:italic,bold />
      <App text="Paragraph upper" v-upper />
      <App text="Paragraph normal" />
    </div>
  `
});
```

The paragraph using the bold and italic arguments therefore uses the v-upper directive in the form v-upper:italic,bold, in which the words bold and italic can be inverted (the order should not matter).

The v-upper directive and the App component therefore become:

v-upper directive using bold and italic arguments

```
Vue.directive("upper", function(el, binding, vnode) {
  var arg = binding.arg;
  if (arg) {
    var args = arg.split(",");
    var bold = args.includes("bold");
    var italic = args.includes("italic");
    vnode.componentInstance.bold = bold;
    vnode.componentInstance.italic = italic;
  }
  vnode.componentInstance.text2 =
vnode.componentInstance.text2.toUpperCase();
});
Vue.component("App", {
  data() {
    return {
      text2 : this.text,
      bold : false,
      italic : false
    }
  },
  props : [
    "text"
  ],
  template : `
    <p>
      <span v-if="bold">
        <span v-if="italic">
          <b><i>{{text2}}</i></b>
        </span>
        <span v-else>
          <b>{{text2}}</b>
```

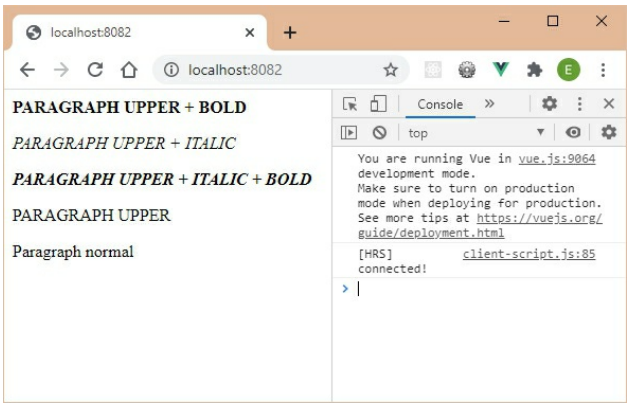
```
      </span>
    </span>
    <span v-else>
      <span v-if="italic">
        <i>{{text2}}</i>
      </span>
      <span v-else>
        {{text2}}
      </span>
    </span>
  </p>
 `
});
var vm = new Vue({
 el : "#root",
 template : `
   <div>
    <App text="Paragraph upper + bold" v-upper:bold />
    <App text="Paragraph upper + italic" v-upper:italic />
    <App text="Paragraph  upper + italic + bold" v-upper:italic,bold />
    <App text="Paragraph upper" v-upper />
    <App text="Paragraph normal" />
   </div>
 `
});
```

If the v-upper directive is used in the form v-upper:bold,italic, the binding.arg parameter is equal to "bold,italic". Just split this string into an array of elements that will contain ["bold", "italic"].

We then test the presence of each of the words in the array using the includes() method, and if the word is present, the reactive bold or italic variable is set to true

in the App component.

The App component then updates itself using the value of each of the bold and italic responsive variables.



Use modifiers in the directive

Another possibility in the directives is to use modifiers. Modifiers are separated from the argument or the name of the directive by prefixing it with a period. Several modifiers can follow each other, each one prefixed with a point.

The binding.modifiers parameter of the callback function allows you to retrieve an object whose keys are the modifiers present in the directive, and each of the values associated with each key is equal to true (meaning the presence of the modifier in the directive).

To obtain the list of modifiers present in the directive, it is therefore sufficient to retrieve the list of keys present in the binding.modifiers object, which can be

done by means of the Object.keys(binding.modifiers) instruction.

Obtain the list of keys of the modifiers present in the directive

```
var modifiers = Object.keys(binding.modifiers);
```

Let us use this principle in order to put in uppercase characters, only the letters indicated in the form of modifiers in the v-upper directive.

For example, to capitalize only the characters "a" and "p" in the indicated text, we would write in the template:

Using the modifiers a and e in the v-upper directive

```
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" v-upper.a.p />
      <App text="Paragraph 3" />
    </div>
  `
});
```

The directive is then written:

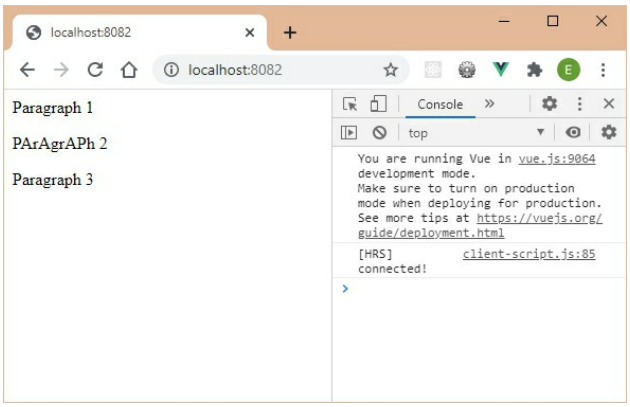
Capitalize the letters indicated in the directive's modifiers

```
Vue.directive("upper", function(el, binding, vnode) {
  var modifiers = Object.keys(binding.modifiers);
  var s= "";
  for (var i=0; i < vnode.componentInstance.text2.length; i++) {
    var c = vnode.componentInstance.text2.charAt(i);
```

```
    if (modifiers.includes(c)) s += c.toUpperCase();
    else s += c;
  }
  vnode.componentInstance.text2 = s;
});
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <p>{{text2}}</p>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" v-upper.a.p />
      <App text="Paragraph 3" />
    </div>
  `
});
```

For each character of the text in the component's text2 variable, we see if it is part of the list indicated in the modifiers array. If so, we copy this character in upper case in a new string. Then this string is copied into the text2 variable of the component.

The second paragraph contains the letters A and P in upper case, as indicated in the directive.



Life cycle in a directive

Let's take the example of the v-log directive, which displays in the console information about the element that uses the directive.

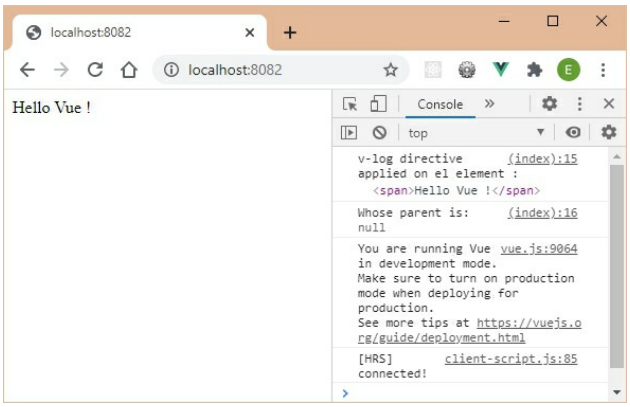
We wrote the following v-log directive, which also displays information about the parent of the element that uses the directive:

Display information about the element and its parent

```
Vue.directive("log", function(el) {
  console.log("v-log directive applied on el element : ", el);
  console.log("Whose parent is:", el.parentElement);
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <span v-log>Hello Vue !</span>
    </div>
```

```
  `
});
```

The display in the console was as follows:



The element is displayed but has no parent (null)! This is a life cycle management issue for the directive. By the time the directive's callback function is executed, the DOM element it applies to is not yet inserted into the DOM tree, causing this result.

To handle these cases, Vue provides callback functions associated with moments in the life cycle of the directive (as for components). These functions are as follows:

When creating the element or component that uses the directive:

- bind() method: called when creating the directive. The associated element is not yet attached to its parent.
- inserted() method: called when creating the directive, after the element has been attached to its

parent. Note that the parent is not necessarily inserted into the DOM yet.

When updating the item or component that uses the directive:

- update() method: called after each item update, but before its children are updated.
- componentUpdated() method: called after each item update, but after its children are updated.

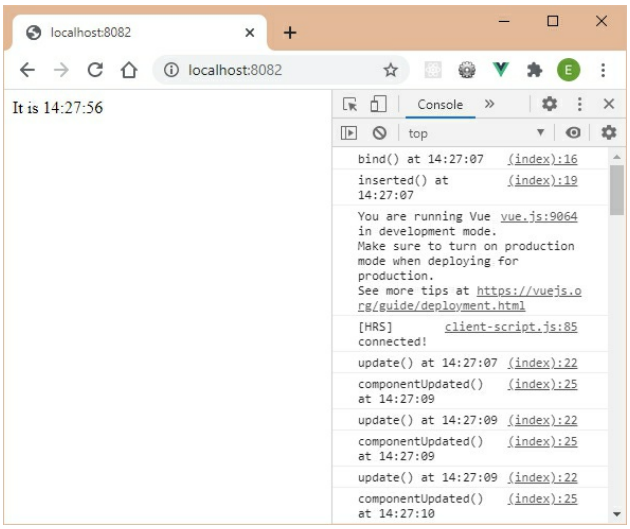
Let's use the v-log directive to display these methods in the console.

View the life cycle of the v-log directive

```
Vue.directive("log", {
  bind : function(el, binding) {
    console.log("bind() at " + el.innerHTML);
  },
  inserted : function(el, binding) {
    console.log("inserted() at " + el.innerHTML);
  },
  update : function(el, binding) {
    console.log("update() at " + el.innerHTML);
  },
  componentUpdated : function(el, binding) {
    console.log("componentUpdated() at " + el.innerHTML);
  }
});
var vm = new Vue({
  el : "#root",
  data : {
    time : new Date()
```

```
  },
  computed : {
    hhmmss() {
      var hh = this.time.getHours();
      var mm = this.time.getMinutes();
      var ss = this.time.getSeconds();
      if (hh < 10) hh = "0" + hh;
      if (mm < 10) mm = "0" + mm;
      if (ss < 10) ss = "0" + ss;
      return `${hh}:${mm}:${ss}`;
    }
  },
  created() {
    setInterval(() => {
      this.time = new Date();
    }, 1000);
  },
  template : `
    <div>
      It is <span v-log>{{hhmmss}}</span>
    </div>
  `
});
```

The display in the console shows the sequence of methods by Vue:



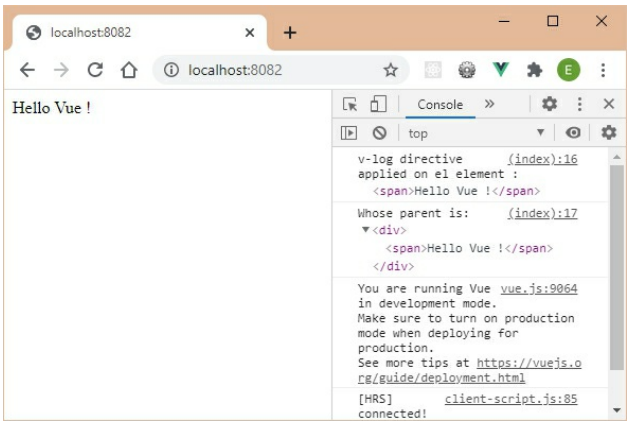
> When writing a directive using the Vue.directive(name, callback) form, the bind() method is actually executed by default. For more specific treatments, the life cycle method used must be indicated.

Now that we know how to use the lifecycle of a directive, let's modify the v-log directive to display information about the parent element. Just use the inserted() method which makes sure that the element has a parent.

Use the v-log directive to display information about the parent

```
Vue.directive("log", {
  inserted(el) {
    console.log("v-log directive applied on el element : ", el);
    console.log("Whose parent is:", el.parentElement);
  }
});
var vm = new Vue({
  el : "#root",
```

```
  template : `
    <div>
      <span v-log>Hello Vue !</span>
    </div>
  `
});
```



The parent of the element is now displayed in the console.

5 – CREATE FILTERS

A filter represents a function that will be executed in one of the following two cases:

- Either when displaying the value of an expression in a template (therefore in an expression surrounded by {{ and }}),
- Or when evaluating an expression in a v-bind directive.

The filter allows in these two cases to modify the final value of the expression to be used (either to display it in the template with {{ and }}, or to use it as a value in the one-way binding with v- bind).

As seen in the previous chapter, we use the following structure of the HTML page, in which we create the div#root element which will contain the HTML elements of the template associated with the Vue object.

Basic index.html file

```
<html>
<head>
```

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
</head>
<body>
<div id="root"></div>
</body>
<script>
var vm = new Vue({
  el : "#root",
  template : `
    <div>
    </div>
  `
});
</script>
</html>
```

Use a filter

A filter is used by means of the sign | which follows the expression to filter, the sign | itself being followed by the name of the filter.

Here are two examples of the use of filters:

- The first is used in a template surrounded by {{ and }},
- The second is used in a v-bind directive.

Each of the filters is called upper, and is used to capitalize the result.

upper filter used in a template

```
<p>{{text|upper}}</p>
```

The text to display, coming from the reactive variable text, will be capitalized before its display, thanks to the upper filter applied to the variable text.

upper filter used in a v-bind directive

```
<App v-bind:text="text|upper" />
```

The text transmitted in the text props, coming from the reactive variable text, will be capitalized before its transmission as props to the App component, thanks to the upper filter applied to the text variable.

> A filter therefore makes it possible to carry out processing on the value of the expression which precedes it.

Create a filter

The upper filter as used above causes a runtime error because it is not yet defined. It must therefore be created before using it.

We can create two kinds of filters:

- Global filters, usable throughout the HTML page. For this, we use the Vue.filter(name, callback) method.
- Local filters, usable only in the component or the Vue object in which they are defined. For this, we use the filter section of the component or the Vue object.

Let's take a look at these two ways to create a filter below.

Create a global filter with Vue.filter()

A filter created with the Vue.filter(name, callback) method will be usable throughout the HTML page because it is global to it.

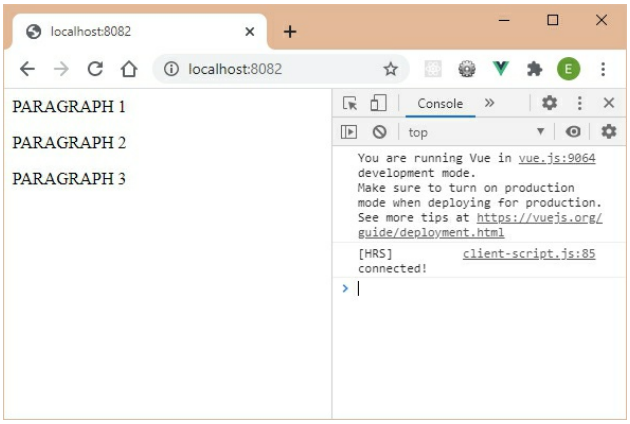
Let's create the upper filter and use it directly in the template (surrounded by {{ and }}).

Create and use the upper filter

```
Vue.filter("upper", function(value) {
  return value.toUpperCase();
});
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <p>{{text2|upper}}</p>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" />
```

```
    <App text="Paragraph 2" />
    <App text="Paragraph 3" />
  </div>
 `
});
```

The filter takes as its first parameter the value of the expression to be processed, here the value of the reactive variable text2, corresponding to the text props transmitted in the App component. Once the filter is applied, the final result is displayed.



Let's use the upper filter now using it in a v-bind directive.

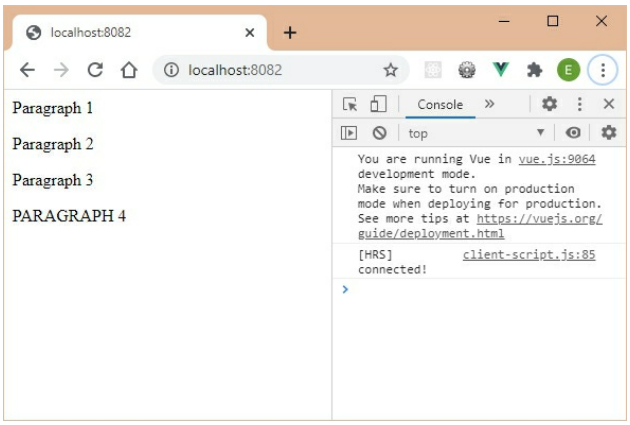
The filter is no longer systematically applied in the template on all the elements to be displayed, but only on the elements for which the filter is specified in the v-bind directive.

Use the upper filter in a v-bind directive

```
Vue.filter("upper", function(value) {
  return value.toUpperCase();
});
```

```
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <p>{{text2}}</p>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    text : "Paragraph 4"
  },
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" />
      <App text="Paragraph 3" />
      <App v-bind:text="text|upper" />
    </div>
  `
});
```

Only the fourth paragraph is now affected by the upper filter.



The value of the reactive variable text is capitalized here thanks to the filter, then this value is transmitted as text props in the App component which displays in the page what it receives.

Create a local filter by defining it in the filters section

The filter can be defined in the filters section of a component or in the Vue object. The difference with the definition of the filter by Vue.filter() is that the filter can only be used where it is defined.

Define the filter in a component

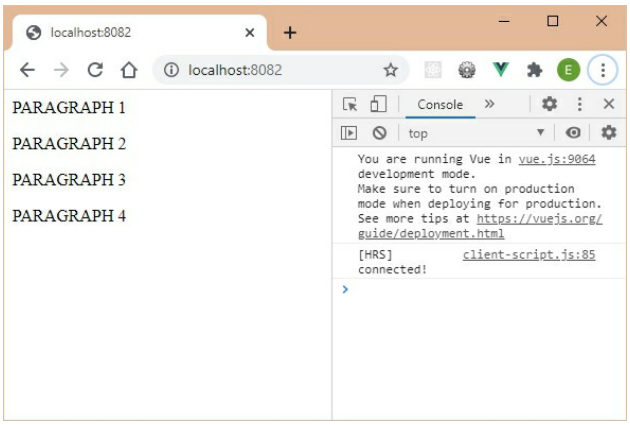
The filter defined in the App component can only be used in the App component.

Define the upper filter in the App component

```
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
```

```
  },
  props : [
    "text"
  ],
  filters : {
    upper(value) {
      return value.toUpperCase();
    }
  },
  template : `
    <p>{{text2|upper}}</p>
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    text : "Paragraph 4"
  },
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" />
      <App text="Paragraph 3" />
      <App v-bind:text="text" />
    </div>
  `
});
```

The upper filter is directly used in the template of the App component.



Using the filter in the Vue object would cause an error because the filter is unknown outside of the App component. So writing <App v-bind:text="text|upper" /> in the template of the Vue object would cause an error.

Define the filter in the Vue object

If the filter is now defined in the Vue object, it becomes usable in this object, but no longer in the App component.

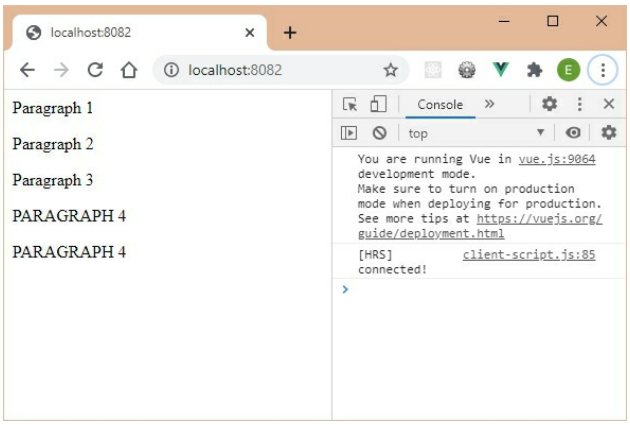
Define the upper filter in the Vue object

```
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <p>{{text2}}</p>
```

```
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    text : "Paragraph 4"
  },
  filters : {
    upper(value) {
      return value.toUpperCase();
    }
  },
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" />
      <App text="Paragraph 3" />
      <App v-bind:text="text|upper" />
      {{text|upper}}
    </div>
  `
});
```

The upper filter is used in two places in the Vue object:

- In the v-bind directive, by modifying the text of the reactive variable text which is then passed in the text props of the App component,
- Directly in the template using {{text|upper}}. The reactive variable text is transformed into uppercase and then displayed in the page.



Use a global filter and a local filter

A filter can be defined (with the same name) globally using Vue.filter() and locally using the filters section.

Let's define the upper filter globally and locally in the App component. We add a text in front of the version of the filter used in order to see which filter is used in each place where it is written.

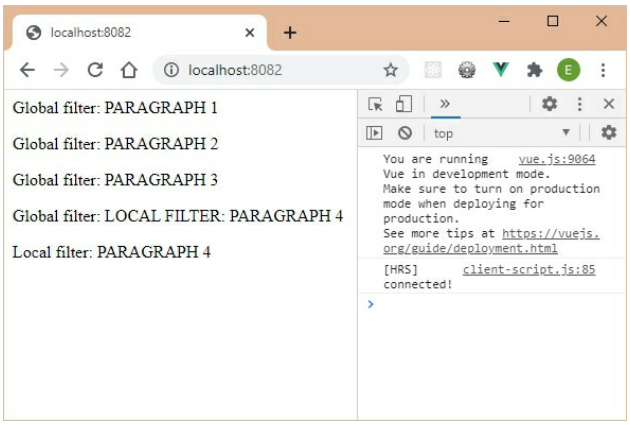
Define the upper filter globally and locally

```
Vue.filter("upper", function(value) {
  return "Global filter" + value.toUpperCase();
});
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <p>{{text2|upper}}</p>
```

```
  `
});
var vm = new Vue({
  el : "#root",
  data : {
    text : "Paragraph 4"
  },
  filters : {
    upper(value) {
      return "Local filter" + value.toUpperCase();
    }
  },
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" />
      <App text="Paragraph 3" />
      <App v-bind:text="text|upper" />
      {{text|upper}}
    </div>
  `
});
```

The local filter adds the text "Local filter:" in front of the returned text, while the global filter adds "Global filter:" in front of it.

The result is the following:



Let's analyze the displayed results:

The first three paragraphs use the global filter because no other filter is available when using them.

On the other hand, the fourth paragraph uses the two filters, first local then global:

- The use of the local filter is due to the writing of the v-bind:text="text|upper" directive,
- Whereas the use of the global filter is due to the passing of the text to the App component which displays it by means of {{text2|upper}} in its template.
- Finally the last paragraph displayed in the page only uses the local filter because nothing is then transmitted to the App component.

Create a filter that returns HTML code

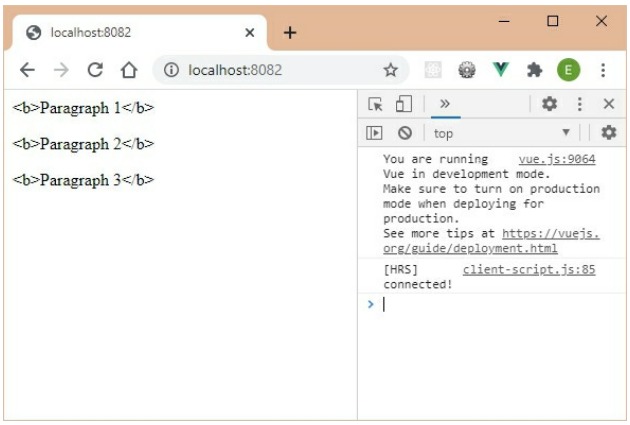
We want to create the bold filter which makes it possible to put the indicated text in bold. To do this, it adds the <b> tag around the text as a parameter

returned by the filter.

bold filter to make the text bold

```
Vue.filter("bold", function(value) {
  return "<b>" + value + "</b>";
});
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <p>{{text2|bold}}</p>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" />
      <App text="Paragraph 3" />
    </div>
  `
});
```

The result is the following:



Tags added by the filter are not interpreted as tags but are displayed as text. Vue should be told that the display should be interpreted as HTML and not as text.

Use the v-html directive to display HTML

Let's use the v-html directive (defined in Vue as standard) which allows you to indicate HTML content for an element, which will be interpreted in HTML and not in plain text.

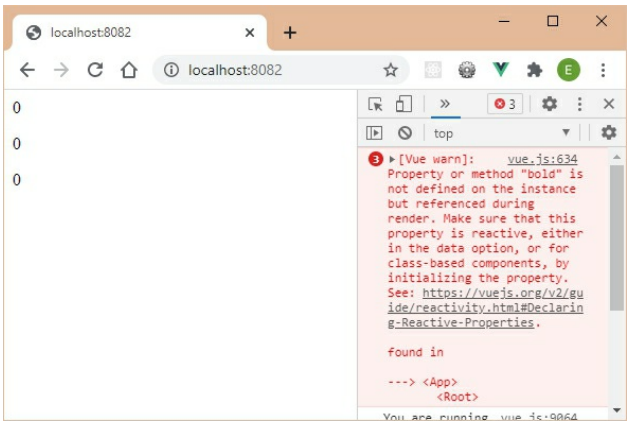
Use the v-html directive to display paragraph text

```
Vue.filter("bold", function(value) {
  return "<b>" + value + "</b>";
});
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
```

```
  template : `
    <p v-html="text2|bold"></p>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" />
      <App text="Paragraph 3" />
    </div>
  `
});
```

We are attempting to use the bold filter in a v-html directive, even though it was previously said that this only works on v-bind directives.

The result shows that it does not work:



Use this.$options.filters object in component

In fact, you should know that each filter definition creates a method in the this.$options.filters object.

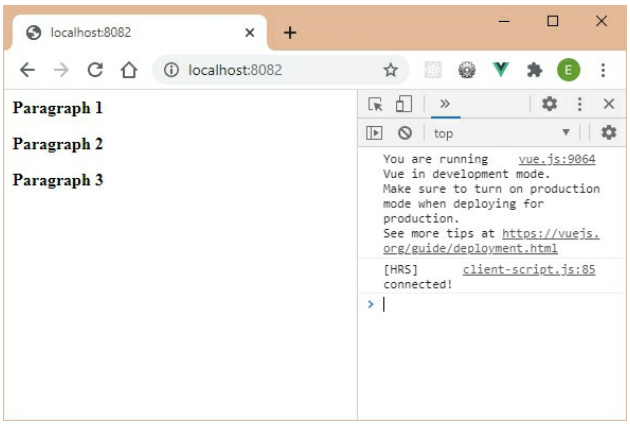
Here we will therefore have access to the bold(value) method by using it in the form this.$options.filters.bold(text2).

Let's change the use of the filter as follows:

Use the bold() method instead of the bold filter

```
Vue.filter("bold", function(value) {
  return "<b>" + value + "</b>";
});
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <p v-html="this.$options.filters.bold(text2)"></p>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" />
      <App text="Paragraph 3" />
    </div>
  `
});
```

The result is now the expected one:



Use a method instead of a filter

But in this case, it's easier to use a method that returns the result of the filter, rather than the filter itself.

Use the bold() method instead of the filter

```
Vue.component("App", {
  data() {
    return {
      text2 : this.text
    }
  },
  methods : {
    bold(value) {
      return "<b>" + value + "</b>";
    }
  },
  props : [
    "text"
  ],
  template : `
    <p v-html="bold(text2)"></p>
  `
});
var vm = new Vue({
  el : "#root",
```

```
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" />
      <App text="Paragraph 3" />
    </div>
  `
});
```

We always use the v-html directive so that the result is interpreted in HTML, but we directly call the bold() method defined in the methods section.

Use several filters in a row

A filter being ultimately a function call, it is possible to chain the call of several filters. Each filter uses the result obtained by calling the filters which preceded it.

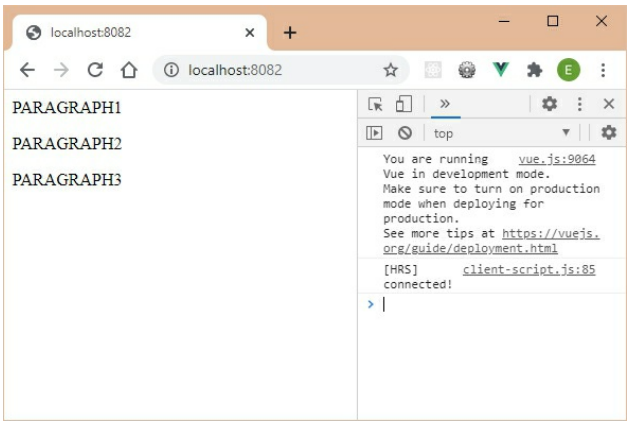
Let's create the two filters upper (already seen previously) and nospaces (which allows you to remove spaces in a string). Then use these two filters by chaining their calls.

upper and nospaces filters

```
Vue.filter("upper", function(value) {
  return value.toUpperCase();
});
Vue.filter("nospaces", function(value) {
  return value.replace(/ /g, "");
});
Vue.component("App", {
  data() {
```

```
    return {
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <p>{{text2|upper|nospaces}}</p>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" />
      <App text="Paragraph 2" />
      <App text="Paragraph 3" />
    </div>
  `
});
```

The transmitted text is capitalized then the spaces between the words are deleted.



Use filters with parameters

The preceding filters have no parameters. They can all be used by simply naming the filter when using it.

But a filter can also have parameters. When using it, all you have to do is indicate the list of filter arguments in parentheses (as for a function call). These arguments will be received by the processing method of the filter after the value parameter.

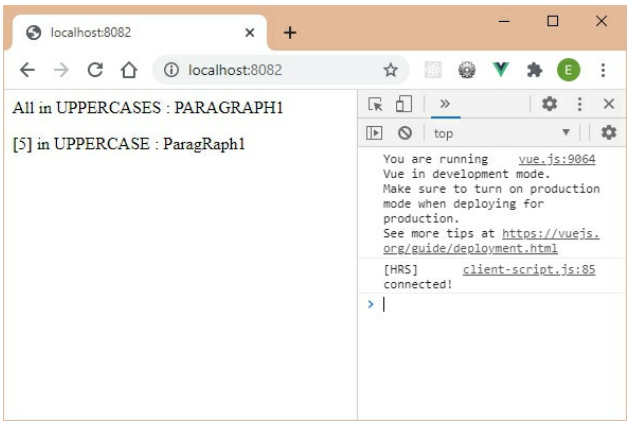
Suppose the upper filter can also be used in the form upper(n), where n indicates the index of the letter to be capitalized. You also need to be able to use it in its old upper form, meaning to capitalize everything.

upper and upper(n) filters

```
Vue.filter("upper", function(value, index) {
  var result = "";
  if (index == undefined) return value.toUpperCase();
  for (i = 0; i < value.length; i++) {
    var letter = value.charAt(i);
    if (i == index) letter = letter.toUpperCase();
    result += letter;
  }
  return result;
});
Vue.filter("nospaces", function(value) {
  return value.replace(/ /g, "");
});
Vue.component("App", {
  data() {
    return {
```

```
      text2 : this.text
    }
  },
  props : [
    "text"
  ],
  template : `
    <div>
      <p>All in UPPERCASES : {{text2|upper|nospaces}}</p>
      <p>[5] in UPPERCASE : {{text2|upper(5)|nospaces}}</p>
    </div>
  `
});
var vm = new Vue({
  el : "#root",
  template : `
    <div>
      <App text="Paragraph 1" />
    </div>
  `
});
```

If the index is not transmitted in the filter, it is therefore undefined, and in this case we return all the text in upper case. Otherwise we capitalize the letter corresponding to the index only.


localhost:8082
localhost:8082
All in UPPERCASES : PARAGRAPH1
[5] in UPPERCASE : ParagRaph1
top
You are running vue.js:9064
Vue in development mode.
Make sure to turn on production mode when deploying for production.
See more tips at https://vuejs.org/guide/deployment.html
[HRS] client-script.js:85
connected!

6 – USING THE VUE CLI UTILITY

The Vue CLI utility (CLI stands for Command Line Interface) allows you to easily create a project tree corresponding to a Vue application.

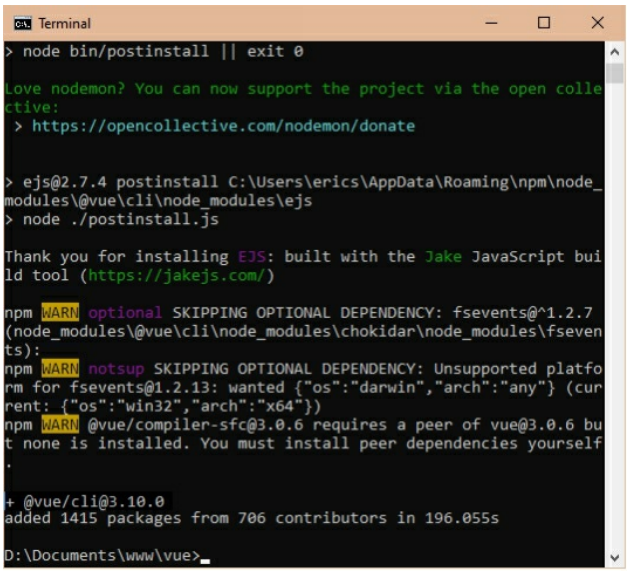
Install Vue CLI

Node.js must have been previously installed. Then type the command npm install -g @vue/cli@3.10.0 from a command shell (here the 3.10.0 version).

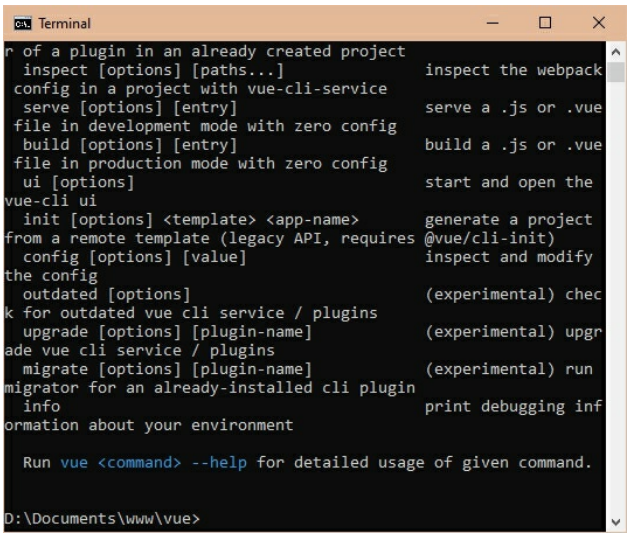
Install Vue CLI from a command interpreter

```
npm install -g @vue/cli@3.10.0
```

Once this command is executed, you get:



The Vue CLI utility is now installed. For a list of available commands, type the vue -h command in the shell.

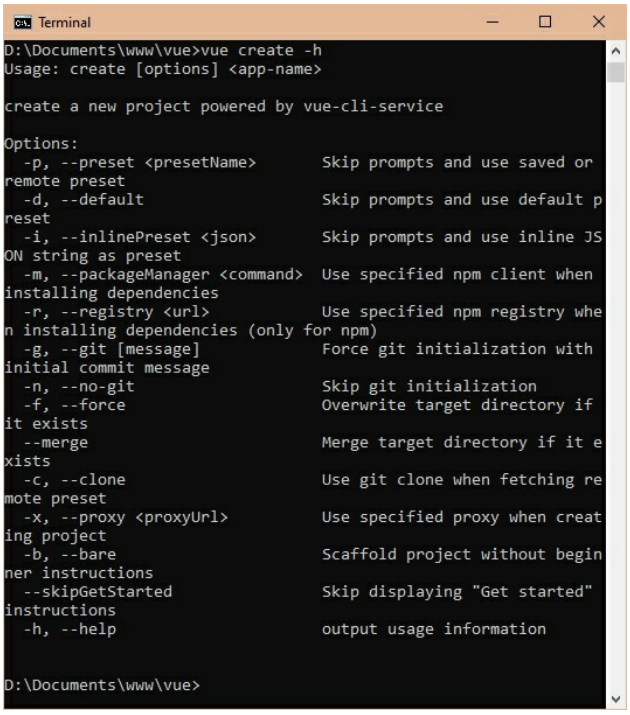


The Vue CLI utility is a command line interface. The create command is used to create a Vue application.

For information on the create command, type (as recommended in the previous screen) the vue create –h command.

Display help on the Vue CLI create command

```
vue create –h
```



We will now be able to create our Vue application from Vue CLI.

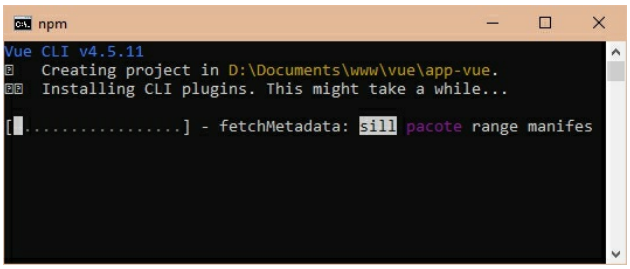
Create a Vue application with Vue CLI

Let's use the create command to create the Vue application tree. The application will be created here in the app-vue directory.

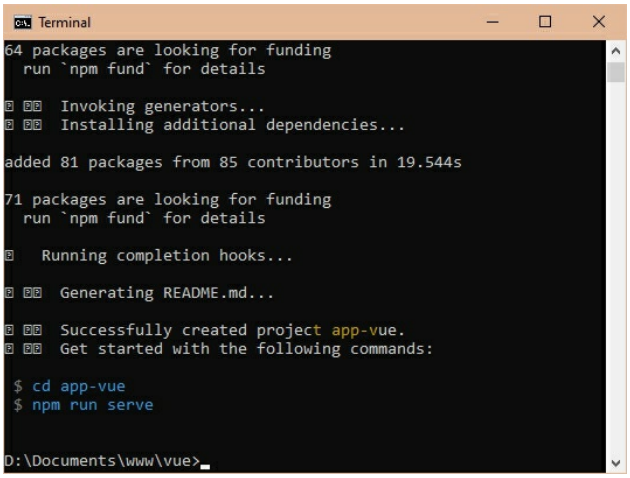
Create the Vue app tree in an app-vue directory

```
vue create app-vue
```

The command is launched:



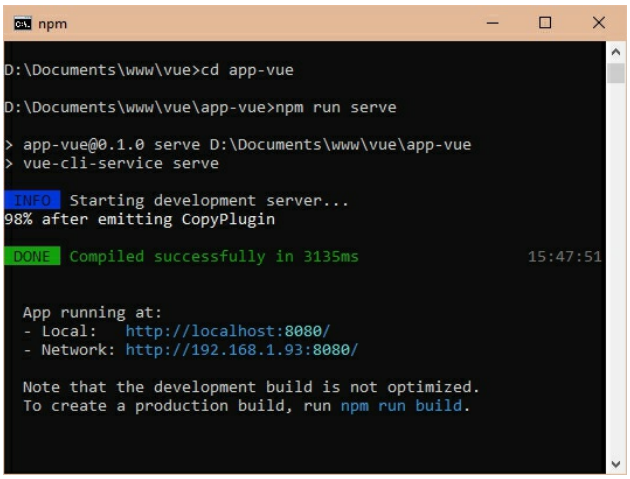
Then after a few minutes:



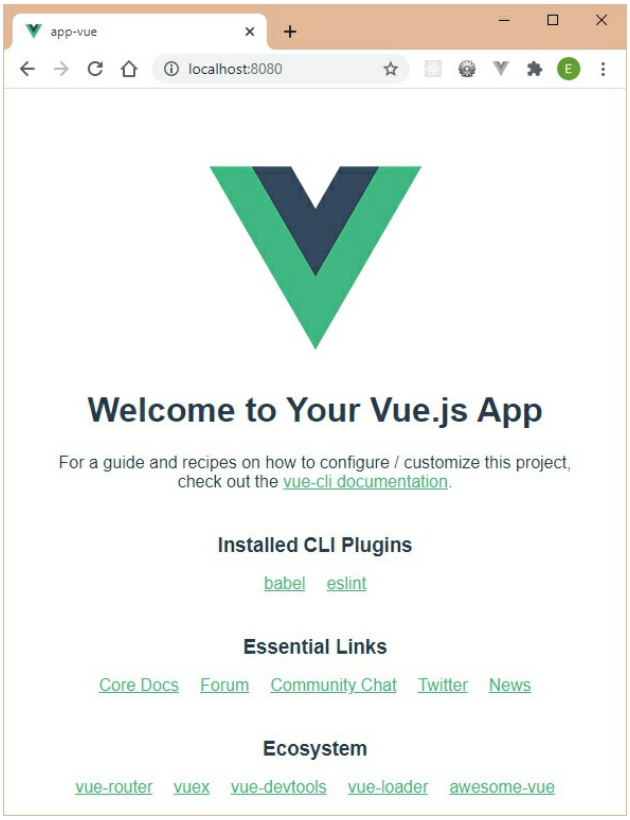
Let's run the commands displayed in the command interpreter, allowing to launch the server associated with the application created in the app-vue directory.

Start the Vue application server in the app-vue directory

```
cd app-vue
npm run serve
```



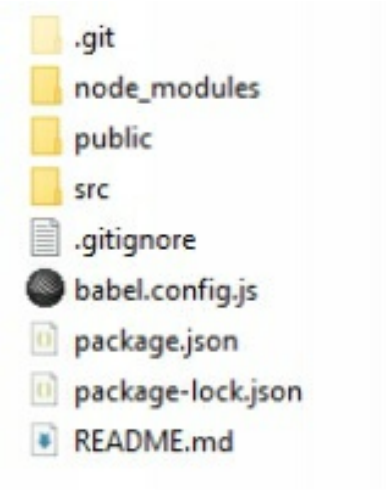
Let's start the server at the given URL http://localhost:8080:



The Vue application created by default is displayed in the browser.

Exploring the project tree

The app-vue directory contains the following files and directories:



Configuration files

The files located directly in the app-vue directory of the application are mainly configuration files or help files.

For example, the README.md file explains the npm commands that can be used to deploy the application on a server (including the npm run serve command launched previously):

README.md file

```
# app-vue

## Project setup
```
npm install
```
```

```
```

Compiles and hot-reloads for development
```
npm run serve
```

Compiles and minifies for production
```
npm run build
```

Run your tests
```
npm run test
```

Lints and fixes files
```
npm run lint
```

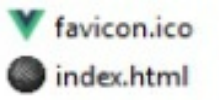
Customize configuration
See [Configuration Reference](https://cli.vuejs.org/config/).

public directory

The public directory contains the two files which will be directly accessible by the browser when accessing the site:

- favicon.ico file: this file corresponds to the application icon used to display it in the tab associated with the browser window that displays the site (here the Vue logo).
- index.html file: this is the main HTML file that

is displayed when accessing the site.



The contents of the index.html file are as follows (content may vary depending on Vue CLI version used):

Index.html file

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-
scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title>app-vue</title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but app-vue doesn't work properly without
JavaScript
            enabled. Please enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

We find in it the having the app id in which the Vue application will be registered. It corresponds to the element el having the root id in the code of the

previous chapters.

Note the comment following this element: the files created will be auto-injected. This is why there is no trace of inclusion of other files (CSS, JS, etc.), even if these files are present when the application is run.

node_modules directory

The node_modules directory is updated when installing packages by the npm install package-name command. It was created when running the vue create app-vue command.

src directory

The src directory is the one that contains the source files of our application. It is therefore the one that will be the most modified and enriched so that the application coincides with our wishes.

The src directory contains the App.vue and main.js files, as well as the two directories assets and components.

main.js file

The main.js file is as follows:

src/main.js file

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

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

We immediately see the contribution of the CLI. It creates a set of files and directories organized in the form of modules which will be imported into our files by means of the import instruction of ES6.

The main.js file creates the Vue object associated with the application and mounts it on the #app element that was created in the index.html file. The App component is that of the application as a whole. Its definition can be found in the App.vue file located in this same directory.

We see that the main.js file is a generic file and has no reason to be modified afterwards.

App.vue file

The App.vue file is as follows:

src/App.vue file

```
<template>
  <div id="app">
    <img alt="Vue logo" src="./assets/logo.png">
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>
```

```
<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
</script>

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

The App.vue file is the file that describes the App component of the Vue application. All components created by CLI (and later by us) are written to .vue extension files.

src/assets directory

The src/assets directory contains image files, CSS files, or JS files containing specific code. These files are shared by all components of the Vue application.

src/components directory

The App.vue file uses a HelloWorld component, taken

from the src/components/HelloWorld.vue file. All Vue components created by us are integrated into the src/components directory and have a .vue extension. This HelloWorld component is described as follows:

src/components/HelloWorld.vue file

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <p>
      For a guide and recipes on how to configure / customize this project,
<br>
      check out the
      <a href="https://cli.vuejs.org" target="_blank" rel="noopener">vue-cli
documentation</a>.
    </p>
    <h3>Installed CLI Plugins</h3>
    <ul>
      <li><a href="https://github.com/vuejs/vue-
cli/tree/dev/packages/%40vue/cli-plugin-babel" target="_blank"
rel="noopener">babel</a></li>
      <li><a href="https://github.com/vuejs/vue-
cli/tree/dev/packages/%40vue/cli-plugin-eslint" target="_blank"
rel="noopener">eslint</a></li>
    </ul>
    <h3>Essential Links</h3>
    <ul>
      <li><a href="https://vuejs.org" target="_blank" rel="noopener">Core
Docs</a></li>
      <li><a href="https://forum.vuejs.org" target="_blank"
rel="noopener">Forum</a></li>
      <li><a href="https://chat.vuejs.org" target="_blank"
rel="noopener">Community Chat</a></li>
      <li><a href="https://twitter.com/vuejs" target="_blank"
```

```
rel="noopener">Twitter</a></li>
      <li><a href="https://news.vuejs.org" target="_blank"
rel="noopener">News</a></li>
    </ul>
    <h3>Ecosystem</h3>
    <ul>
      <li><a href="https://router.vuejs.org" target="_blank"
rel="noopener">vue-router</a></li>
      <li><a href="https://vuex.vuejs.org" target="_blank"
rel="noopener">vuex</a></li>
      <li><a href="https://github.com/vuejs/vue-devtools#vue-devtools"
target="_blank" rel="noopener">vue-devtools</a></li>
      <li><a href="https://vue-loader.vuejs.org" target="_blank"
rel="noopener">vue-loader</a></li>
      <li><a href="https://github.com/vuejs/awesome-vue" target="_blank"
rel="noopener">awesome-vue</a></li>
    </ul>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
h3 {
  margin: 40px 0 0;
}
ul {
  list-style-type: none;
```

```
  padding: 0;
}
li {
  display: inline-block;
  margin: 0 10px;
}
a {
  color: #42b983;
}
</style>
```

Thanks to these two components App and HelloWorld, we can deduce the general structure of a Vue component managed by the Vue CLI architecture.

Structure of a Vue component in the tree structure generated by Vue CLI

The example files found in the src/components directory (here HelloWorld.vue) and the global application file App.vue located in the src directory are used to indicate the structure of Vue components which should be written later:

Each Vue component will be written to a .vue extension file, located in the src/components directory, (except for the main App component which is located in the src directory). There will be only one component per .vue file.

A .vue file will be structured with a <template> section, a <script> section and a <style> section. For styles to be component local, all you have to do is indicate the scoped attribute in the <style> element (as mentioned in the HelloWorld component code).

Component <template> section

The <template> section of the .vue file is used to indicate the HTML code to display for this component, in the same way as that previously used in the template section of a component.

The code in this section can use the components, directives, filters, methods, and reactive variables defined in the <script> section of the component.

Component <style> section

The <style> section of the .vue file allows you to define styles that will be local to the component (using the scoped attribute in the <style> element). These styles are defined using CSS selectors, much like writing a CSS styles file.

Component <script> section

The <script> section corresponds to the component's writing in JavaScript. It is similar to writing components that we did before, with some differences

due to writing as modules and the fact that the project will then need to be deployed to a production server.

The main differences are as follows:

The component is no longer declared by the Vue.component() method but is described as an object exported using the export default { … } statement.

- The template section is no longer registered in the component but in a <template> section of the .vue file.
- If a component uses other components to work, it must import (by the import instruction) the corresponding .vue files, and declare these components in a components section of the exported object.

Inserting components into the project

Let us give examples of the use of components in the form of modules, using the components already written in the previous chapters.

All these components will be used here by being integrated into the main App component of the application.

Timer component

Consider the Timer component already created in chapter 2 (*"Creating components"*). We write this component in the form explained above by creating a Timer.vue file.

src/components/Timer.vue file

```
<template>
  <div>
   <div v-if="sec||min"> Remaining time {{time}} </div>
   <div v-else>End of timer</div>
  </div>
</template>
<script>
export default {
  data() {
    return {
        min : this.minutes,
      sec : this.seconds
    }
  },
  props : [
     "minutes",
    "seconds"
  ],
  computed : {
    time() {
      var min = this.min;
      var sec = this.sec;
      if (min < 10) min = "0" + min;
      if (sec < 10) sec = "0" + sec;
      return min + ":" + sec;
    }
  },
  created() {
```

```
    var timer = setInterval(() => {
      this.sec -= 1;
      if (this.sec < 0) {
        this.min -= 1;
        if (this.min < 0) {
          this.min = 0;
          this.sec = 0;
          clearInterval(timer);
        }
        else this.sec = 59;
      }
    }, 1000);
  }
}
</script>
<style>
</style>
```

There are three sections <template>, <script> and <style> of a component. The <style> section is empty here because no style is used in the component.

Then we use this Timer component in the main App component of the application:

src/App.vue file

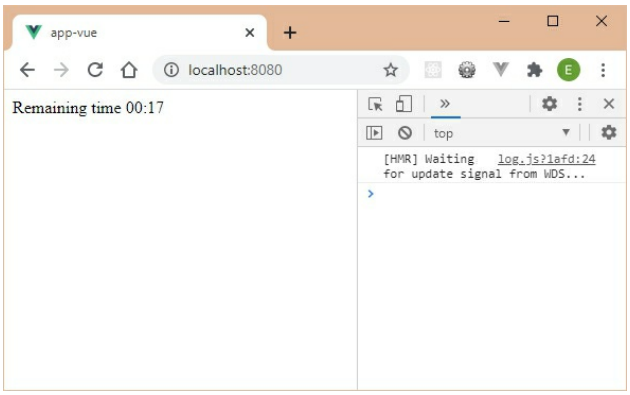
```
<template>
  <div>
    <Timer minutes="0" seconds="20"/>
  </div>
</template>

<script>

import Timer from "@/components/Timer.vue";
```

```
export default {
  components: {
    Timer
  }
}
</script>

<style>

</style>
```

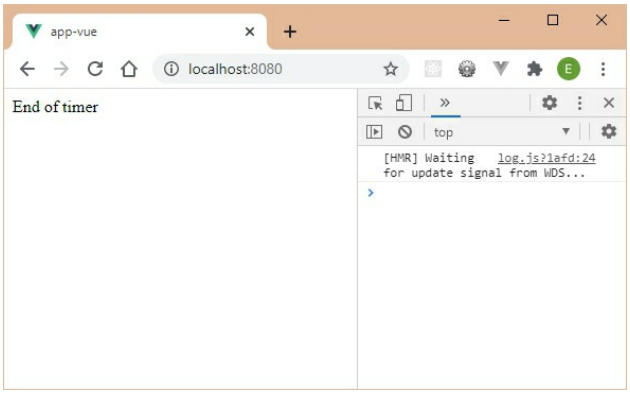
The App component imports the Timer component using JavaScript's import statement. Notice how the name of the Timer.vue file is shown. We indicate as directory a name starting with @. This is a convention to directly name the src directory from anywhere you are located, without worrying about using relative paths with "." or "..".

The Timer component is also mentioned in the components section of the App component otherwise it would be inaccessible in the App component.

It is checked that the operation is identical to the usual timer. The timer is triggered as soon as the page is displayed:



Then the timer stops and displays the message "End of timer":



Timer, StartStop, TimerStartStop and TimerStartCount components

Now let's integrate several components while respecting the same rules. We take the components already written previously by writing them according to the new conventions.

src/components/Timer.vue file

```
<template>
  <div>
   <b>{{timername}}</b> :
   <span v-if="sec||min"> {{time}} </span>
```

```
    <span v-else>End of timer</span>
    <slot v-bind:timername="timername"></slot>
  </div>
</template>
<script>
export default {
  data() {
    return {
        min : this.minutes,
      sec : this.seconds
    }
  },
  props : [
    "timername",
     "minutes",
    "seconds"
  ],
  computed : {
    time() {
      var min = this.min;
      var sec = this.sec;
      if (min < 10) min = "0" + min;
      if (sec < 10) sec = "0" + sec;
      return min + ":" + sec;
    }
  },
  mounted() {
    var timer;
    this.$on("startstop", function(stop) {
      if (stop) {
        if (timer) clearInterval(timer);
      }
      else {
        timer = setInterval(() => {
          this.sec -= 1;
          if (this.sec < 0) {
```

```
          this.min -= 1;
          if (this.min < 0) {
            this.min = this.minutes;
            this.sec = this.seconds;
            if (this.$children.length) this.$children[0].start_stop();
            else this.$emit("startstop", 0);
            clearInterval(timer);
          }
          else this.sec = 59;
        }
      }, 1000);
    }
    console.log(this.timername + " " + (stop ? "stopped" : "in progress"));
  });
  if (!this.$children.length) this.$emit("startstop", 0);
 }
}
</script>
<style>
</style>
```

src/components/StartStop.vue file

```
<template>
  <div>
   <button v-if="stopped" v-on:click="start_stop">
      Start {{timername}}
   </button>
   <button v-else v-on:click="start_stop">
      Stop {{timername}}
   </button>
 </div>
</template>
<script>
export default {
 data() {
```

```
    return {
      stopped : parseInt(this.stop)
    }
  },
  props : [
     "timername",
    "stop"
  ],
  created() {
    if (!this.stopped) {
      this.stopped = !this.stopped;
      this.start_stop();
    }
  },
  methods : {
    start_stop() {
      this.$nextTick(function() {
        this.stopped = !this.stopped;
        this.$parent.$emit("startstop", this.stopped);
       });
    }
  },
  components: {
  }
}
</script>
<style>
</style>
```

src/components/TimerStartStop.vue file

```
<template>
  <div>
     <Timer v-bind:timername="timername"
            v-bind:minutes="minutes"
          v-bind:seconds="seconds">
```

```
      <StartStop v-bind:timername="timername" v-bind:stop="1" />
    </Timer>
  </div>
</template>
<script>
import Timer from "@/components/Timer.vue";
import StartStop from "@/components/StartStop.vue";
export default {
  props : [
    "timername",
     "minutes",
    "seconds"
  ],
  components: {
    Timer,
    StartStop
  }
}
</script>
<style>
</style>
```

src/components/TimerStartCount.vue file

```
<template>
  <div>
     <Timer v-bind:timername="timername"
            v-bind:minutes="minutes"
          v-bind:seconds="seconds" v-on:startstop="startstop">
      <StartStop v-bind:timername="timername" v-bind:stop="1" />
    </Timer>
    <br>
    <b>Starts</b> : {{count}}
  </div>
</template>
<script>
```

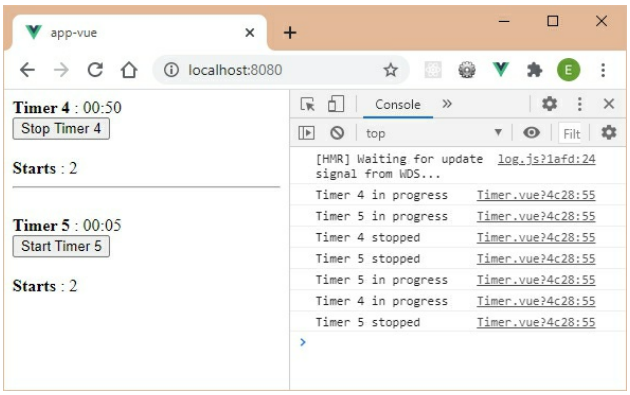
```
import Timer from "@/components/Timer.vue";
import StartStop from "@/components/StartStop.vue";
export default {
  props : [
    "timername",
     "minutes",
    "seconds"
  ],
  data() {
    return {
      count : 0
    }
  },
  methods : {
    startstop(stop) {
      if (!stop) this.count++;
    }
  },
  components: {
    Timer,
    StartStop
  }
}
</script>
<style>
</style>
```

src/App.vue file

```
<template>
  <div>
    <TimerStartCount timername="Timer 4" minutes="1" seconds="15">
    </TimerStartCount>
    <hr><br>
    <TimerStartCount timername="Timer 5" minutes="0" seconds="5">
    </TimerStartCount>
```

```
  </div>
</template>

<script>

import TimerStartCount from "@/components/TimerStartCount.vue";

export default {
  components: {
    TimerStartCount
  }
}
</script>

<style>

</style>
```

Let’s still check that our components are working properly:



Element, Elements, ContextMenu and ItemMenu components

Let us take again the components related to the management of a list of elements. These are the components:

- Element, used to manage a list element,
- Elements, allowing you to manage all the elements of the list,
- ContextMenu, used to display a context menu at the location clicked on the list item,
- ItemMenu, used to display an item from the context menu.
- The App component is used to display the list from the reactive variable elements which is passed to it as props.

These components can be written as .vue modules using the above rules.

First, remember that we were using an eventBus variable, accessible in Element and App components, which was used to update the reactive elements variable defined in the Vue object (vm object).

We will therefore create an eventBus.js file in the assets directory which will contain the management of this eventBus variable (which will be exported to be usable in the Element and App components).

src/assets/eventBus.js file

```
import Vue from "vue"
import vm from "@/main.js"
var eventBus = new Vue();
eventBus.$on("remove", function(index) {
```

```
  vm.elements = vm.elements.filter(function(element, i) {
    if (i == index) return false;
    else return true;
  });
});
eventBus.$on("modify", function(index, value) {
  vm.elements[index].text = value;
});
eventBus.$on("add", function() {
  var text = "Element " + (vm.elements.length + 1);
  vm.elements.push({text});
});
eventBus.$on("displaycontextmenu", function(contextmenu) {
  vm.contextmenu = contextmenu;
});
eventBus.$on("hidecontextmenu", function() {
   if (vm.contextmenu)
    vm.contextmenu.$parent.displayContextMenu = false;
});
export default eventBus;
```

The Vue object here named vm corresponds to the one created in the main.js file. This main.js file is slightly modified to take into account the use of the App component including the elements props (the standard main.js file does not allow specifying props to the App component).

src/main.js file

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

var vm = new Vue({
```

```
//  render: h => h(App)
  data : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ]
  },
  components : {
    App
  },
  template : `
    <App v-bind:elements="elements" />
  `
}).$mount('#app')

export default vm;
```

We export the vm variable so that we can use it in the eventBus.js file that includes it.

> As we indicate a template in the Vue object, we must configure the Vue compiler so that it is executed in runtime (otherwise an error occurs during the execution of the page). You must therefore create a new vue.config.js file which will contain this option, and restart the server for it to be taken into account (see https://cli.vuejs.org/config/#global-cli-config) .

The vue.config.js file is located in the main directory of the server, in the same location as the configuration files such as package.json. We specify this option in

the vue.config.js file.

vue.config.js file (server's root)

```
module.exports = {
  runtimeCompiler: true
}
```

> Once this file has been created or modified, restarting the server is mandatory.

The App component file is as follows:

src/App.vue file

```
<template>
  <div style="position:relative; height:100%;" v-on:click="click">
    <button v-on:click="add">Add Element</button>
    <Elements v-bind:elements="elements" />
  </div>
</template>

<script>

import Elements from "@/components/Elements.vue";
import eventBus from "@/assets/eventBus.js";

export default {
  props : [
    "elements"
  ],
  methods : {
    add() {
      eventBus.$emit("add");
    },
    click() {
      eventBus.$emit("hidecontextmenu");
    }
  },
```

```
 components: {
  Elements
 }
}
</script>

<style>

</style>
```

The other files to write are for the Element, Elements, ContextMenu, and ItemMenu components. We write them in the usual way, including the necessary files and declaring the components in the components section.

src/components/Element.vue file

```
<template>
 <li style="margin-top:10px;">
   <input v-if="modifyOn" type="text"
        v-on:keydown.enter="modify"
        v-on:blur="modify"
          v-bind:value="text"
        ref="input">
  <div v-else>
     <span v-on:dblclick="modifyOn=true"
       v-on:click.stop="displayCtxMenu" style="cursor:pointer;">
     {{text}}
    </span>
    <ContextMenu v-show="displayContextMenu" v-bind:x="x" v-
bind:y="y" />
    <button style="margin-left:10px; font-size:11px;"
         v-on:click="remove">
      Remove
    </button>
```

```
    </div>
  </li>
</template>
<script>
import ContextMenu from "@/components/ContextMenu.vue";
import eventBus from "@/assets/eventBus.js";
export default {
  props : [
    "text",
    "index"
  ],
  data() {
    return {
      modifyOn : false,
      displayContextMenu : false,
        x : 0,
      y : 0
    }
  },
  created() {
    this.$on("modify", function() {
      this.modifyOn = true;
      this.displayContextMenu = false;
    });
    this.$on("remove", function() {
      this.remove();
      this.displayContextMenu = false;
    });
  },
  updated() {
    if (this.$refs.input) this.$refs.input.focus();
  },
  methods : {
    remove() {
      eventBus.$emit("remove", this.index);
    },
```

```
    modify(event) {
      this.modifyOn = false;
      var newText = event.target.value;
      eventBus.$emit("modify", this.index, newText);
    },
    displayCtxMenu(event) {
      this.x = event.clientX;
      this.y = event.clientY;
      this.displayContextMenu = true;
      this.$parent.$children.forEach((child) => {
          if(child.index != this.index)
         child.displayContextMenu = false;
      });
      eventBus.$emit("displaycontextmenu", this.$children[0]);
    }
  },
  components: {
    ContextMenu
  }
}
</script>
<style>
</style>
```

src/components/Elements.vue file

```
<template>
  <div>
    <ul v-if="elements.length">
        <Element v-for="(element, index) in elements" style="margin-
top:10px;"
                 v-bind:key="index"
                 v-bind:index="index"
                 v-bind:text="element.text">
      </Element>
    </ul>
```

```
    <div v-else><br>Empty List</div>
  </div>
</template>
<script>
import Element from "@/components/Element.vue";
export default {
  props : [
    "elements"
  ],
  data() {
    return {
    }
  },
  components: {
    Element
  }
}
</script>
<style>
</style>
```

src/components/ContextMenu.vue file

```
<template>
   <div v-bind:style="{position:'absolute', top:y + 'px', left:x + 'px',
                       backgroundColor:'gainsboro'}">
    <ul style="margin-left:0px; padding:5px; list-style-type:none;
cursor:pointer;">
      <ItemMenu v-on:click="modify">Modify</ItemMenu>
      <ItemMenu v-on:click="remove">Remove</ItemMenu>
    </ul>
  </div>
</template>
<script>
import ItemMenu from "@/components/ItemMenu.vue";
export default {
```

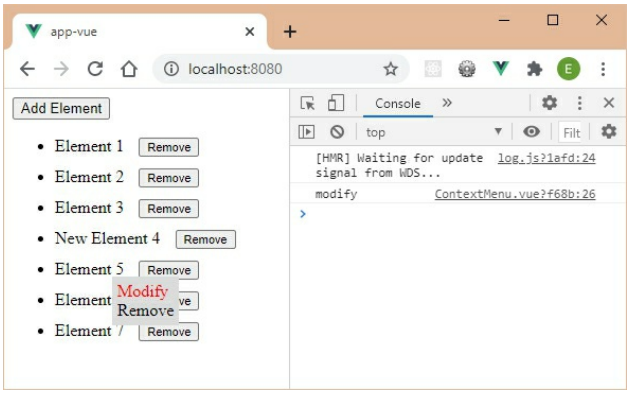
```
  props : [
    "x",
    "y"
  ],
  data() {
    return {
    }
  },
  methods : {
    modify() {
      console.log("modify");
      this.$parent.$emit("modify");
    },
    remove() {
      this.$parent.$emit("remove");
    }
  },
  components: {
    ItemMenu
  }
}
</script>
<style>
</style>
```

src/components/ItemMenu.vue file

```
<template>
   <li v-on:click="$emit('click')"
         v-on:mouseover="mouseover" v-on:mouseout="mouseout"
       v-bind:style="{color:color}">
     <slot/>
  </li>
</template>
<script>
export default {
```

```
  data() {
    return {
      color:""
    }
  },
  methods : {
    mouseover() {
      this.color = "red";
    },
    mouseout() {
      this.color = "";
    }
  },
  components: {
  }
}
</script>
<style>
</style>
```

It remains to verify that everything works identically:



Thanks to this example, we saw how to modify the App component used by default, and pass props to it using reactive variables. In addition, we saw how to outsource the management of the eventBus event bus

and make it accessible to files that require it.

> Before continuing, we return the src/main.js file to its initial state:

src/main.js file

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

Insertion of directives into the project

We previously wrote new directives using the Vue.directive() method. Let's see how to integrate these guidelines into our application.

Creation of files associated with directives

First, let's create a directives directory in the src directory, which will contain the JavaScript files for our directives. Then create a file for each of our directives:

- log.js file for the v-log directive,
- hide.js file for the v-hide directive.

src/directives/log.js file

```
import Vue from "vue"
Vue.directive("log", {
  inserted(el) {
    console.log("v-log directive applied on el element : ", el);
    console.log("Whose parent is:", el.parentElement);
  }
});
```

The directive file must include the Vue module in order to access the Vue.directive() method.

src/directives/hide.js file

```
import Vue from "vue"
Vue.directive("hide", function(el, binding) {
  var hide = binding.value;
  console.log("hide = " + hide);
  if (hide) el.style.display = "none";
});
```

The code of each of these directives are identical to those written previously (by adding the import Vue from "vue" instruction at the beginning of the file).

Using directives in a component

To use each of these directives in a component, just import the .js file for each one.

Let's use the App component which displays three paragraphs using these v-log and v-hide directives.

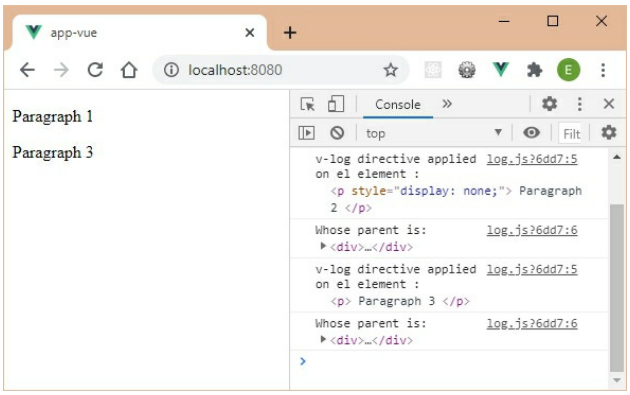
src/App.vue file

```
<template>
  <div>
    <p> Paragraph 1 </p>
```

```
    <p v-hide="true" v-log> Paragraph 2 </p>
    <p v-log> Paragraph 3 </p>
  </div>
</template>

<script>

import {} from "@/directives/hide.js";
import {} from "@/directives/log.js";

export default {
  components: {
  }
}
</script>

<style>

</style>
```

Each directive is imported using the import {} from "@/directives/name.js" statement, if name matches the name of the directive.

Let's verify that the second paragraph is invisible and that the v-log directives display messages in the console.



Inserting filters into the project

Inserting filters into the project follows the same process as that of directives. We create a filters directory in the src directory, in which we then create a JavaScript file associated with each filter.

Let's use the upper and nospaces filters created previously. We register them respectively in the upper.js and nospaces.js files, which we then import into the App.vue file.

src/filters/upper.js file

```
import Vue from "vue"
Vue.filter("upper", function(value) {
  return value.toUpperCase();
});
```

src/filters/nospaces.js file

```
import Vue from "vue"
Vue.filter("nospaces", function(value) {
  return value.replace(/ /g, "");
});
```

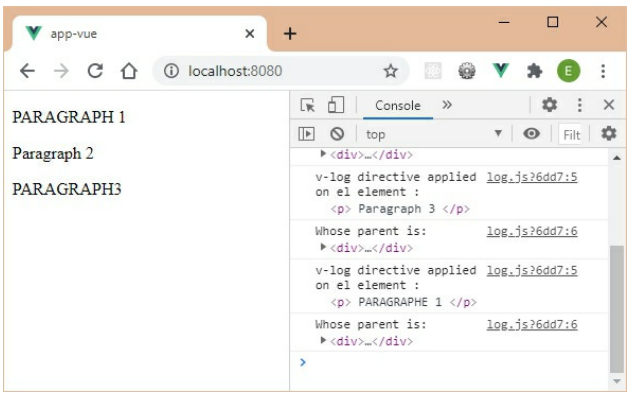
The App component displays three paragraphs, the first and last of which use the preceding filters.

src/App.vue file

```
<template>
  <div>
    <p v-log> {{"Paragraph 1"|upper}} </p>
    <p> Paragraph 2 </p>
    <p> {{"Paragraph 3"|nospaces|upper}} </p>
```

```
  </div>
</template>

<script>

import {} from "@/directives/hide.js";
import {} from "@/directives/log.js";
import {} from "@/filters/upper.js";
import {} from "@/filters/nospaces.js";

export default {
  components: {
  }
}
</script>

<style>

</style>
```

We also keep the previous v-log directives, which can be used with filters.



Inserting external JavaScript files into the project

Finally, suppose we need to use one or more external

JavaScript files in our project.

We assume here that we want to use the two JavaScript functions alertMsg1(msg) and alertMsg2(msg) which both display the message in the console (they are used to encapsulate the console.log() method of JavaScript) .

Let's write these two functions in an external JavaScript file, named here helpers.js, located for example in the assets directory.

src/assets/helpers.js file

```
function alertMsg1(...msg) {
  console.log(...msg);
}
function alertMsg2(...msg) {
  console.log(...msg);
}
export { alertMsg1, alertMsg2 };
```

As the number of parameters transmitted to the console.log() method is variable, we use the ES6 explode … function to transmit these parameters.

The log.js filter file uses both of these functions. We therefore import the helpers.js file into the log.js file.

src/filters/log.js file

```
import Vue from "vue"
import { alertMsg1, alertMsg2 } from "@/assets/helpers.js"
Vue.directive("log", {
  inserted(el) {
    alertMsg1("v-log directive applied on el element : ", el);
```

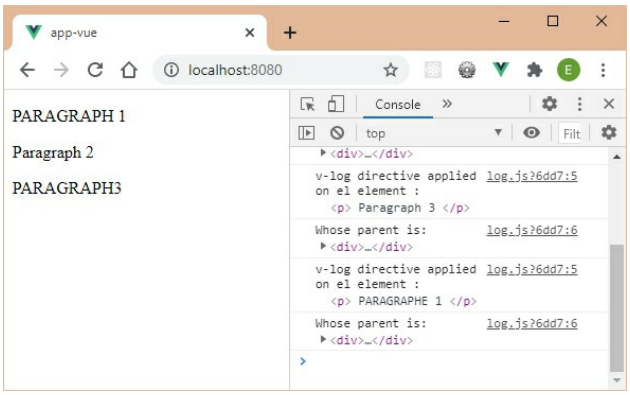
```
    alertMsg2("Whose parent is:", el.parentElement);
  }
});
```

The App.vue file is identical to the previous one:

src/App.vue file

```
<template>
  <div>
    <p v-log> {{"Paragraph 1"|upper}} </p>
    <p> Paragraph 2 </p>
    <p> {{"Paragraph 3"|nospaces|upper}} </p>
  </div>
</template>

<script>

import {} from "@/directives/hide.js";
import {} from "@/directives/log.js";
import {} from "@/filters/upper.js";
import {} from "@/filters/nospaces.js";

export default {
  components: {
  }
}
</script>

<style>

</style>
```

The displays in the console are now produced by the alertMsg1() and alertMsg2() functions.



Dynamic creation of components

We have already used the dynamic creation of components when a contextual menu was created dynamically (see chapter 3 "*Managing the elements of a list as components*").

But when using Vue CLI, the components are organized in .vue files and the dynamic creation of the components has some differences from what was explained previously.

We want to dynamically create a list of elements, which will be associated with an Elements component with the elements props. The elements props represents an array of elements to display in the list.

In order to show the differences, we'll first do it the traditional way, inserting the Elements component directly into the main App component, and then we'll create the Elements component dynamically within the App component.

> You can find more examples of using components in JavaScript in the official documentation, for example https://vuejs.org/v2/guide/render-function.html.

Elements component inserted in the traditional way

Let's show the classic way to display an Elements component in the main App component of the application.

src/App.vue file

```
<template>
  <div ref="list">
    <Elements v-bind:elements="elements"/>
  </div>
</template>

<script>

import Elements from "@/components/Elements.vue"

export default {
  data() {
    return {
      elements : [
        "Element 1",
        "Element 2",
        "Element 3",
        "Element 4",
        "Element 5"
      ]
    }
  },
```

```
  components : {
    Elements
  }
}

</script>

<style>

</style>
```

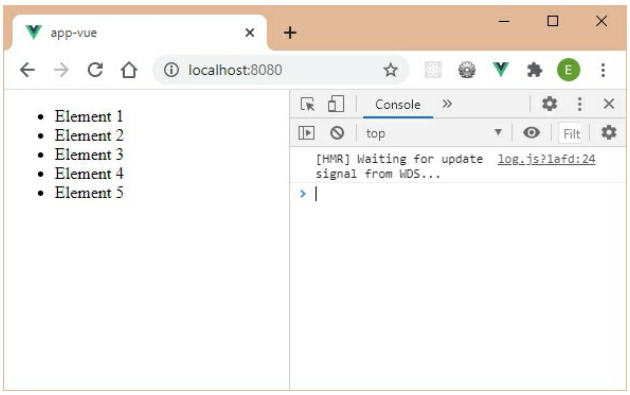
The Elements component is inserted directly into the template of the App component. The list of elements is obtained by a reactive variable elements, then passed in the elements props of the Elements component.

The Elements component is as follows:

src/components/Elements.vue file

```
<template>
  <div>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
        {{element}}
      </li>
    </ul>
  </div>
</template>
<script>
export default {
  props : [
    "elements"
  ]
}
</script>
```

The result is the expected one:



Elements component dynamically inserted

If we dynamically create the Elements component, it should no longer be registered in the template of the App component.

On the other hand, the mounted() method of the App component will be used to create an instance of this component, and to insert it into the DOM tree.

The App component becomes (the Elements component is unchanged):

src/App.vue file

```
<template>
  <div ref="list">
  </div>
</template>

<script>

import Vue from "vue";
import Elements from "@/components/Elements.vue"

export default {
  data() {
    return {
```

```
      elements : [
        "Element 1",
        "Element 2",
        "Element 3",
        "Element 4",
        "Element 5"
      ]
    }
  },
  components : {
    //Elements
  },
  mounted() {
    // <Elements v-bind:elements="elements"/>
    var Klass = Vue.extend(Elements);
    var elements = new Klass({
      propsData : {
        elements : this.elements
      }
    });
    elements.$mount();
    this.$refs.list.appendChild(elements.$el);
  }
}

</script>

<style>

</style>
```

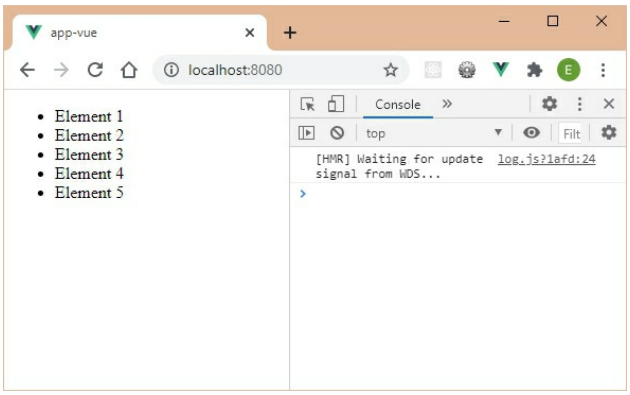
The mounted() method is the heart of the process:

- We use Vue.extend() which allows, from a .vue module, to transform it into a class representing a Vue component,

- Then we instantiate an elements object of this class (here called Klass), passing it the elements props of value this.elements.
- This elements object is then associated with Vue using the $mount() method,
- Then the corresponding DOM element (elements.$el) is inserted into the DOM tree using the appendChild() method.

Note that since we are using an unconventional mechanism here, the components property of the App component may no longer contain the reference to the Elements component (hence its commenting).

Let's check that it works:



Conclusion

The Vue CLI utility allows you to create a tree structure for a Vue project, whose components are .vue files used as JavaScript modules.

We will use this tree structure in the rest of our work.

7 – USING AJAX

Ajax is a fairly classic concept that allows you to make requests to the server while staying on the current page. This makes it possible to refresh the current page from the information retrieved from the server, for example in a database.

Over time, many libraries have been created to facilitate Ajax requests to a server. Currently, one of the most used libraries with Vue is the Axios library (see https://github.com/axios/axios). This is the one we will use here for our examples.

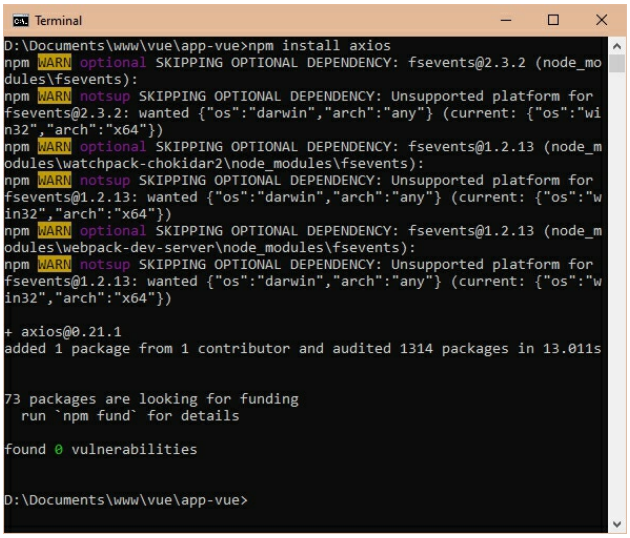
Install the Axios library

It is assumed that the tree structure created with Vue CLI (see previous chapter) is used in order to benefit from the modular structure of the components.

The Axios library is installed from the npm install axios command, which you type from the main directory of the application created with Vue CLI (here the app-vue directory).

Install the Axios library

```
npm install axios
```



The node_modules directory of the application has been enriched with the axios directory (and any other packages dependent on it).

> Remember to restart the server with npm run serve in case you stopped it to install Axios.

Use Axios to query an external server

We propose here to show how to access an external server via an API provided by this server. Many services are offered on the internet, allowing access, for example:

- To film data (https://www.themoviedb.org/),
- Weather data for cities

(https://openweathermap.org/),

- Country characteristics (http://restcountries.eu/),
- Etc.

Among these available services, we have selected the one allowing access to the characteristics of the countries (language, capital, currency, flag, etc.), accessible at the URL http://restcountries.eu/.

The service is very easy to use, there is no need to register on the site to obtain an access key for the API (which is frequently requested on most sites).

Example of access to the server API from a browser

Before we start coding with Axios, let's run some URLs in the browser to see how the server API works.

Let's display the details for a particular country, here France. For this, we introduce the API URL allowing access to this information https://restcountries.eu/rest/v2/name/france (as indicated on the site https://restcountries.eu/).

Once this URL has been entered in the browser, it displays the response from the server in JSON format, here an array of objects, in which we find:

- The name field: corresponds to the name of the country in English,
- The nativeName field: corresponds to the name of the country in the language of the country,
- The topLevelDomain field: corresponds to the internet extension associated with the websites of this country (here ".fr"),
- The flag field: corresponds to the URL displaying the flag of the country,
- Etc.

swqxcfd

Other server API accesses are described on the page http://restcountries.eu/#api-endpoints.

For our example, we will only use the one allowing to retrieve the information of a country by having indicated its name, therefore in the form https://restcountries.eu/rest/v2/name/france.

Using the Server API with Axios

Let's use the Axios API to retrieve the information like that shown in the previous answer. You can find more information on using Axios at https://github.com/axios/axios.

Here we want to make an Ajax call on the URL https://restcountries.eu/rest/v2/name/france using

Axios, and display the results returned in a reactive data variable of the page.

For this, we use the App component:

src/App.vue file

```
<template>
  <div>
    {{data}}
  </div>
</template>

<script>

import axios from "axios";

export default {
  data() {
    return {
      data : ""
    }
  },
  created() {
    axios
    .get("https://restcountries.eu/rest/v2/name/france")
    .then((response) => {
      console.log(response.data);
      this.data = response.data;
    })
    .catch((response) => {
      console.log(response);
      this.data = response;
    });
  },
  components: {
  }
```

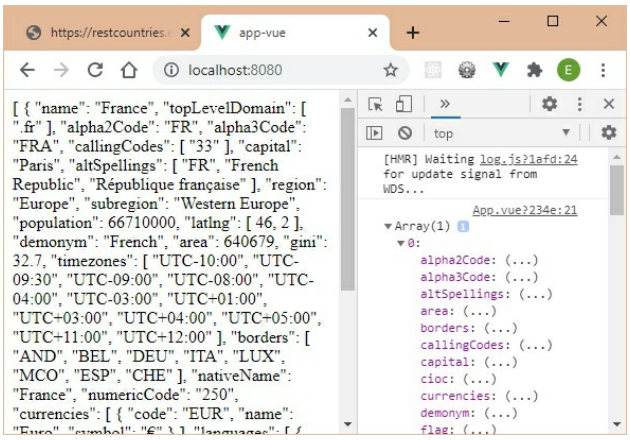
```
}
</script>

<style>

</style>
```

The Axios module is imported by the import axios from "axios" statement, then the request to the URL https://restcountries.eu/rest/v2/name/france of the external server is made using the get(url). This method returns a Promise object on which we can chain the then(callback) and catch(callback) methods:

- The callback function of the then(callback) method is executed if the request was successful,
- The callback function of the catch(callback) method is executed if the request produces an error.

Each of the callback functions has a response parameter, which is used to take the response into account. If the request goes well, Axios initializes the response.data parameter with the response from the server. It is this content that is displayed in the reactive data variable of the App component.

> Note the use of the ES6 => notation for callback functions, so as not to lose the value of this in the callback function.



Let's improve our component to produce a more synthetic display. We want to display the following information in the page:

- The name of the country,
- The region of the world to which it belongs,
- The URL of its flag,
- And finally the image of its flag.

The App component is modified for this:

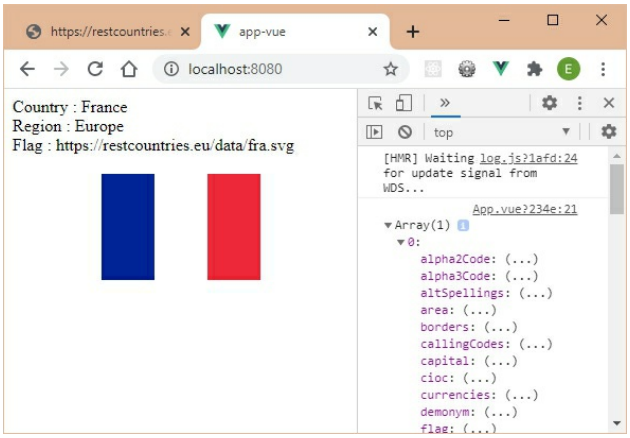
src/App.vue file

```
<template>
  <div>
    <div>Country : <span>{{name}}</span> </div>
    <div>Region : <span>{{region}}</span> </div>
    <div>Flag : <span>{{flag}}</span> </div>
    <br>
    <div style= "text-align:center"><img v-bind:src="flag" height="100"
/></div>
  </div>
</template>

<script>
```

```
import axios from "axios";

export default {
  data() {
    return {
      data : {}
    }
  },
  computed : {
    name() {
      return this.data.name;
    },
    flag() {
      return this.data.flag;
    },
    region() {
      return this.data.region;
    }
  },
  created() {
    axios
    .get("https://restcountries.eu/rest/v2/name/france")
    .then((response) => {
      this.data = response.data[0];
    })
    .catch((response) => {
      console.log(response);
    });
  },
  components: {
  }
}
</script>

<style>

</style>
```

The reactive variable data is now an object initialized with the first element of the response.data array returned by Axios.

From the reactive variable data, we create new calculated properties (computed section of the component) which returns the name of the country, the region and the URL of the flag. These calculated properties are then used in the template.



Creating a user interface

Let's improve our interface by allowing you to enter the country for which we want to display information.

src/App.vue file

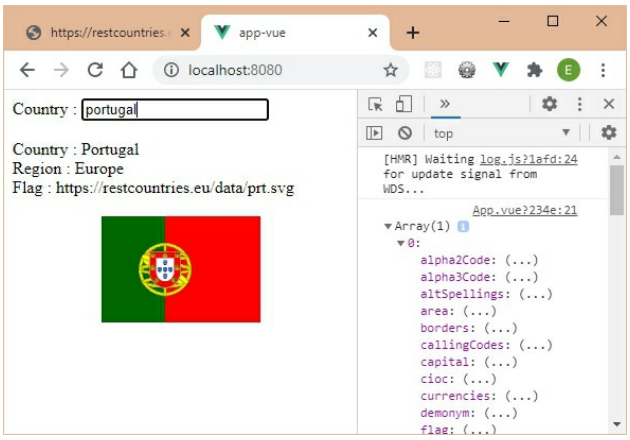
```
<template>
 <div>
  <div>Country : <input type="text" v-on:keyup.enter="valid"> </div>
  <br>
  <div>Country : <span>{{name}}</span> </div>
  <div>Region : <span>{{region}}</span> </div>
  <div>Flag : <span>{{flag}}</span> </div>
```

```
   <br>
   <div v-if="!data.name && country"> No country found with this name
</div>
   <div style= "text-align:center"><img v-bind:src="flag" height="100" />
</div>
  </div>
</template>

<script>

import axios from "axios";

export default {
  data() {
    return {
      data : {},
      country : ""
    }
  },
  computed : {
    name() {
      return this.data.name;
    },
    flag() {
      return this.data.flag;
    },
    region() {
      return this.data.region;
    }
  },
  methods : {
    valid(event) {
      this.country = event.target.value.trim();
      if (this.country) {
        axios
        .get("https://restcountries.eu/rest/v2/name/" + this.country)
        .then((response) => {
```

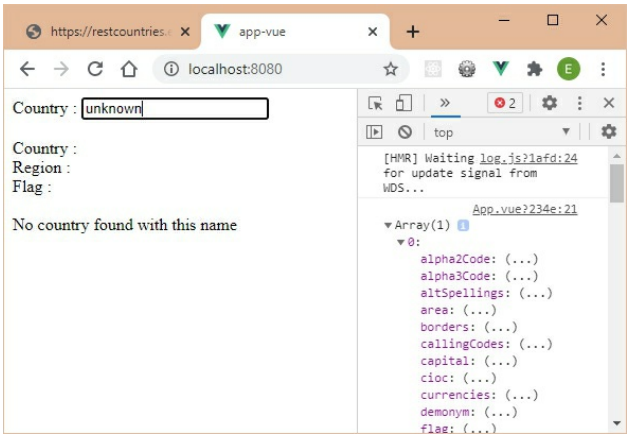
```
          this.data = response.data[0];
        })
        .catch((response) => {
          console.log(response);
          this.data = { };
        });
      }
      else this.data = { };
    }
  },
  created() {
  },
  components: {
  }
}
</script>

<style>

</style>
```

The name of the country is entered in a field, and when the entry is validated by Enter, the valid() method is called. The reactive variable country is initialized from the content of the input field (event.target.value), then the Ajax request is performed (if the input field is not empty, otherwise the screen fields are deleted by the reset to {} of the reactive variable data).

Note the error message displayed only if the reactive variable data contains nothing even though the country has been entered and validated (reactive variable country).



If you enter an unknown country:



Use Axios to make a request on our server (PHP server for example)

The previous example uses an external server using an API provided by that server. But very often we want to use our own servers on which we have installed our data. It is this data that we want to manipulate now.

Suppose that we have a PHP server on which PHP scripts are running allowing to manipulate data from a MySQL database. We want to access this data through

our Vue application which runs on the server installed by Vue CLI. This last server does not know anything about PHP (it works under Node.js), but it must access data on a PHP server.

> The principle described here works for all types of servers, and not just PHP.

Vue does this. All you have to do is configure the vue.config.js file (located at the root of the application) indicating that URLs starting with a specified string must be propagated to another server.

vue.config.js file

```
module.exports = {
  devServer: {
    proxy: {
      '^/api-php/': {
        target: 'http://localhost:80',
        changeOrigin: true
      }
    }
  }
}
```

> Once the vue.config.js file has been modified, you must restart the server on which the Vue application is running for the changes to be taken into account (by typing the npm run serve command again).

We indicate here that the URLs starting with the string /api-php/ (or any other string) have as destination

server the one indicated in the target field, here http://localhost:80 which is the server under PHP. This server must contain an api-php directory under the root, in which the PHP scripts called from the Vue program will be deposited.

Display the server response in raw form

Under the root of the PHP server, create the api-php directory containing the following get_names.php file. It is this file that will be used during the Ajax request.

api-php/get_names.php file on PHP server

```
<?php
echo json_encode(array(
  "Aaron Abbott",
  "Abe Adams",
  "Abraham Adamson",
  "Adam Adcock",
  "Adel Addams",
  "Adrian Adhams",
  "Aidan Aindreis",
  "Al Allan",
  "Alan Allen",
  "Alban Allison",
  "Albert Alyn",
  "Albin Ambrose",
  "Alexander Anderson",
  "Alfred Andres",
  "Allan Andrew",
  "Allen Andrews",
  "Allister Anew",
  "Alphonse Anthony",
```

```
  "Alvin Apple",
  "Amos Archdeacon",
  "Andre Archer",
  "Andrew Ash",
  "Andy Ashley",
  "Angus Atcock",
  "Anthony Austen",
  "Antony Austin",
  "Arnold Aylen",
  "Arthur Aylin",
  "Augustus Ayling"
));
?>
```

This PHP script returns in JSON form a list of names that will be used in the Vue program which performs the Ajax request.

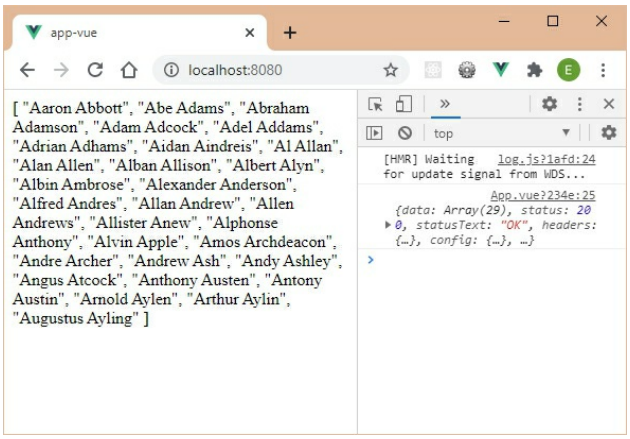
The App component that makes the Ajax call is as follows:

src/App.vue file

```
<template>
  <div>
    {{names}}
  </div>
</template>

<script>

import axios from "axios";

export default {
  data() {
    return {
      names : { }
```

```
    }
  },
  computed : {
  },
  methods : {
  },
  created() {
    axios
    .get("/api-php/get_names.php")
    .then((response) => {
      console.log(response);
      this.names = response.data;
    })
    .catch((response) => {
      console.log(response);
      this.names = response;
    });
  },
  components: {
  }
}
</script>

<style>

</style>
```

The Ajax request is made from the created() method of the App component. It retrieves the response from the server in response.data and displays it in the names reactive variable.



Display the response from the server as a list

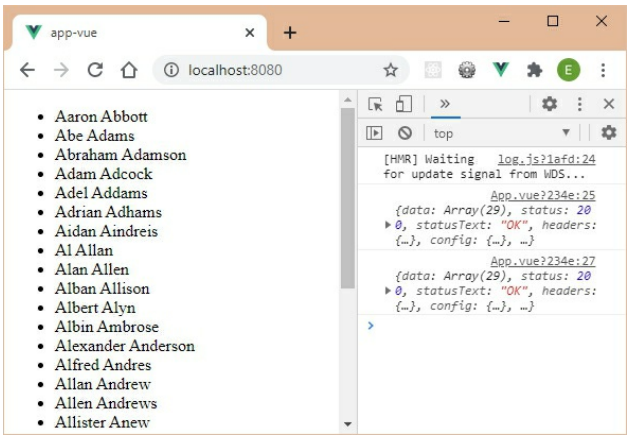
It now remains to format the received list to display it as an HTML list.

Just change the template slightly and use the v-for directive in it.

Display the received list as an HTML list

```
<template>
  <div>
    <ul>
      <li v-for="(name, index) in names" v-bind:key="index"> {{name}}
</li>
    </ul>
  </div>
</template>

<script>

import axios from "axios";

export default {
  data() {
    return {
```

```
    names : { }
   }
  },
  computed : {
  },
  methods : {
  },
  created() {
   axios
   .get("/api-php/get_names.php")
   .then((response) => {
    console.log(response);
    this.names = response.data;
   })
   .catch((response) => {
    console.log(response);
    this.names = response;
   });
  },
  components: {
  }
}
</script>

<style>

</style>
```


app-vue
localhost:8080
Aaron Abbott
Abe Adams
Abraham Adamson
Adam Adcock
Adel Addams
Adrian Adhams
Aidan Aindreis
Al Allan
Alan Allen
Alban Allison
Albert Alyn
Albin Ambrose
Alexander Anderson
Alfred Andres
Allan Andrew
Allen Andrews
Allister Anew
top
[HMR] Waiting for update signal from WDS... log.js?1afd:24
App.vue?234e:25
{data: Array(29), status: 20
0, statusText: "OK", headers: {…}, config: {…}, …}
App.vue?234e:27
{data: Array(29), status: 20
0, statusText: "OK", headers: {…}, config: {…}, …}

8 – USING VUEX

Vuex is a library making it easier to manage the state of our application. The state of the application is represented by the reactive variables found in the components (data section).

For this, Vuex proposes to centralize certain reactive variables which will thus be common to several components, which will allow them to be used more easily between several components.

> Note that only certain reactive variables will be centralized in Vuex, and not all reactive variables in the application. Indeed, only the state variables which need to be common to several components will be concerned. The other state variables will remain local to the component in which they are defined (and will therefore not be managed by Vuex).

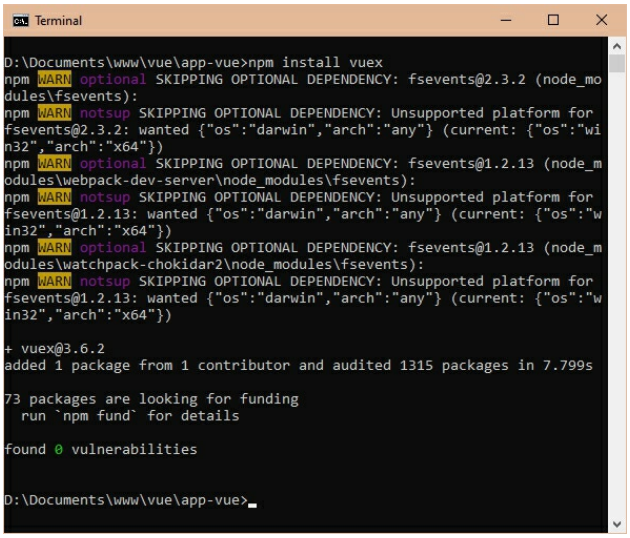
Install Vuex

Vuex is a module that is installed with the npm install vuex command. Once installed, the module is

registered in the node_modules directory of the Vue application created previously by Vue CLI.

Install the vuex module

```
npm install vuex
```



> Remember to restart the server with npm run serve in case you stopped it to install Vuex.

Vuex principle

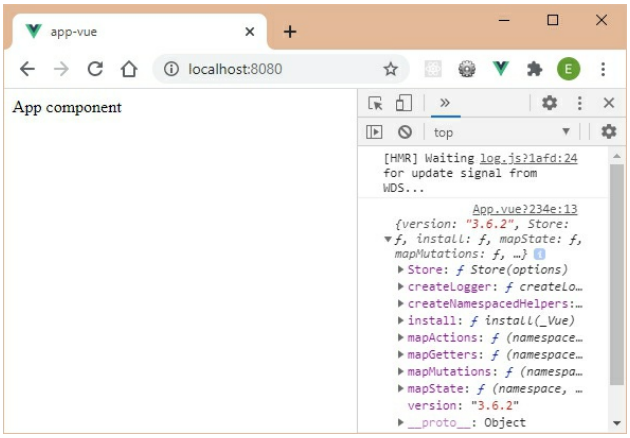
Vuex allows you to manage the state of the application thanks to the vuex module imported into the JavaScript program using the import Vuex from "vuex" statement. This module is then linked to the Vue application by the Vue.use(Vuex) instruction.

Let's write these instructions in an App.vue file that will be displayed using the URL http://localhost:8080.

Import and link Vuex to a Vue app (src/App.vue file)

```
<template>
  <div>
    App component
  </div>
</template>

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);
console.log(Vuex);

export default {
}

</script>

<style>

</style>
```

The template of the App component here only includes the display of its name, while the imported Vuex module is displayed in the console.



The Vuex module displayed in the console mainly

includes methods that we will study later. The first method used will be Vuex.Store() which allows you to create the store variable which manages the state of the application.

Create the application store

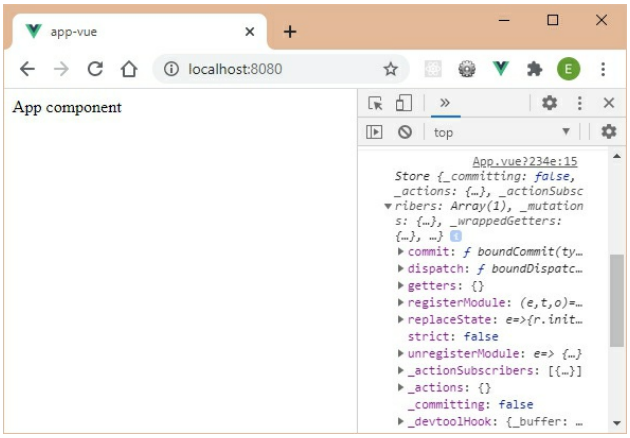
Let's use Vuex.Store() to create the application store, required to use Vuex. This method is in fact a class used to create the store object (here called the store).

The store is what manages the state of the Vue application with Vuex, offering methods of reading and updating the state.

Create and display the application store (src/App.vue file)

```
<template>
  <div>
    App component
  </div>
</template>

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store();
console.log(store);

export default {
}
```

```
</script>

<style>

</style>
```



The store variable created by new Vuex.Store() is a JavaScript object with, among other things, the state property.

The state property of the store object is the one that stores the state of the application. It is a JavaScript object (called the state here), each property of which represents one of the reactive variables for which we want to centralize state management.

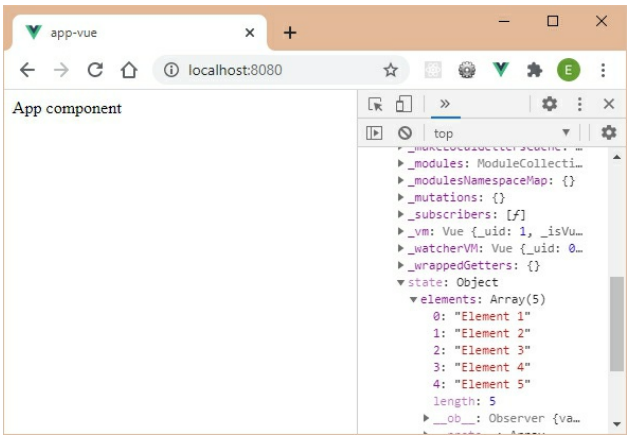
Initialize the state of the application

We can indicate as a parameter of Vuex.Store() an object whose state property indicates the initial state of the application. This state object indicated as a parameter of Vuex.store() will initialize the one stored in store.state (which is initially empty).

Let's initialize the state object with a list of elements that will be displayed: "Element 1",…, "Element 5".

Initialize state(src/App.vue file)

```
<template>
  <div>
    App component
  </div>
</template>

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  }
});
console.log(store);

export default {
}

</script>

<style>

</style>
```



The store.state property now has an elements property initialized with the five elements specified.

This elements property is inserted in the state managed by Vuex. This means that any modification of this property (by updating an element, or by adding or deleting an element) will be considered by Vue as being the update of a reactive variable, and will therefore cause a refresh of the components where it is used.

Use state in the application

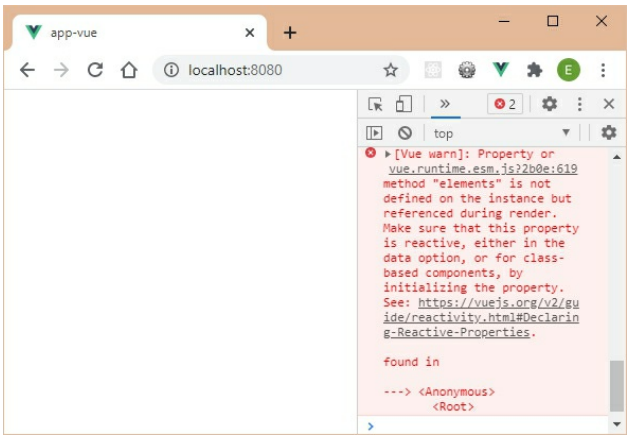
Let's display the contents of the elements property of the state. For this, we use the v-for directive on the <li> element.

Display the contents of the elements property of the state

```
<template>
  <div>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
```

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

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  }
});
console.log(store);

export default {
}

</script>

<style>

</style>
```

The displayed result shows an error (but easy to correct):



The error message explains that the direct use of the elements variable in the App component is not possible because this variable is not known as a reactive variable in the App component.

We should have defined this elements variable in the data section of the component, but if we do that, the elements variable will no longer be that of the store managed by Vuex (and it must be!).

Use the state in the application via calculated properties

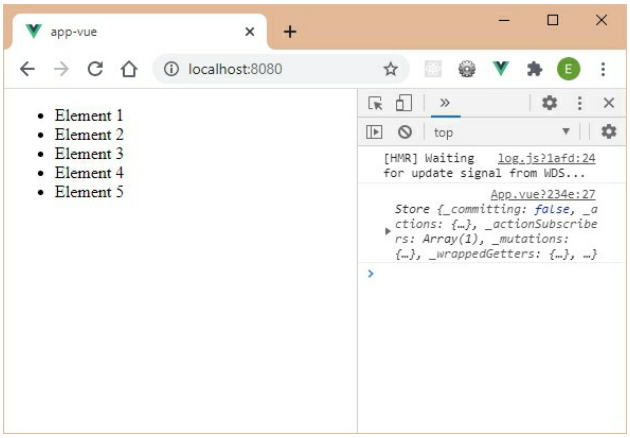
The solution is to create a computed property (computed section of the component) which will be responsible for returning the elements variable of the store (stored in store.state.elements). Then we use this property calculated in the template of the App component.

Let's call this computed property elements, which will be used under this name in the template.

Use a computed property that returns the elements property of the state in the store (src/App.vue file)

```
<template>
  <div>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
  </div>
</template>

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  }
});
console.log(store);

export default {
  computed : {
    elements() {
      return store.state.elements;
    }
  }
```

```
}

</script>

<style>

</style>
```



> Note that if we directly use the store.state.elements variable in the App component template, a message is displayed indicating that the store variable is unknown. Indeed, the template of a component only has access to its own reactive variables, defined in the data section, or to calculated properties defined in the computed section (what we do here with the elements property) .

Update the state of the application

Let's modify the App component to display an Add Element button to insert a new element in the list. Clicking on this button calls the add() method of the component which modifies the state by adding a new

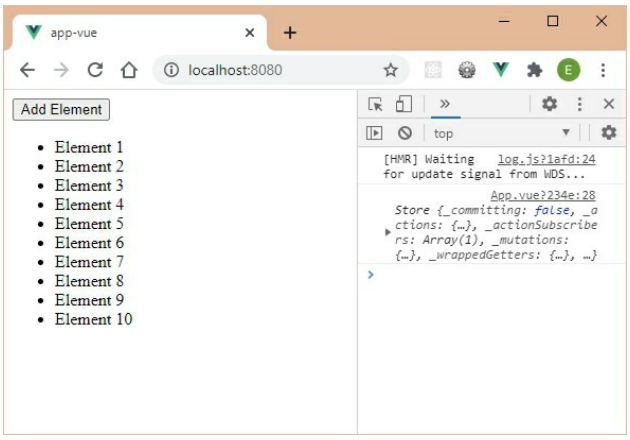
element at the end of the elements array.

Insert an item in the list (src/App.vue file)

```
<template>
  <div>
    <button v-on:click="add">Add Element</button>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
  </div>
</template>

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  }
});
console.log(store);

export default {
  computed : {
    elements() {
      return store.state.elements;
```

```
  }
 },
 methods : {
  add() {
   var element = "Element " + (store.state.elements.length + 1);
   store.state.elements.push(element);
  }
 }
}

</script>

<style>

</style>
```

After adding several items to the list:



The state positioned in the Vuex store is therefore reactive to changes, like the traditional reactive variables defined in the components.

Use the commit() method to update the state

Adding a new item to the item list was done using the store.state.elements variable directly. But this is not recommended because the direct manipulation of the state prevents the use of analysis and trace tools which allow to follow all the modifications of the state (in particular in Vue Dev Tools).

The store.commit(mutation) method is used to execute the mutation(state) method which updates the state. The mutation(state) method is defined in the options when creating the store object (in the mutations section).

Let's modify the previous program to use the store.commit(mutation) method. The mutation(state) method corresponds here to the add_element(state) method which adds a new element in the state.elements array. We will therefore use store.commit("add_element") to update the state.

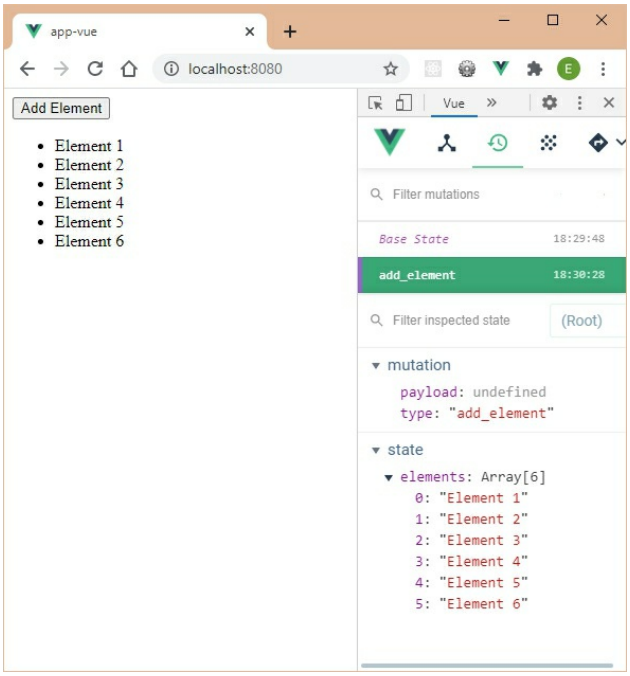
Use store.commit() to update state(src/App.vue file)

```
<template>
  <div>
    <button v-on:click="add">Add Element</button>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
  </div>
</template>
```

```
<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  },
  mutations : {
    add_element(state) {
      var element = "Element " + (state.elements.length + 1);
      state.elements.push(element);
    }
  }
});

export default {
  computed : {
    elements() {
      return store.state.elements;
    }
  },
  methods : {
    add() {
      store.commit("add_element");
    }
  }
}
```

```
</script>

<style>

</style>
```

The add_element(state) method is defined in the mutations section when creating the store object. It modifies the state indicated in parameters.



When clicking on the Add Element button, the element is added to the list, and the Vue tab also displays the mutations carried out (here the "add_element" type). With the resulting state below, for each mutation made.

Transmit arguments during the transfer

The mutation carried out during

store.commit("mutation") takes as the first parameter the state before its possible modification (as in add_element(state)).

It is also possible to indicate a second parameter which corresponds to the payload parameter, which represents an object comprising additional parameters to effect the mutation.

The mutation(state) method can therefore be written in the form mutation(state, payload), knowing that the payload parameter is the one transmitted during store.commit("mutation", payload).

Let's use the payload parameter to pass the text of the new element to insert into the elements array. Rather than calculating this new text in the add_element(state) method as before, we pass it to it as parameters in add_element(state, payload).

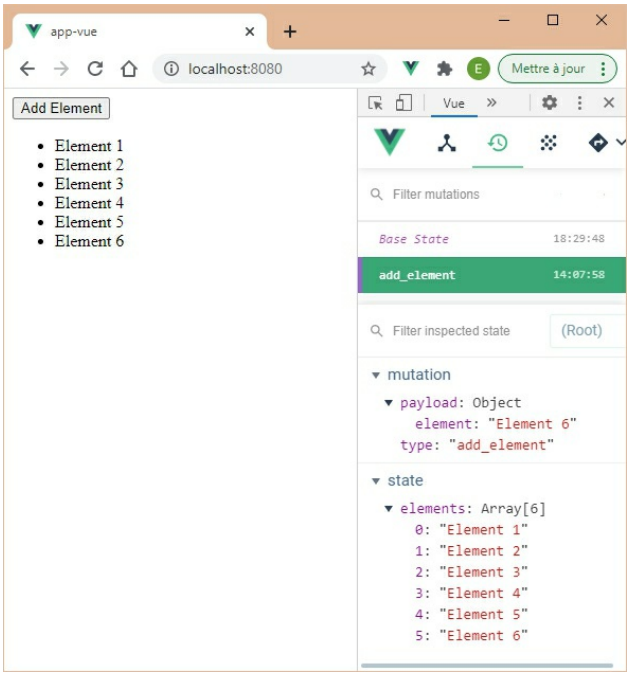
Transmit the text of the element to be inserted during the transfer

```
<template>
  <div>
    <button v-on:click="add">Add Element</button>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
  </div>
</template>
```

```
<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  },
  mutations : {
    add_element(state, payload) {
      var element = payload.element;
      state.elements.push(element);
    }
  }
});

export default {
  computed : {
    elements() {
      return store.state.elements;
    }
  },
  methods : {
    add() {
      var element = "Element " + (store.state.elements.length + 1);
      store.commit("add_element", {element : element});
    }
  }
```

```
}

</script>

<style>

</style>
```

The add() method calculates the new element to insert, and passes it as the second parameter of the store.commit() method.



We see that the payload parameter is now displayed in the mutation part.

Access to the store object from a component

Access to the store object from a component is necessary in order to:

- Read the contents of the state: for example the calculated property elements returns the elements array in the return store.state.elements statement.
- Modify the content of the state: for example by performing a mutation, as in the store.commit("add_element") instruction.

This means that the store object should be easily accessible in all components that will need to read the state's content or update it.

In the previous example, accessing the store object is easy because the store variable is declared in the App.vue file which, for now, is the only component to use it. But how do you access it when many components want to use it?

Vuex has foreseen this case by allowing, in a component, to indicate that the store object must be transmitted in all the tree structure of the components which results from it. If this is indicated in the root component (the App component), the store will be transmitted in all the components of the application.

To do this, all you have to do is indicate when creating the component, a store property having the value of the store object. This store property will be passed to all components that derive from this component.

> The store will then be accessible in each component using this.$store.
> It will also be accessible in the template of a component using $store.

Let's specify this store property when creating the App component:

Use the store property in the App component

```
<template>
  <div>
    <button v-on:click="add">Add Element</button>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
  </div>
</template>

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  },
```

```
  mutations : {
    add_element(state, payload) {
      var element = payload.element;
      state.elements.push(element);
    }
  }
});

export default {
  store : store,
  computed : {
    elements() {
      console.log(this.$store);
      return this.$store.state.elements;
    }
  },
  methods : {
    add() {
      var element = "Element " + (this.$store.state.elements.length + 1);
      this.$store.commit("add_element", {element : element});
    }
  }
}

</script>

<style>

</style>
```

Once the store property has been specified in the component, the store can be accessed by the this.$store variable in each component.

Note that thanks to the ES6 notation, we can also write more simply (assuming that the store variable has been

created previously):

Writing simplification to set the store property

```
export default {
  store,    // store : store,
  computed : {
    elements() {
      console.log(this.$store);
      return this.$store.state.elements;
    }
  },
  methods : {
    add() {
      var element = "Element " + (this.$store.state.elements.length + 1);
      this.$store.commit("add_element", {element : element});
    }
  }
}
```

The store property having the value of the variable of the same name, we can simply indicate store, instead of store: store in the definition of the component.

In case the store property has been passed to the component, the component template can also be written (directly using $store.state.elements instead of the computed elements property):

Writing the template using the store passed in the component's store property

```
<template>
  <div>
    <ul>
      <li v-for="(element, index) in $store.state.elements" v-
```

```
bind:key="index">{{element}}</li>
    </ul>
  </div>
</template>
```

Creating calculated properties in the component

We have created the computed elements property which returns the elements property of the state.

It is possible to create new calculated properties which will be useful in the component. For example :

- The elementFirst property which returns the first element of the list,
- The elementLast property that returns the last element in the list.

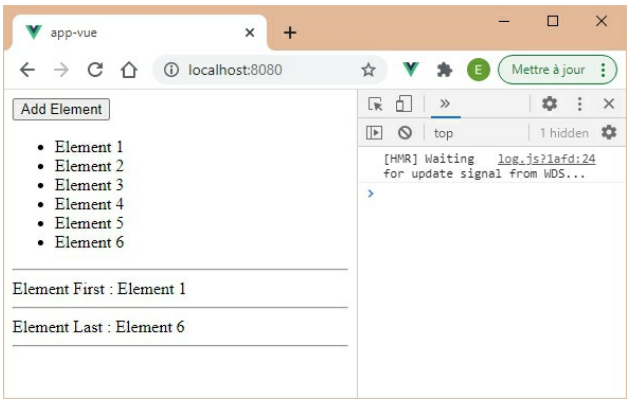
ElementFirst and elementLast properties

```
<template>
  <div>
    <button v-on:click="add">Add Element</button>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
    <hr>
   Element First : {{elementFirst}}
    <hr>
    Element Last : {{elementLast}}
    <hr>
  </div>
```

```
</template>

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  },
  mutations : {
    add_element(state, payload) {
      var element = payload.element;
      state.elements.push(element);
    }
  }
});

export default {
  store,
  computed : {
    elements() {
      return this.$store.state.elements;
    },
    elementFirst() {
      return this.$store.state.elements[0];
    },
    elementLast() {
      return this.$store.state.elements[this.$store.state.elements.length-1];
```

```
  }
 },
 methods : {
  add() {
   var element = "Element " + (this.$store.state.elements.length + 1);
   this.$store.commit("add_element", {element:element});
  }
 }
}

</script>

<style>

</style>
```

The elementFirst and elementLast properties being linked to the state in the store, they are updated each time the state is modified, for example when an element is added to the list.



Creation of getters in the store

To facilitate the creation of calculated properties in components, Vuex planned to be able to create state access methods that would be registered in the store.

These methods are called getters because they are used to return certain elements of the state.

The advantage of these methods is that they are registered in a single place (when creating the store, in the getters section), and not in each component. On the other hand, each component must call on these methods by using them in its calculated properties.

A getter takes the state as a parameter, which allows it to return the element of the state associated with this getter. But a getter can also have other parameters (for example, the index of the desired element in the elements array). Let's take a look at these two forms of getters.

Getter without parameter

Let's write two getter methods:

- The elementFirst() method returns the element of index 0 of the state.elements array,
- The elementSecond() method returns the element of index 1 of the state.elements array.

These two methods will be registered in the getters section when creating the store object.

To use them, we also write two calculated properties in the App component:

- The elementFirst calculated property will use

the elementFirst() getter,

- The elementSecond calculated property will use the elementSecond() getter.

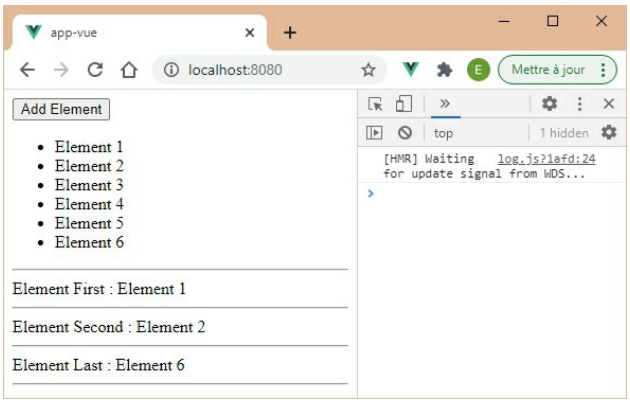
Getter elementFirst() and elementSecond()

```
<template>
  <div>
    <button v-on:click="add">Add Element</button>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
    <hr>
    Element First : {{elementFirst}}
    <hr>
    Element Second : {{elementSecond}}
    <hr>
    Element Last : {{elementLast}}
    <hr>
  </div>
</template>

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
```

```
      "Element 5"
    ]
  },
  mutations : {
    add_element(state, payload) {
      var element = payload.element;
      state.elements.push(element);
    }
  },
  getters : {
    elementFirst(state) {
      return state.elements[0];
    },
    elementSecond(state) {
      return state.elements[1];
    }
  }
});

export default {
  store,
  computed : {
    elements() {
      return this.$store.state.elements;
    },
    elementFirst() {
      //return this.$store.state.elements[0];
      return this.$store.getters.elementFirst;
    },
    elementSecond() {
      //return this.$store.state.elements[1];
      return this.$store.getters.elementSecond;
    },
    elementLast() {
      return this.$store.state.elements[this.$store.state.elements.length-1];
    },
```

```
  },
  methods : {
    add() {
      var element = "Element " + (this.$store.state.elements.length + 1);
      this.$store.commit("add_element", {element:element});
    }
  }
}

</script>

<style>

</style>
```

The calculated properties of the App component now use the getters defined in the store object, rather than directly accessing the state as before.



Getter with parameter

The getters defined above could be improved because they access index 0 of state.elements for the first, and index 1 of the same object for the second.

The improvement would be to pass the index of the element as a parameter of the getter. And therefore to

create a more generic getter which would have the following form getElementByIndex(index). The problem is that the only parameter of a getter is the state, and by no means the index of an element.

The solution consists in writing a getter which returns a function having the desired parameters, here the index of the element. Let's write the corresponding getElementByIndex() getter.

Getter getElementByIndex()

```
<template>
  <div>
    <button v-on:click="add">Add Element</button>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
    <hr>
    Element First : {{elementFirst}}
    <hr>
    Element Second : {{elementSecond}}
    <hr>
    Element Last : {{elementLast}}
    <hr>
  </div>
</template>

<script>

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);
```

```
var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  },
  mutations : {
    add_element(state, payload) {
      var element = payload.element;
      state.elements.push(element);
    }
  },
  getters : {
    elementFirst(state) {
      return state.elements[0];
    },
    elementSecond(state) {
      return state.elements[1];
    },
    getElementByIndex(state) {
      return function(index) {
        return state.elements[index];
      }
    }
  }
});

export default {
  store,
  computed : {
    elements() {
      return this.$store.state.elements;
```

```
    },
    elementFirst() {
      //return this.$store.state.elements[0];
      //return this.$store.getters.elementFirst;
      return this.$store.getters.getElementByIndex(0);
    },
    elementSecond() {
      //return this.$store.state.elements[1];
      //return this.$store.getters.elementSecond;
      return this.$store.getters.getElementByIndex(1);
    },
    elementLast() {
      return this.$store.state.elements[this.$store.state.elements.length-1];
    },
  },
  methods : {
    add() {
      var element = "Element " + (this.$store.state.elements.length + 1);
      this.$store.commit("add_element", {element:element});
    }
  }
}

</script>

<style>

</style>
```

The getElementByIndex() getter returns a function with an index parameter, which itself returns the corresponding element of the state. The getter is used in the calculated properties of the App component by transmitting the index of the desired element (here 0 or 1).

Improve the elementLast computed property

The elementLast computed property of the App component does not yet use the new getElementByIndex() getter defined here. Let's improve the getter to allow the elementLast computed property to use it.

Since the last element of the list does not have a fixed index, suppose that if we indicate -1 as index in the getElementByIndex() getter it means to find the last element of the list.

Use the getElementByIndex() getter to retrieve the last element

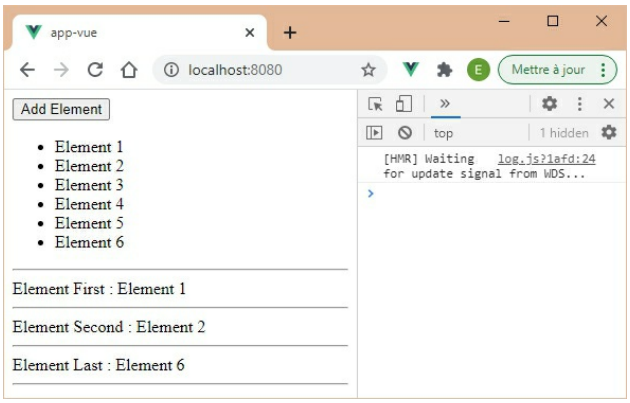
```
<template>
  <div>
    <button v-on:click="add">Add Element</button>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
    <hr>
    Element First : {{elementFirst}}
    <hr>
    Element Second : {{elementSecond}}
    <hr>
    Element Last : {{elementLast}}
    <hr>
  </div>
</template>

<script>
```

```
import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  },
  mutations : {
    add_element(state, payload) {
      var element = payload.element;
      state.elements.push(element);
    }
  },
  getters : {
    elementFirst(state) {
      return state.elements[0];
    },
    elementSecond(state) {
      return state.elements[1];
    },
    getElementByIndex(state) {
      return function(index) {
        if (index != -1) return state.elements[index];
        else return state.elements[state.elements.length - 1];
      }
    }
  }
});
```

```
export default {
  store,
  computed : {
    elements() {
      return this.$store.state.elements;
    },
    elementFirst() {
      //return this.$store.state.elements[0];
      //return this.$store.getters.elementFirst;
      return this.$store.getters.getElementByIndex(0);
    },
    elementSecond() {
      //return this.$store.state.elements[1];
      //return this.$store.getters.elementSecond;
      return this.$store.getters.getElementByIndex(1);
    },
    elementLast() {
      //return this.$store.state.elements[this.$store.state.elements.length-1];
      return this.$store.getters.getElementByIndex(-1);
    },
  },
  methods : {
    add() {
      var element = "Element " + (this.$store.state.elements.length + 1);
      this.$store.commit("add_element", {element:element});
    }
  }
}

</script>

<style>

</style>
```

All calculated properties of the App component now use the getElementByIndex() getter defined in the store

object.



List management as components using Vuex

We have in chapter 6 ("*Using the Vue CLI utility*") how to manage a list of elements in the form of components, all architectured in the form of .vue modules.

The list of elements was centralized in the reactive variable elements defined in the main Vue object of the application. This required us to use an event bus (eventBus variable) which allowed to send the add, remove, and modify events to manage the list centrally (and also to manage the contextual menu displayed when clicking on a list item).

Rather than using an event bus, it is better to use Vuex. The global state of the application would then be:

- The elements property which contains the

elements of the list, in the form of { text } objects, the text property corresponding to the text of the displayed element.

- The contextmenu property that contains the component associated with the displayed context menu (null if no menu is displayed).

> The src/assets/eventBus.js file is no longer necessary here, since we no longer use the event bus.

In addition, the src/main.js file is also simplified, and reset to its initial state:

src/main.js file

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

new Vue({
  render: h => h(App)
}).$mount('#app')
```

The src/App.vue file now becomes:

src/App.vue file

```
<template>
  <div style="position:relative; height:100%;" v-on:click="click">
    <button v-on:click="add">Add Element</button>
    <Elements v-bind:elements="elements" />
  </div>
</template>

<script>
```

```
import Elements from "@/components/Elements.vue";
//import eventBus from "@/assets/eventBus.js";

import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

var store = new Vuex.Store({
  state : {
    elements : [
      { text : "Element 1" },
      { text : "Element 2" },
      { text : "Element 3" },
      { text : "Element 4" },
      { text : "Element 5" }
    ],
    contextmenu : null
  },
  mutations : {
    add_element(state) {
      var text = "Element " + (state.elements.length + 1);
      state.elements.push({ text });
    },
    remove_element(state, {index}) {
      state.elements = state.elements.filter(function(element, i) {
        if (i == index) return false;
        else return true;
      });
    },
    modify_element(state, {index, value}) {
      state.elements = state.elements.map(function(element, i) {
        if (i == index) return { text : value };
        else return element;
      });
    },
    set_contextmenu(state, {contextmenu}) {
```

```
      state.contextmenu = contextmenu;
    }
  },
});

export default {
  store,
  computed : {
    elements() {
fv      return this.$store.state.elements;
    }
  },
  methods : {
    add() {
      //eventBus.$emit("add");
      this.$store.commit("add_element");
    },
    click() {
      //eventBus.$emit("hidecontextmenu");
      if (this.$store.state.contextmenu)
        this.$store.state.contextmenu.$parent.displayContextMenu = false;
      this.$store.commit("set_contextmenu", { contextmenu : null });
    }
  },
  components: {
    Elements
  }
}
</script>

<style>

</style>
```

Of course, the store is created in the usual way, including the state and the corresponding mutations:

- The add_element() mutation allows you to insert a new element in the list,
- The remove_element() mutation removes the indicated element from the list,
- The modify_element() mutation allows to modify the indicated element,
- The set_contextmenu() mutation allows you to store the displayed context menu (null if none).

The src/components/Elements.vue file is not modified:

src/components/Elements.vue file

```
<template>
  <div>
    <ul v-if="elements.length">
       <Element v-for="(element, index) in elements" style="margin-
top:10px;"
             v-bind:key="index"
             v-bind:index="index"
             v-bind:text="element.text">
      </Element>
    </ul>
    <div v-else><br>Empty List</div>
  </div>
</template>
<script>
import Element from "@/components/Element.vue";
export default {
  props : [
    "elements"
  ],
  data() {
```

```
    return {
    }
  },
  components: {
    Element
  }
}
</script>
<style>
</style>
```

The src/components/Element.vue file is modified to use Vuex state rather than the event bus:

src/components/Element.vue file

```
<template>
  <li style="margin-top:10px;">
     <input v-if="modifyOn" type="text"
          v-on:keydown.enter="modify"
          v-on:blur="modify"
             v-bind:value="text"
          ref="input">
    <div v-else>
       <span v-on:dblclick="modifyOn=true"
          v-on:click.stop="displayCtxMenu" style="cursor:pointer;">
       {{text}}
      </span>
      <ContextMenu v-show="displayContextMenu" v-bind:x="x" v-
bind:y="y" />
      <button style="margin-left:10px; font-size:11px;"
            v-on:click="remove">
        Remove
      </button>
    </div>
  </li>
```

```
</template>
<script>
import ContextMenu from "@/components/ContextMenu.vue";
//import eventBus from "@/assets/eventBus.js";
export default {
  props : [
    "text",
    "index"
  ],
  data() {
    return {
      modifyOn : false,
      displayContextMenu : false,
        x : 0,
      y : 0
    }
  },
  created() {
    this.$on("modify", function() {
      this.modifyOn = true;
      this.displayContextMenu = false;
    });
    this.$on("remove", function() {
      this.remove();
      this.displayContextMenu = false;
    });
  },
  updated() {
    if (this.$refs.input) this.$refs.input.focus();
  },
  methods : {
    remove() {
      //eventBus.$emit("remove", this.index);
      this.$store.commit("remove_element", { index : this.index });
    },
    modify(event) {
```

```
      this.modifyOn = false;
      var newText = event.target.value;
      //eventBus.$emit("modify", this.index, newText);
      this.$store.commit("modify_element", { index : this.index, value :
newText });
    },
    displayCtxMenu(event) {
      this.x = event.clientX;
      this.y = event.clientY;
      this.displayContextMenu = true;
      this.$parent.$children.forEach((child) => {
          if(child.index != this.index)
         child.displayContextMenu = false;
      });
      //eventBus.$emit("displaycontextmenu", this.$children[0]);
      this.$store.commit("set_contextmenu", { contextmenu :
this.$children[0] });
    }
  },
  components: {
    ContextMenu
  }
}
</script>
<style>
</style>
```

Remember that the store having been specified in the App component (store property), it is accessible in all child components via this.$store.

The ContextMenu and ItemMenu components do not use the store and are therefore not modified:

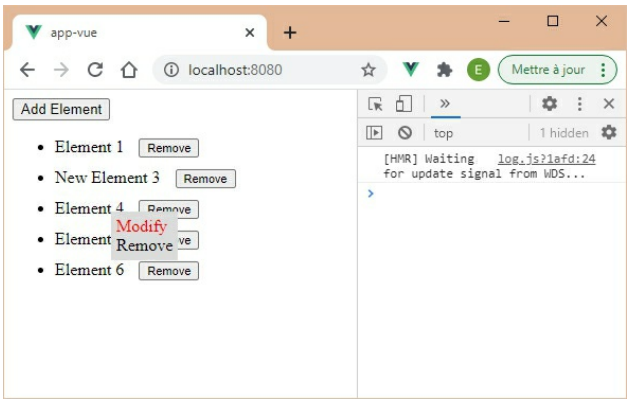
src/components/ContextMenu.vue file

```
<template>
  <div v-bind:style="{position:'absolute', top:y+'px', left:x+'px',
                  backgroundColor:'gainsboro'}">
   <ul style="margin-left:0px; padding:5px; list-style-type:none;
cursor:pointer;">
     <ItemMenu v-on:click="modify">Modify</ItemMenu>
     <ItemMenu v-on:click="remove">Remove</ItemMenu>
   </ul>
 </div>
</template>
<script>
import ItemMenu from "@/components/ItemMenu.vue";
export default {
 props : [
   "x",
   "y"
 ],
 data() {
   return {
   }
 },
 methods : {
   modify() {
     this.$parent.$emit("modify");
   },
   remove() {
     this.$parent.$emit("remove");
   }
 },
 components: {
   ItemMenu
 }
}
</script>
<style>
</style>
```

src/components/ItemMenu.vue file

```
<template>
  <li v-on:click="$emit('click')"
       v-on:mouseover="mouseover" v-on:mouseout="mouseout"
      v-bind:style="{color:color}">
    <slot/>
  </li>
</template>
<script>
export default {
  data() {
    return {
      color:""
    }
  },
  methods : {
    mouseover() {
      this.color = "red";
    },
    mouseout() {
      this.color = "";
    }
  },
  components: {
  }
}
</script>
<style>
</style>
```

It remains to check that the operation is identical:


app-vue
localhost:8080
Mettre à jour
Add Element
Element 1
Remove
New Element 3
Remove
Element 4
Element
Modify
Remove
Element 6
Remove
top
1 hidden
[HMR] Waiting for update signal from WDS...
log.js?1afd:24

9 – USING VUE ROUTER

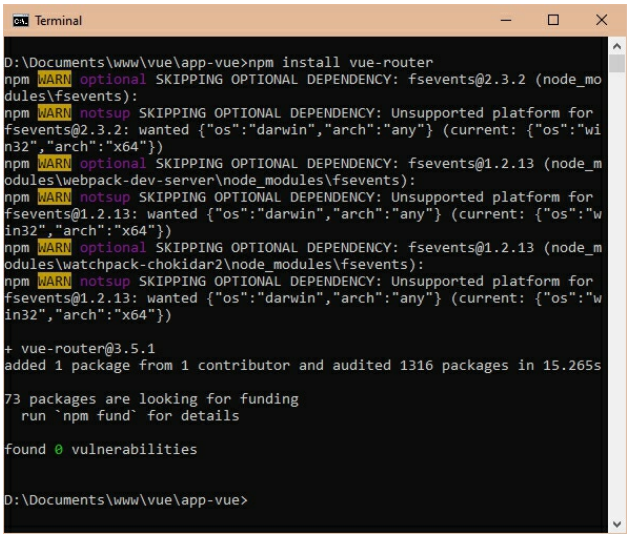
Vue Router is a route manager associated with Vue. It allows to have different URLs in an application on a single HTML page (SPA, Single Page Application).

Install Vue Router

Vue Router is installed using the npm install vue-router command. The associated module is installed directly in the node_modules directory of the project (here in the app-vue directory).

Install Vue Router

```
npm install vue-router
```



> Remember to restart the server with npm run serve in case you stopped it to install Vue Router.

Vue Router principle

Vue Router is used quite similar to Vuex. We include the "vue-router" module by the import VueRouter from "vue-router" instruction, then we create a router object of class VueRouter.

Let's use Vue Router in an App component.

Minimum use of Vue Router (src/App.vue file)

```
<template>
  <div>
    App component using Vue Router
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
  ]
});

console.log(router);

export default {
  router   // router : router
}
```

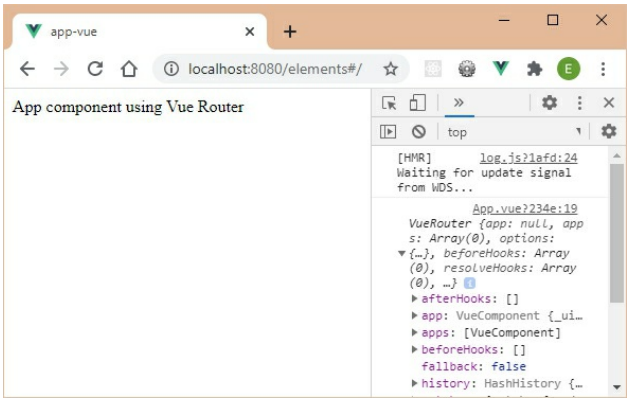
```
</script>

<style>

</style>
```

The router object is passed to the App component, which allows it to know the routes on which it should act.

> As with the store object in Vuex, the router object is passed into all components included in the App component. All the components of the application will therefore have access to Vue Router thanks to this.

Routes are specified in the routes property when creating the router object. For the moment no route is defined (the routes table associated with the routes is empty []).

Let us indicate the URL http://localhost:8080 in the browser:



The router object is displayed in the console. We will study its properties in the following sections.

Note that the URL displayed in the address bar of the browser is slightly modified, because it incorporates "/#/" at the end, meaning that the router is taken into account.

Define a route

Routes are defined in the routes section of the router object. Each route is defined as an object { path, component }:

- The path property represents a path (ie some form of URL, for example "/" or "/home"). It is this path that will trigger the route when it is indicated in the address bar of the browser.
- The component property is the name of the Vue component to display when this route is enabled.

Let's define the route "/elements" which allows to display the list of the elements of a list. This list of elements is managed by an Elements component. We will therefore have the route { path: "/elements", component: Elements } defined in the routes section of the router object.

Define a route /elements

```
<template>
  <div>
    App component using Vue Router
  </div>
```

```
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements", component : Elements }
  ]
});

export default {
  router
}

</script>

<style>

</style>
```

The route /elements is now defined, it remains to define the Elements component that will be displayed.

Define the component displayed by the road

The Elements component here is the one indicated in the definition of the /elements route. It must therefore be defined:

- Either minimally (i.e. only with a template),
- Either in the traditional way by using a reactive

variable elements.

We describe these two forms below. Both forms work with the examples that follow.

Elements component (src/components/Elements.vue file) minimally described

```
<template>
  <div>
    <ul>
      <li>Element 1</li>
      <li>Element 2</li>
      <li>Element 3</li>
      <li>Element 4</li>
      <li>Element 5</li>
    </ul>
  </div>
</template>
```

Elements component described with a reactive variable elements

```
<template>
  <div>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
  </div>
</template>
<script>
export default {
  data() {
    return {
      elements : [
```

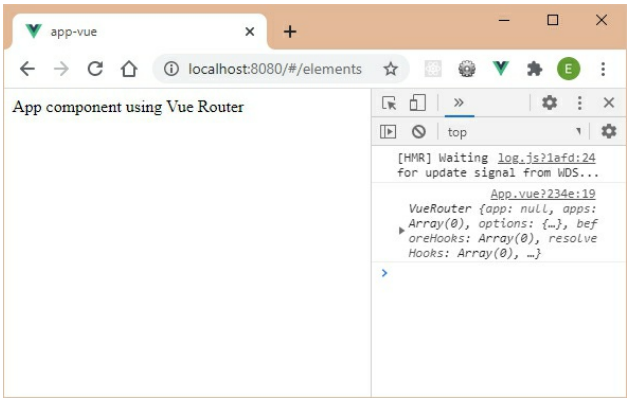
```
      "Element 1",
      "Element 2",
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  }
 }
}
</script>
```

It now remains to display the route /elements in the browser.

Display the route in the browser

To display the route /elements in the browser, just add this route at the end of the displayed URL. So the new URL is now http://localhost:8080/#/elements.

Let's type this URL into the browser:



The URL is displayed in the browser's address bar, but the page displayed is not modified ...

In fact, this is normal, because Vue Router requires

you to indicate the location of the page where the component associated with the route should be rendered. If this location is not specified in the page, the component associated with the route cannot be displayed (which is the case here).

The location must be indicated in the page by means of the <router-view> tag, which is written in the form <router-view /> or <router-view></router-view>. The component will be rendered in this tag.

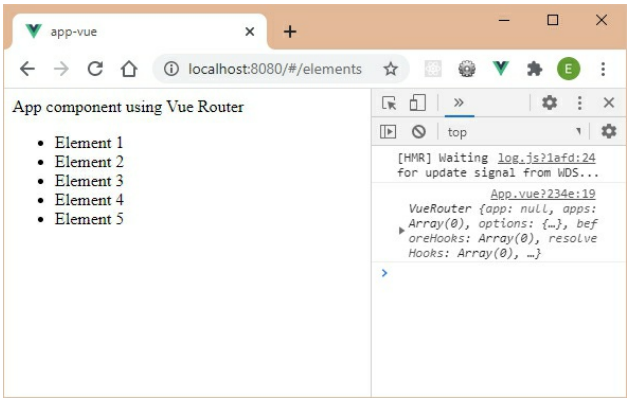
Let's add this tag in the template of the App component:

App component including the <router-view> tag

```
<template>
  <div>
    App component using Vue Router
    <br>
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements", component : Elements }
```

```
  ]
});

export default {
  router
}

</script>

<style>

</style>
```

The Elements component can now be displayed in the HTML page:



Define a new route

Let's define a new /home route, which will activate the Home component, defined below.

Home component (src/components/Home.vue file)

```
<template>
  <div>
    <h1>This is Home component</h1>
  </div>
</template>
```

The definition of the new route is carried out following the previous one in the routes section of the router object:

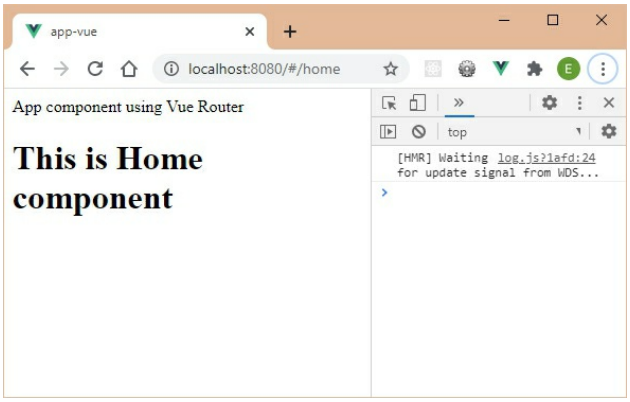
Adding the /home route (src/App.vue file)

```
<template>
  <div>
    App component using Vue Router
    <br>
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements", component : Elements },
    { path : "/home", component : Home }
  ]
});

export default {
  router
}

</script>

<style>
```

```
</style>
```

> When several routes are indicated in the routes section, the first route registered in the list which corresponds to the requested URL will be the one chosen by Vue Router, even if several routes are possible for this URL.

The /home route is displayed by typing the URL http://localhost:8080/#/home in the browser's address bar.



> Note that the Back button of the browser allows you to change the displayed URL, and therefore to return to the previous URL which was http://localhost:8080/#/elements. It is in the interest of using a route system such as Vue Router, because in addition to the URLs which clearly indicate the action performed, you can return to the previous action without leaving the application.

In fact, in a Single Page Application (SPA) application, the browser back button normally causes the

application to exit, due to the application being on a single HTML page. But using Vue Router we just go back to the URL of the previous route.

Show links on the page to navigate between routes

For the moment we indicate the desired URL by typing it in the address bar of the browser. But it is rare to do it this way, because instead we use links in the page that allow navigation.

Vue Router offers a new beacon allowing this navigation. This is the <router-link> tag. The to attribute of the tag is used to indicate the path of the route to which we will be directed after clicking on the link.

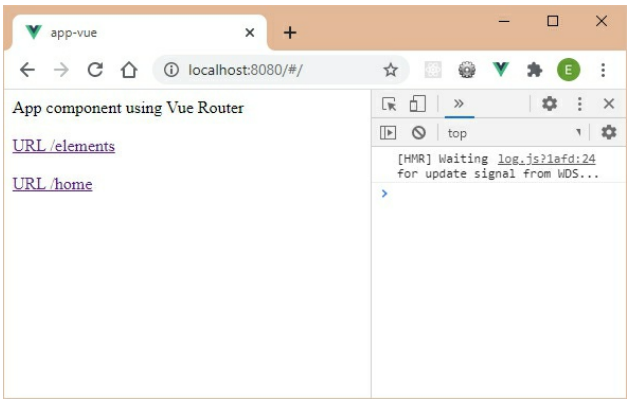
Let's insert this tag in the template of the App component to navigate to the routes /elements and /home defined previously.

Use the <router-link> tag

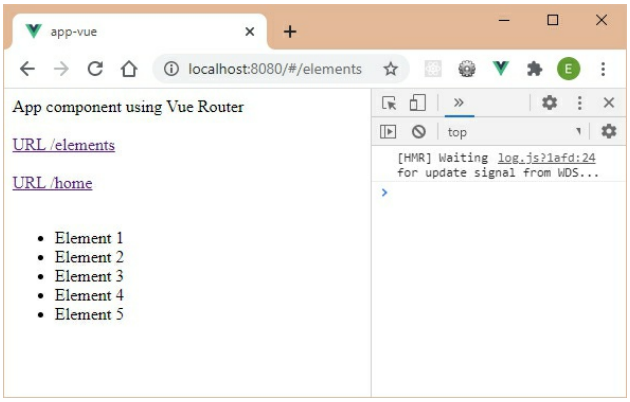
```
<template>
  <div>
    App component using Vue Router
    <br><br>

     <router-link to="/elements">URL /elements</router-link>
    <br><br>
```

```
    <router-link to="/home">URL /home</router-link>
   <br><br>


    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements", component : Elements },
    { path : "/home", component : Home }
  ]
});

export default {
  router
}

</script>

<style>

</style>
```

When displaying the main URL of the page http://localhost:8080/#/ :

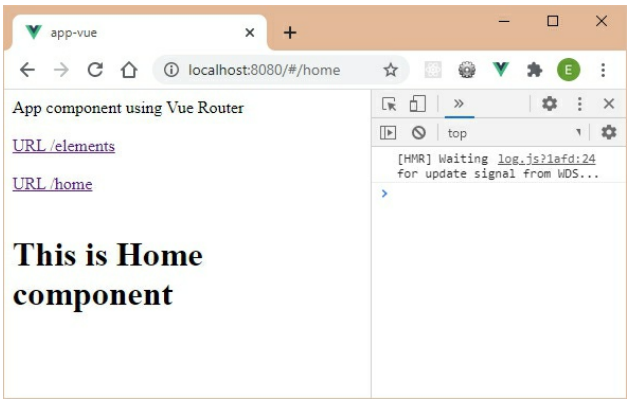


The two links corresponding to the <router-link> tags are displayed. Let's click on the first link displaying the route /elements:



The Elements component is displayed in the <router-view> part of the App component.

Let's click on the second link of the /home page:



It is now the Home component that is displayed in the <router-view> part of the App component.

Style the link corresponding to the displayed route

When the URL of the displayed page matches a URL used by <router-link>, the router-link-exact-active CSS class is added by Vue Router in the corresponding link. If we style this CSS class, we can visualize the link selected in the page.

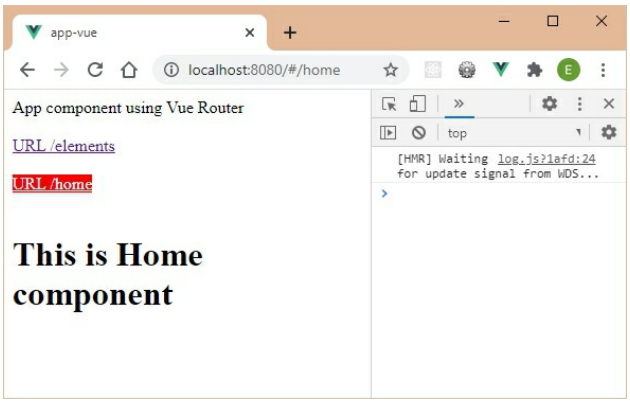
Style the link selected in the page with a red background

```
<template>
  <div>
    App component using Vue Router
    <br><br>


     <router-link to="/elements">URL /elements</router-link>
    <br><br>


     <router-link to="/home">URL /home</router-link>
    <br><br>
```

```
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements", component : Elements },
    { path : "/home", component : Home }
  ]
});

export default {
  router
}

</script>

<style>

.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```

All you have to do is indicate the router-link-exact-active CSS class in the <style> tag. And depending on

the link you click on, the link is displayed according to the style applied in the CSS class.



In this section, we have studied the basics of Vue Router. Let us now examine the more specific concepts of this library.

Use parameters in routes

Let's start with the possibility of specifying parameters in the routes. This means the ability to pass parameters to components that use the route.

Add the parameter index in the routes

Consider the following routes:

- The route /elements allows to display all the elements of the list,
- The route /elements/2 allows to display only the list element having this index (here 2, therefore "Element 1").

In the case of the last route, the value of the index (here 2) is a parameter which can vary. To allow this variation, we indicate in the path of the route "/elements/:index" (when declaring it in the routes section of the router object).

The notation :index represents an index name parameter that can be used in the component that processes the route.

Let's add these routes to the router, and display them in the App component template:

src/App.vue file

```
<template>
  <div>
    <router-link to="/elements">URL /elements</router-link>
    <br><br>
    <router-link to="/elements/2">URL /elements/2</router-link>
    <br><br>
    <router-link to="/home">URL /home</router-link>
    <br><br>
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";

Vue.use(VueRouter);
```

```
var router = new VueRouter({
  routes : [
    { path : "/elements", component : Elements },
    { path : "/elements/:index", component : Elements },
    { path : "/home", component : Home }
  ],
});

export default {
  router
}

</script>

<style>

.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```

The second URL displayed in the template is /elements/2. This means that the value 2 will be passed to the Elements component in the index parameter of the route.

Let's modify the Elements component to handle this index parameter. The index passed in the URL is retrieved from the component, in the variable this.$route.params.index.

The this.$route variable is available in all components that have access to the router (ie all components from the main App component).

> We use this.$route in the JavaScript part of a component, and $route in the template part.

src/components/Elements.vue file

```
<template>
  <div>
    <ul>
      <li v-for="(element, i) in elements" v-bind:key="i"
        v-show="index==undefined || index==i">{{element}}</li>
    </ul>
  </div>
</template>
<script>
export default {
  data() {
    return {
      elements : [
        "Element 1",
        "Element 2",
        "Element 3",
        "Element 4",
        "Element 5"
      ]
    }
  },
  computed : {
    index() {
      return this.$route.params.index;
    }
  }
}
</script>
```

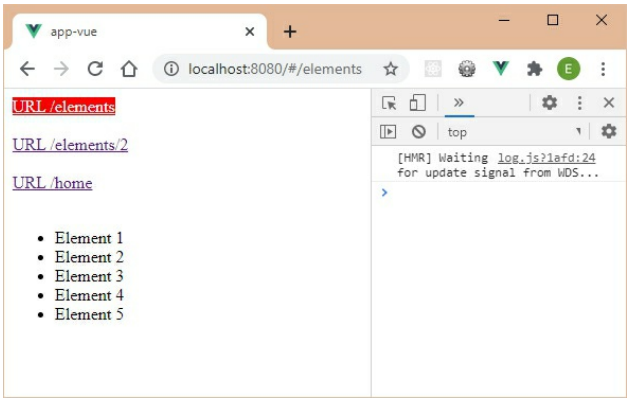
We use a calculated property index which returns the index transmitted in the URL, in order to simplify

writing in the template. However, if the URL is used in the form /elements (without specifying an index), the value of the index in this.$route.params.index will be undefined.

We therefore test the value of the index transmitted:

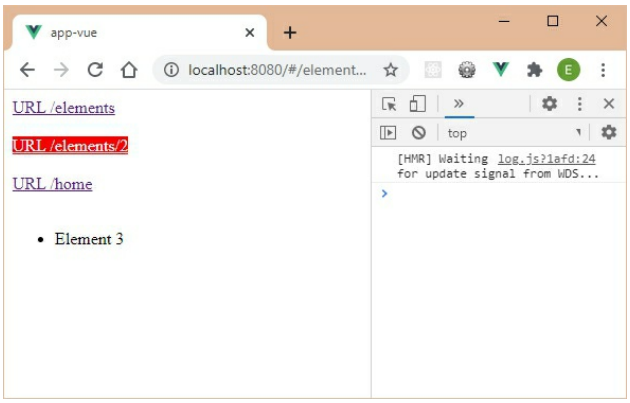
- If index is undefined, it means that it is not transmitted in the URL (which is of the form /elements), so we must display each of the elements of the list.
- Otherwise (the index is indicated in the URL, which is of the form /elements/2), we check if the value of the index corresponds to that of the list element that we display (variable i in the template). If the two values are the same, then this list item is displayed (hiding all the others).

By clicking on the first link /elements, the whole list is displayed:



By clicking on the second link /elements/2, only the list

item with index 2 is displayed:



Optional index parameter

We have written two routes in the route list:

- The route /elements,
- The route /elements/:index.

These two routes activate the same Elements component.

We could just keep the route /elements/:index in the list, indicating that the index parameter is optional. Just add the sign "?" following the parameter. This means that this parameter can be present 0 or 1 time (it is therefore optional).

Optional index parameter

```
var router = new VueRouter({
  routes : [
//    { path : "/elements", component : Elements },
    { path : "/elements/:index?", component : Elements },
    { path : "/home", component : Home }
```

```
  ],
});
```

The route /elements having no more use, is put in comments.

Index parameter in integer form

The index parameter must be written as an integer, otherwise the corresponding element will not exist in the list. But since you can enter the index directly in the URL, you have to watch out for any typing errors.

We can indicate in a route the form of a parameter, for example that it is an integer. As for regular expressions, we write (\\d+) after the name of the parameter in the path.

Index parameter in integer form

```
var router = new VueRouter({
  routes : [
//    { path : "/elements", component : Elements },
    { path : "/elements/:index(\\d+)?", component : Elements },
    { path : "/home", component : Home }
  ],
});
```

The route will only be activated if the index parameter is specified as an integer.

> You can find more information on the possibilities of regular expressions in writing routes at https://github.com/pillarjs/path-to-regexp, because

> Vue Router uses this library to analyze the route.

Use parameters as props

The previous example shows how to pass parameters in routes. These parameters are retrieved from the component using this.$route.params which is an object containing the parameters passed.

A small drawback of this mechanism is that to access the value of a parameter, you have to use this object, which sometimes takes a bit long to write. It would be easier to write index rather than this.$route.params.index.

Vue Router allows this, by indicating that the parameter is passed in props (in addition to being passed in this.$route.params). All you have to do is indicate the props property set to true in the route description (routes section of the router object). This props (here index) will be integrated into the props of the component and accessible like the other props of the component (on condition of course to indicate the props in the props of the component, in the props section).

Let's add the index props to the Elements component props, and indicate that the route uses props:

Use the index props to pass the element's index(src/App.vue

file)

```
var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home", component : Home }
  ],
});
```

> The props property must be set to true in the route for the route parameters to be passed in the props. If the props property is set to false, no parameters are passed in the props.

It is also necessary to modify the Elements component so that it takes into account this new props.

src/components/Elements.vue file

```
<template>
  <div>
    <ul>
      <li v-for="(element, i) in elements" v-bind:key="i"
          v-show="index==undefined || index==i">{{element}}</li>
    </ul>
  </div>
</template>
<script>
export default {
  data() {
    return {
      elements : [
        "Element 1",
        "Element 2",
        "Element 3",
```

```
        "Element 4",
        "Element 5"
      ]
    }
  },
//  computed : {
//    index() {
//      return this.$route.params.index;
//    }
//  },
  props : [
    "index"
  ]
}
</script>
```

The index props allows you to no longer use the computed property of the same name (hence the comment in this section).

Indicate that a route is unknown

In the event that the URL specified in the browser does not match any of the routes indicated in the routes section, an error should be displayed. To do this, we indicate last (in the list of routes) the route to choose when all the previous routes have failed (remember that Vue Router chooses the first route whose path is similar to the browser URL).

Let us indicate in the path of the last route a path which we are sure will be that of the displayed URL. For that,

we indicate the path worth "*", that means any characters, even none.

We can create a special component that will be used to display the error. You can also indicate the template in the definition of the route, as below.

Indicate a path for an unknown URL

```
var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home", component : Home },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1></div>" } }
  ],
});
```

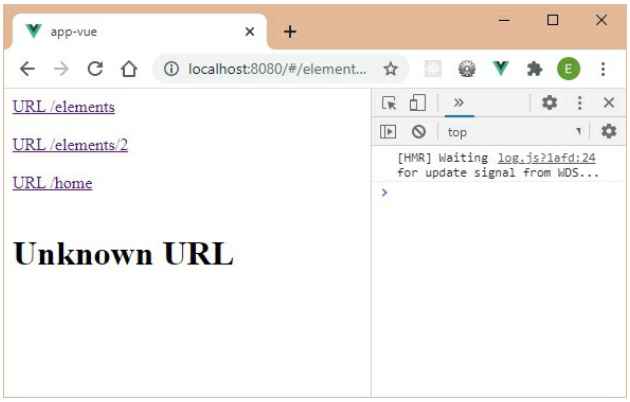
> If the component template is defined in the route itself, you must configure the vue.config.js file as follows (see below), then restart the server. This configuration of the vue.config.js file is unnecessary if the component template is defined in an external component (such as the Home and Elements components).

vue.config.js file (server's root)

```
module.exports = {
  runtimeCompiler: true
}
```

> Once this file has been created or modified, restarting the server is mandatory.

When the URL does not match any of the authorized URLs (except the last one in the list which captures them all):



Named routes

Rather than using a route via its path in the template (e.g. /elements/2), it can sometimes be easier to give a name (name attribute) to the route and use it by name rather than by its path in the template.

The use in the template is made via the <router-link> tag by passing it an attribute :to (and not to as before), whose value is a JSON object.

For example, to simulate the path /elements/3, we write:

Simulate URL /elements/3

```
<router-link :to="{name:'index', params:{index:3}}">URL /elements/3</router-link>
```

Let's show on an example how to use named routes.

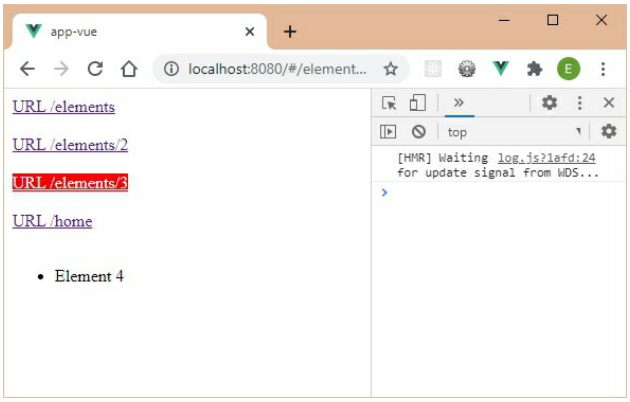
We assign a name to the /elements/:index route, and we use it with that name in the template.

Create and use a named route (src/App.vue file)

```
<template>
  <div>
    <router-link to="/elements">URL /elements</router-link>
    <br><br>
    <router-link to="/elements/2">URL /elements/2</router-link>
    <br><br>
    <router-link :to="{name:'index', params:{index:3}}">URL /elements/3</router-link>
    <br><br>
    <router-link to="/home">URL /home</router-link>
    <br><br>
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true, name : "index" },
    { path : "/home", component : Home }
  ],
});
```

```
export default {
  router
}

</script>

<style>

.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```

The /elements/2 and /elements/3 routes work identically, but the /elements/2 route uses the classic system with the path, while the /elements/3 route uses the named route system.



Named views

In all of the previous examples, we have assumed that displaying a route causes the display of a single component, which is registered in the <router-view>

tag of the template.

> We therefore had a single component to display, in a single place on the page represented by the <router-view> tag. This location is called a view.

Now imagine that displaying a route causes several components to be displayed in different locations on the page. This is the principle of named views.

Display a component in multiple places on the page

Let's start with the simplest. Let's display the components in the App component template at three successive locations on the page.

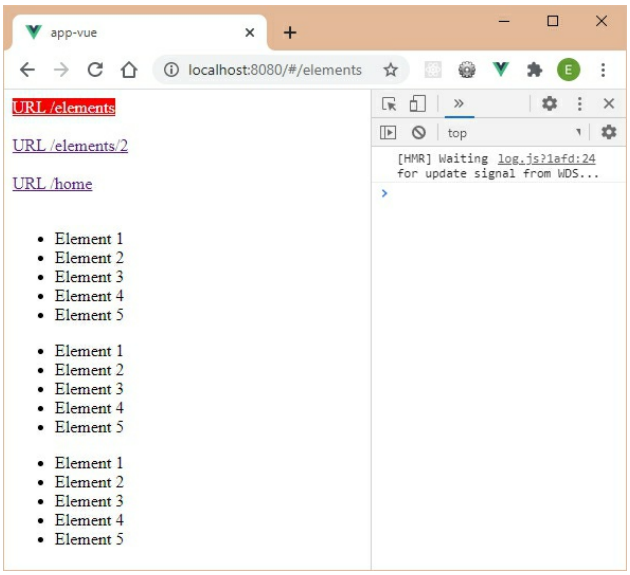
To do this, we duplicate the <router-view> tag three times in the template, without changing anything else in the App component.

<router-view> tag duplicated three times in the template(src/App.vue file)

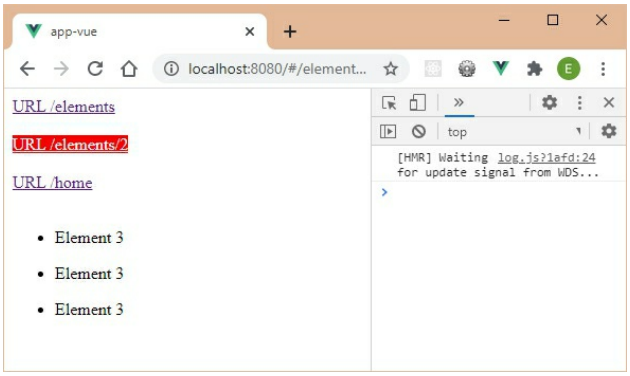
```
<template>
  <div>
    <router-link to="/elements">URL /elements</router-link>
    <br><br>
    <router-link to="/elements/2">URL /elements/2</router-link>
    <br><br>
    <router-link to="/home">URL /home</router-link>
    <br><br>
    <router-view/>
    <router-view/>
```

```
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home", component : Home },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1>
</div>" } }
  ]
});

export default {
  router
}

</script>

<style>

.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```

Using the /elements route, the global list of elements is displayed three times:



Using the route /elements/2, element with index 2 is also displayed three times:



It would be the same with the other routes. In fact, each component of each route is associated here with the view named "default", which is the name of the view associated by default with a <router-view> tag.

Default view

The <router-view> tag can also be written using its name attribute in the template, specifying the value

"default":

Use the name attribute in the <router-view> tag

```
<router-view name="default"/>
```

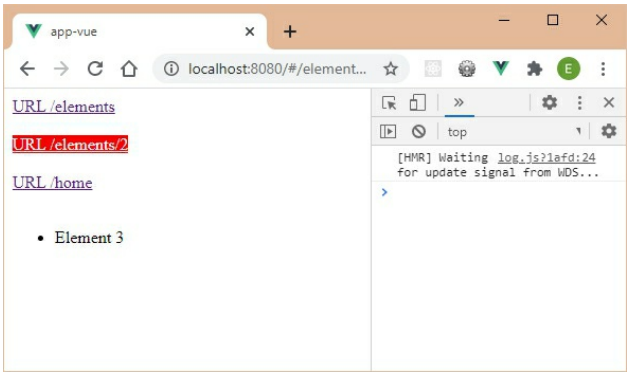
If the last two views of the template are named "view1" and "view2", the components associated with the routes will no longer be displayed because they are displayed (by default) in the default view (the one whose name is empty or which is "default").

Name the last two views "view1" and "view2"

```
<template>
  <div>
    <router-link to="/elements">URL /elements</router-link>
    <br><br>
    <router-link to="/elements/2">URL /elements/2</router-link>
    <br><br>
    <router-link to="/home">URL /home</router-link>
    <br><br>
    <router-view name="default"/>
    <router-view name="view1"/>
    <router-view name="view2"/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";

Vue.use(VueRouter);
```

```
var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home", component : Home },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1>
</div>" } }
  ]
});

export default {
  router
}

</script>

<style>

.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```

The result is now different, because the views "view1" and "view2" are no longer affected by the routes indicated (only the default view displays its component).



We therefore see that we have a mechanism allowing to display, according to the indicated route, one component or another in the page. Let us now study this mechanism.

Assign a component to a view

When indicating a route in the form:

Route assigned to a component

```
{ path : "/elements/:index(\\d+)?", component : Elements, props : true }
```

It actually looks like this:

Route assigned to a component

```
{ path : "/elements/:index(\\d+)?", components : { default : Elements },
props : {default:true} }
```

> Note in this case the use of the components property instead of component, as a path can use multiple components (actually multiple views).

Which can be written on several lines:

Route assigned to a component

```
{ path : "/elements/:index(\\d+)?",
  components : {
    default : Elements
  },
  props : {
    default : true
  }
}
```

We indicate for a route the views that will be

displayed, and for each view the component that corresponds to it. In our example, for the indicated route, we have a single view named "default", in which we display the Elements component.

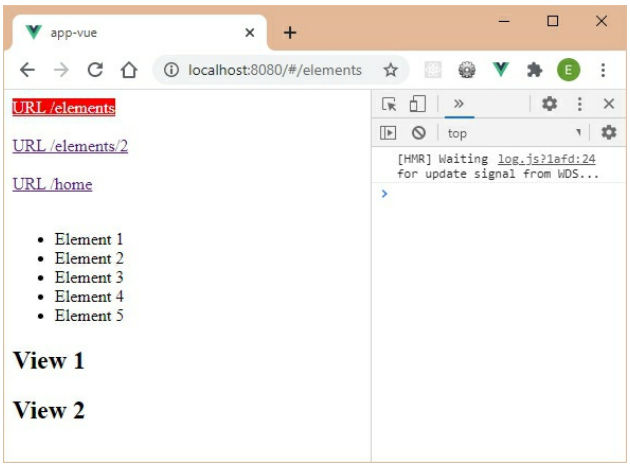
But we could use additional views (in which other components would be displayed). So if we add the views "view1" and "view2" in the route:

Added views "view1" and "view2" to the /elements/:index route

```
<template>
  <div>
    <router-link to="/elements">URL /elements</router-link>
    <br><br>
    <router-link to="/elements/2">URL /elements/2</router-link>
    <br><br>
    <router-link to="/home">URL /home</router-link>
    <br><br>
    <router-view name="default"/>
    <router-view name="view1"/>
    <router-view name="view2"/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";

Vue.use(VueRouter);
```

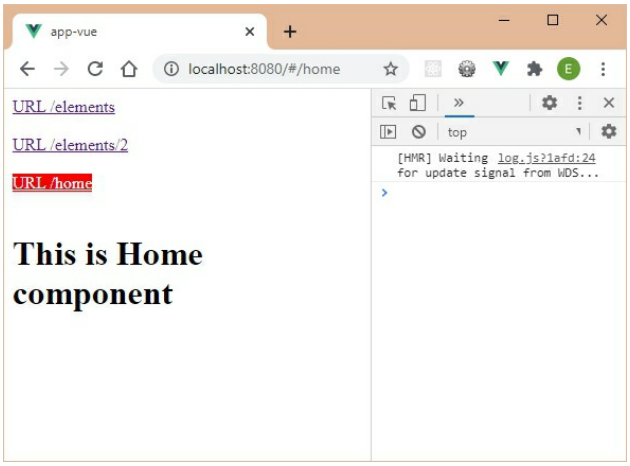
```
var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?",
      components : {
        default : Elements,
        view1 : { template : "<h2> View 1 </h2>" },
        view2 : { template : "<h2> View 2 </h2>" }
      },
      props : {
        default : true
      }
    },
    { path : "/home", component : Home },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1>
</div>" } }
  ]
});

export default {
  router
}

</script>

<style>

.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```

When displaying the route /elements, we get:



All three views are displayed.

While for the /home route, only the Home component of the default view is displayed:



We therefore have a simple and powerful mechanism to display specific views according to the URLs used.

Nested routes and components

So far, the <router-view> tag has only been used in the main App component of the application. But this <router-view> tag can be used inside other components

that will be displayed through the App component.

Consider the Home component, which is displayed when the URL is /home. We also want:

- If the URL is /home/connect, an internal component is displayed (in the Home component) which suggests entering the user's email address,
- If the URL is /home/register, another internal component (in the Home component) is displayed which offers the entry of the user's email and password.

The Home component is modified to integrate the display of these new components. We insert the <router-view> tag in the Home component to indicate that a new component will be included when selecting this path.

src/components/Home.vue file

```
<template>
  <div>
    <h1>This is Home component</h1>
    <router-view />
  </div>
</template>
```

The new routes are called child roads, and are grouped together in the children property of the route that defines the main /home route.

Define child routes with the children property (src/App.vue file)

```
<template>
  <div>
    <router-link to="/elements">URL /elements</router-link>
    <br><br>
    <router-link to="/elements/2">URL /elements/2</router-link>
    <br><br>
    <router-link to="/home">URL /home</router-link>
    <br><br>
    <router-link to="/home/connect">URL /home/connect</router-link>
    <br><br>
    <router-link to="/home/register">URL /home/register</router-link>
    <br><br>
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";
import Connect from "@/components/Connect.vue";
import Register from "@/components/Register.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home",
      component : Home,
      children : [
        { path : "connect", component : Connect },
```

```
      { path : "register", component : Register }
    ]
   },
   { path : "*", component : { template : "<div><h1>Unknown URL</h1>
</div>" } }
 ]
});

export default {
 router
}

</script>

<style>

.router-link-exact-active {
 background-color:red;
 color: white;
}

</style>
```

The children property of a route is an array of internal routes used. Note that the path indicated does not include the "/" at the beginning, because this would be interpreted as being the root of the site (it would then be necessary to introduce the path /connect or /register to reach the component).

The Connect and Register components are:

src/components/Connect.vue file

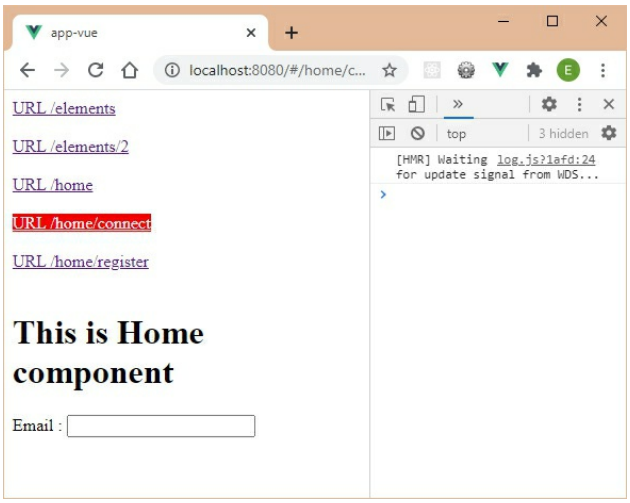
```
<template>
 <div>
  Email : <input type="email">
```

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

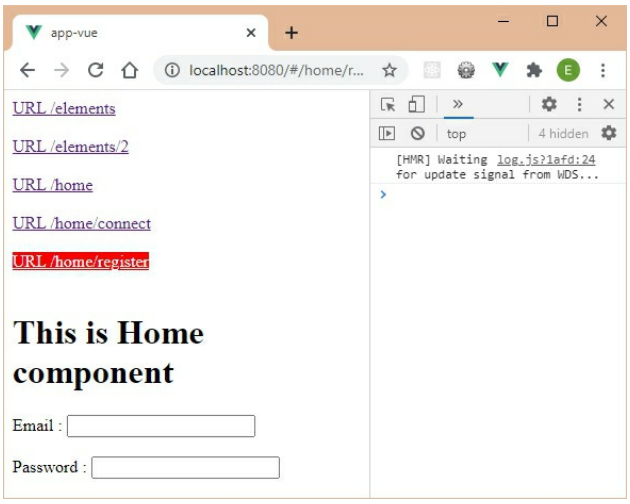
src/components/Register.vue file

```
<template>
  <div>
    Email : <input type="email"><br><br>
    Password : <input type="password">
  </div>
</template>
```

Using the /home/connect path, we display the Home component including the Connect component:



While using the path /home/register, we display the Home component including the Register component:



Use redirects

A redirection allows you to display a path other than the one chosen. We are redirected to the new path.

For this, we use the redirect property in the route, indicating as value the path to which we redirect this route.

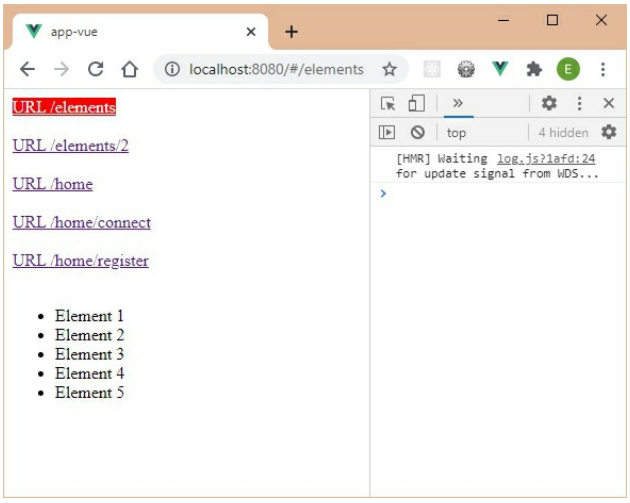
Let's indicate a redirect from /home/connect to /elements.

Redirection from /home/connect to /elements

```
var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home",
      component : Home,
      children : [
        { path : "connect", component : Connect, redirect : "/elements" },
        { path : "register", component : Register }
      ]
```

```
    },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1>
</div>" } }
  ]
});
```

Let's click on the path /home/connect in the displayed page. The URL automatically changes to display the path /elements.



Use aliases

An alias is used to indicate a new path for a route. The route can therefore be accessed:

- By its path indicated in the path attribute,
- Or by its other path indicated in the alias attribute.

Let's give the /home/connect route a new alias. Let's simplify it by simply specifying /connect.

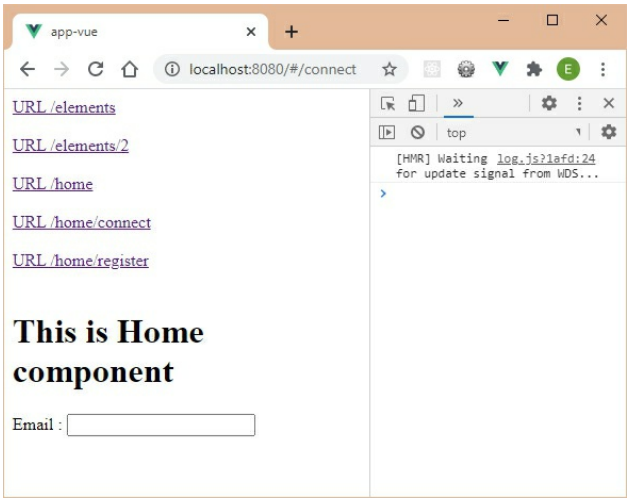
The new route is now, indicating the alias /connect to

the /home/connect route:

Indicate the alias /connect to the /home/connect route (src/App.vue file)

```
var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home",
      component : Home,
      children : [
        { path : "connect", component : Connect, alias : "/connect" },
        { path : "register", component : Register }
      ]
    },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1></div>" } }
  ]
});
```

Let's indicate the route /connect in the navigator. Although the route displayed in the address bar is /connect, it is the /home/connect route that is displayed in the window.



An alias will therefore make it possible to simplify certain URLs by making it possible to indicate a path simpler than the normal path.

Using the Vue Router API

Vue Router already provides a set of interesting features by allowing the creation of routes which then display Vue components.

But Vue Router can also be used by JavaScript methods. So rather than writing <router-link> tags which display links allowing to navigate in the page, we can use the router.push(location) method which allows to display the page corresponding to the indicated path.

The navigation methods are described below, and also here https://router.vuejs.org/guide/essentials/navigation.html for more details.

router.push(location) method

The router.push(location, callbackSuccess, callbackError) method navigates to the specified path. This path is represented by the location variable which is:

- Either a character string indicating a path (for

example "/elements"),

- Let be an object (eg { path: "/elements" }).

The push() method is part of the router object created by new VueRouter(). The router object can also be accessed in components using this.$router.

Let's create a button in the page simulating a click on a link /elements. We keep the elements already created in the page, by simply adding this button and the associated process in the App component.

/elements button (src/App.vue file)

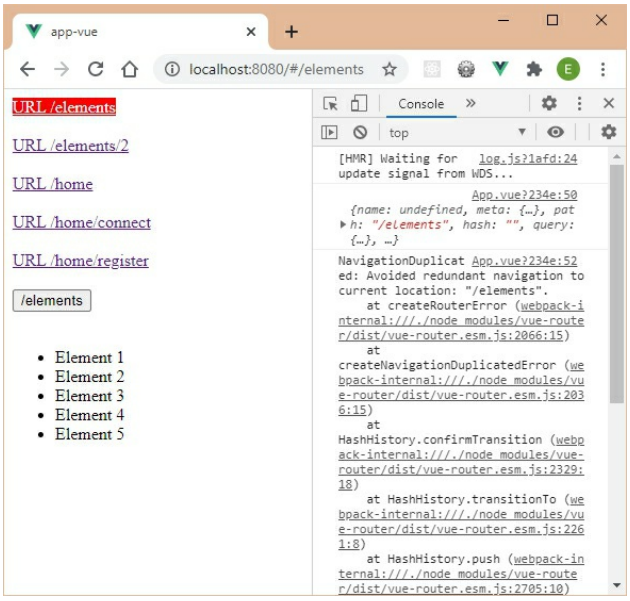
```
<template>
  <div>
    <router-link to="/elements">URL /elements</router-link>
    <br><br>
    <router-link to="/elements/2">URL /elements/2</router-link>
    <br><br>
    <router-link to="/home">URL /home</router-link>
    <br><br>
    <router-link to="/home/connect">URL /home/connect</router-link>
    <br><br>
    <router-link to="/home/register">URL /home/register</router-link>
    <br><br>
    <button v-on:click="elements">/elements</button>
    <br><br>
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
```

```
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";
import Connect from "@/components/Connect.vue";
import Register from "@/components/Register.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home",
      component : Home,
      children : [
        { path : "connect", component : Connect, alias : "/connect" },
        { path : "register", component : Register }
      ]
    },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1>
</div>" } }
  ]
});

export default {
  router,
  methods : {
    elements() {
      this.$router.push("/elements", function(route) {
        console.log(route);
      }, function(route) {
        console.log(route);
      });
    }
  }
}

</script>
```

```
<style>

.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```

Clicking the /elements button activates the elements() method of the App component. The router object is used through this.$router in the App component.

The push() method is used here by indicating the path as a string. The two callback functions listed below display the route object they use. In the case of two successive presses on the button, the error function is called indicating a NavigationDuplicated object because we are trying to display the URL already displayed.



Note that the push() method call can also be written using an object as the first parameter. For example :

Use an object to indicate the path

```
this.$router.push({ path : "/elements" }, function(route) {
  console.log(route);
}, function(route) {
  console.log(route);
});
```

router.replace(location) method

The router.replace(location) method is similar to router.push(location). The difference is that push() introduces a new entry into the browsing history, unlike replace() which replaces the previous entry with the new one.

router.go(n) method

Depending on the value of the integer n, the router.go(n) method is used to navigate in the browsing history:

- If n is positive, we advance in the history of n pages (similar to the Forward button of the browser),
- If n is negative, we go back in the history of n pages (similar to the Back button in the browser).

Let's create the Back and Forward buttons in the page

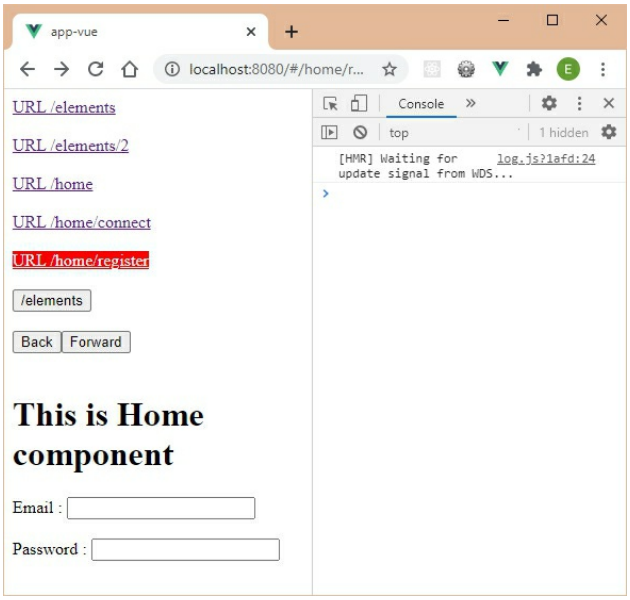
allowing to go back to the previous or next page.

Back and Forward buttons (src/App.vue file)

```
<template>
  <div>
    <router-link to="/elements">URL /elements</router-link>
    <br><br>
    <router-link to="/elements/2">URL /elements/2</router-link>
    <br><br>
    <router-link to="/home">URL /home</router-link>
    <br><br>
    <router-link to="/home/connect">URL /home/connect</router-link>
    <br><br>
    <router-link to="/home/register">URL /home/register</router-link>
    <br><br>
    <button v-on:click="elements">/elements</button>
    <br><br>
    <button v-on:click="back">Back</button>
    <button v-on:click="forward">Forward</button>
    <br><br>
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";
import Connect from "@/components/Connect.vue";
import Register from "@/components/Register.vue";

Vue.use(VueRouter);

var router = new VueRouter({
```

```
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home",
      component : Home,
      children : [
        { path : "connect", component : Connect, alias : "/connect" },
        { path : "register", component : Register }
      ]
    },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1>
</div>" } }
  ]
});

export default {
  router,
  methods : {
    elements() {
      this.$router.push({ path : "/elements" }, function(route) {
        console.log(route);
      }, function(route) {
        console.log(route);
      });
    },
    back() {
      this.$router.go(-1);
    },
    forward() {
      this.$router.go(1);
    }
  }
}

</script>

<style>
```

```
.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```



Manage events on the routes

Vue Router allows processing to be performed during certain events on the routes, and whether or not to continue processing on the route (for example, despite clicking on a link, you can decide not to go on this route).

Events can be managed at different locations in the program:

- On the router object: these events are used to perform global processing in the application, for

all routes.

- On the definition of a route: these events are used to perform processing on a particular route.
- On a component associated with the route: these events are used to perform processing on the routes of this component.

> The events on the routes are therefore processed by methods positioned on the router object, or in the definition of the route, or in the components of the application.

These events are triggered regardless of the mode in which the new route is accessed (click on a link, Back or Forward buttons in the browser, push(), replace() or go() methods previously seen on the router object).

Let us now examine the modes of definition of these events according to the place where they are positioned (router object, definition of a route or component).

Events associated with the router object

The events associated with the router object are used to act on all the routes of the application.

The router.beforeEach(callback) method is used to receive events related to a change of route (before it has actually occurred), regardless of the route. The callback function is triggered during this event.

The callback function is of the form callback(to, from, next):

- The parameter to indicates the route towards which we are heading,
- The parameter from represents the current route, which we will leave to go to the route to,
- The next parameter is a function allowing to go or not on the new route.

Let's use this beforeEach() method in the previous program, displaying in the console the value of the to object of the callback function.

Use router.beforeEach() method (src/App.vue file)

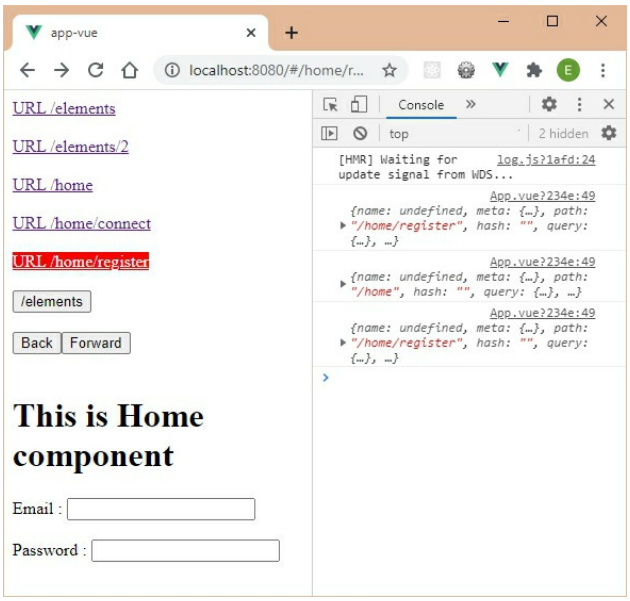
```
<template>
 <div>
  <router-link to="/elements">URL /elements</router-link>
  <br><br>
  <router-link to="/elements/2">URL /elements/2</router-link>
  <br><br>
  <router-link to="/home">URL /home</router-link>
  <br><br>
  <router-link to="/home/connect">URL /home/connect</router-link>
  <br><br>
  <router-link to="/home/register">URL /home/register</router-link>
  <br><br>
  <button v-on:click="elements">/elements</button>
  <br><br>
  <button v-on:click="back">Back</button>
  <button v-on:click="forward">Forward</button>
  <br><br>
```

```
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";
import Connect from "@/components/Connect.vue";
import Register from "@/components/Register.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home",
      component : Home,
      children : [
        { path : "connect", component : Connect, alias : "/connect" },
        { path : "register", component : Register }
      ]
    },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1>
</div>" } }
  ]
});

router.beforeEach(function(to, from, next) {
  console.log(to);
  next();
});

export default {
  router,
```

```
  methods : {
    elements() {
      this.$router.push({ path : "/elements" }, function(route) {
        console.log(route);
      }, function(route) {
        console.log(route);
      });
    },
    back() {
      this.$router.go(-1);
    },
    forward() {
      this.$router.go(1);
    }
  }
}

</script>

<style>

.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```

We simply added the call to the router.beforeEach() method, displaying the value of the to route each time the event is triggered. Then we call the next() method which allows you to go on the new route.

> Forgetting to call the next() method produces an error when the page is executed.



In the same way that we are warned before switching to another route, Vue Router can tell us that we have switched to a new route using the router.afterEach(callback) method.

The callback function in this case is of the form callback(to, from):

- The parameter to designates the route to which we have just accessed,
- The from parameter designates the route from which we came.

> Note that there is no next parameter here (since we have already reached the destination route).

Events associated with the definition of a route

You can also define events when defining a route. We then use the beforeEnter(callback) method to authorize or not the entry on this route (the route in which the method is defined).

> Note that the afterEnter() method does not exist on the definition of a route.

The callback function is of the form callback(to, from, next). The next parameter is used to validate access to the route, as we have seen for the router.beforeEach() method.

Let's see this on an example. We use the beforeEnter() method on the /home route.

Using the beforeEnter() method on the /home route

```
<template>
  <div>
    <router-link to="/elements">URL /elements</router-link>
    <br><br>
    <router-link to="/elements/2">URL /elements/2</router-link>
    <br><br>
    <router-link to="/home">URL /home</router-link>
    <br><br>
    <router-link to="/home/connect">URL /home/connect</router-link>
    <br><br>
    <router-link to="/home/register">URL /home/register</router-link>
    <br><br>
    <button v-on:click="elements">/elements</button>
    <br><br>
    <button v-on:click="back">Back</button>
    <button v-on:click="forward">Forward</button>
```

```
    <br><br>
    <router-view/>
  </div>
</template>

<script>

import Vue from "vue";
import VueRouter from "vue-router";

import Elements from "@/components/Elements.vue";
import Home from "@/components/Home.vue";
import Connect from "@/components/Connect.vue";
import Register from "@/components/Register.vue";

Vue.use(VueRouter);

var router = new VueRouter({
  routes : [
    { path : "/elements/:index(\\d+)?", component : Elements, props : true },
    { path : "/home",
      component : Home,
      beforeEnter(to, from, next) {
        console.log("beforeEnter() route /home");
        next();
      },
      children : [
        { path : "connect", component : Connect, alias : "/connect" },
        { path : "register", component : Register }
      ]
    },
    { path : "*", component : { template : "<div><h1>Unknown URL</h1>
</div>" } }
  ]
});

router.beforeEach(function(to, from, next) {
  next();
```

```
});

router.afterEach(function(to, from) {
  console.log(from);
});

export default {
  router,
  methods : {
    elements() {
      this.$router.push({ path : "/elements" }, function(route) {
        console.log(route);
      }, function(route) {
        console.log(route);
      });
    },
    back() {
      this.$router.go(-1);
    },
    forward() {
      this.$router.go(1);
    }
  }
}

</script>

<style>

.router-link-exact-active {
  background-color:red;
  color: white;
}

</style>
```

The beforeEnter() method is defined in the properties of the /home route. The call to the next() method

validates access to the route.

Events associated with a component

Finally, events can be defined on the component associated with a route. These events are handled by the following methods:

- The beforeRouteEnter(to, from, next) method is used to perform pre-entry processing on the route that displays this component. The call to the next() method allows to validate or not the access to the route.
- The beforeRouteUpdate(to, from, next) method is used to perform processing before the route is updated. This means that the same component will be used, as it is just an update of the route. If it is not an update of the route, it is the method beforeRouteLeave(to, from, next) which is called (see below).
- The beforeRouteLeave(to, from, next) method is called when we leave the route and also the component (otherwise it would be beforeRouteUpdate() that would be called).

Notice that in these three methods the this object represents the component in which the method is defined, except for the first beforeRouteEnter() method

for which this is null. Indeed, as we have not yet validated the entry on the route in this method, the component has not yet been created.

Let's use the Elements component on which we define these three methods. We also show how to access the this object represented by the Elements component, specifically in the beforeRouteEnter() method. Indeed, in this case, the call to the next() method uses a callback function, which when called, will have as a parameter an object representing the this of the component (because in this callback, the component is finally created).

Use route events on the Elements component (src/components/Elements.vue file)

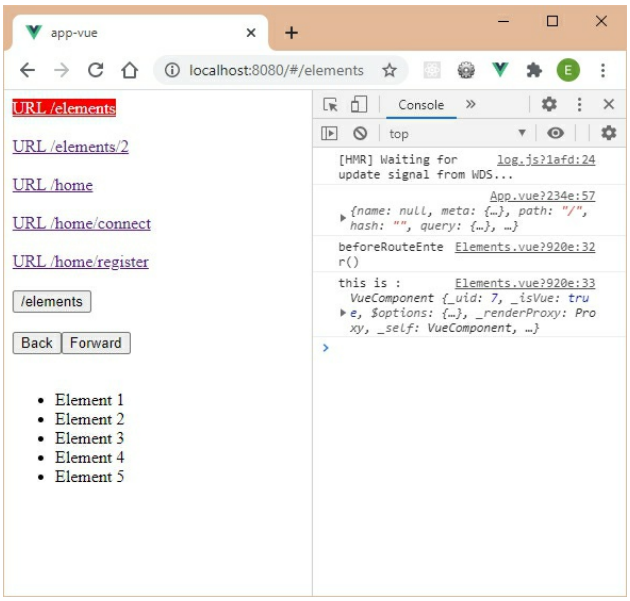
```
<template>
  <div>
    <router-view/>
    <ul>
      <li v-for="(element, i) in elements" v-bind:key="i"
          v-show="index==undefined || index==i">{{element}}</li>
    </ul>
  </div>
</template>
<script>
export default {
  data() {
    return {
      elements : [
        "Element 1",
        "Element 2",
```

```
      "Element 3",
      "Element 4",
      "Element 5"
    ]
  }
 },
 computed : {
  index() {
   return this.$route.params.index;
  }
 },
 beforeRouteEnter(to, from, next) {
  next(function(vm) {
   console.log("beforeRouteEnter()");
   console.log("this is : ", vm);
  });
 },
 beforeRouteUpdate(to, from, next) {
  console.log("beforeRouteUpdate()");
  console.log("this is : ", this);
  next();
 },
 beforeRouteLeave(to, from, next) {
  console.log("beforeRouteLeave()");
  console.log("this is : ", this);
  next();
 }
}
</script>
```

In the beforeRouteEnter() method, we use the callback function associated with the next(callback) method to obtain the component's this as a parameter.

In all other methods, the this object associated with the

component is directly accessible.



10 – USE MIXINS AND PLUGINS

Mixins and plugins are ways to easily import functionality into Vue programs. They allow the basic functionality of Vue to be enriched by adding those written in special modules called mixins and plugins.

This is for example what we did using Vuex and Vue Router (see previous chapters). Vuex and Vue Router are actually two plugins that allow respectively:

- Add a common store functionality to the entire application (for Vuex),
- Add a routing system with URLs (for Vue Router).

Vue makes it easy to integrate this kind of functionality. Let's see this on examples.

Using mixins

A mixin is a JavaScript object used to add new functionality to a Vue component, or to the Vue

module itself:

- If we add the functionalities to the Vue module used in our program, these functionalities will be accessible to all the components of the program,
- If the functionalities are added only to one or more components, these functionalities will only be accessible in the components for which they have been added.

> The Vue module is the one retrieved by an instruction of the type import Vue from "vue".

Let's first look at how to create a mixin, then how to integrate it into our program according to these two types of use of mixins.

Creating a mixin

A mixin is a JavaScript object similar to those already used when creating components. In this object, you will find the usual sections used when defining a component:

- The data section allows you to define the reactive variables,
- The methods section allows you to define methods that use reactive variables if necessary,
- The computed section defines the computed properties,

- The filters section allows you to define the filters.

> There is no props or template section because that is not the function of a mixin.

In addition, as the mixin will be integrated into the Vue module or a Vue component, we can add to these sections those that allow you to manage the life cycle of a component (methods created(), beforeMount(), mounted(), etc ...).

Now let's create a mixin to manage a person's name and city. The mixin is created in a MyMixin.js file located in the src/mixins directory (directory which will contain all the mixins created for the application).

The mixin being defined in a module, the object associated with the module is exported by means of the export default { … } instruction.

src/mixins/MyMixin.js file

```
export default {
  data() {
    return {
      name : "Robert",
      city : "Atlanta"
    }
  },
  created : function() {
    console.log("MyMixin.created :", this);
  },
```

```
  methods : {
    nameInCity() {
      return this.name + " in " + this.city;
    },
    displayNameInCity : function() {
      console.log(this.nameInCity());
    }
  },
  computed : {
    name_in_city() {
      return this.nameInCity();
    },
    name_in_city_h1() {
      return "<h1>" + this.nameInCity() + "<h1>";
    }
  }
};
```

The mixin created here is similar to creating a component. It will allow you to add the features that are described in the mixin to the Vue module directly or to the components that wish to use it.

This means that components that have access to the mixin, will be able to use the reactive variables, methods and calculated properties of the mixin as if they were internal to the component.

Adding a mixin to the Vue module

Adding the mixin to the Vue module allows all our components to benefit from the features described in the mixin module.

> Attention: this must be well thought out because all the components are modified by this addition! For example all reactive variables of the mixin will become reactive variables of all components.

Reactive variables (data section), calculated properties (computed section), methods (methods section) and lifecycle methods will be accessible in all components used in the application.

Adding the mixin to the Vue module is done using the Vue.mixin(MyMixin) instruction, having previously imported the MyMixin mixin using the import instruction.

If you want to integrate several mixins, you just have to perform the Vue.mixin(...) instruction several times.

Import and add the mixin to the Vue module (src/App.vue file)

```
<template>
  <Elements />
</template>

<script>

import Vue from "vue";
import Elements from "@/components/Elements.vue";
import MyMixin from "@/mixins/MyMixin.js";

Vue.mixin(MyMixin);

export default {
  created() {
    console.log("App created");
```

```
  },
  components : {
    Elements
  },
  methods : {
  }
}

</script>

<style>

</style>
```

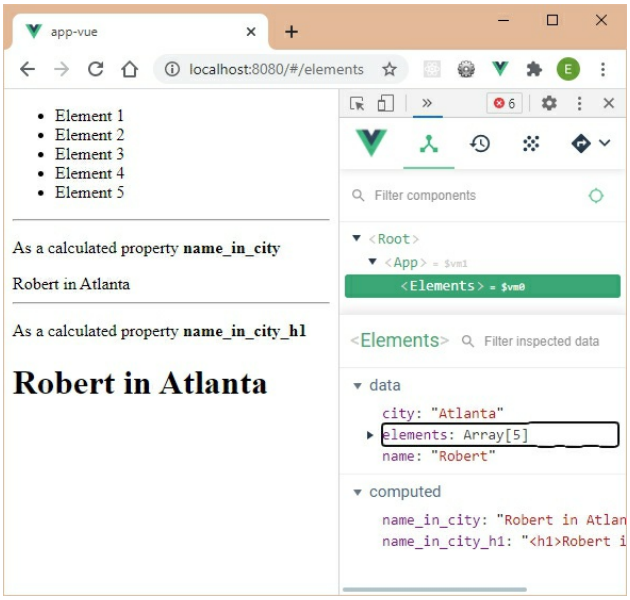
The App component imports the mixin to the Vue module, and displays in its template the Elements component, described below:

src/components/Elements.vue file

```
<template>
  <div>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
    <hr>
    <p>As a calculated property <b>name_in_city</b></p>
    {{name_in_city}}
    <hr>
    <p>As a calculated property <b>name_in_city_h1</b></p>
    <div v-html="name_in_city_h1"></div>
  </div>
</template>
<script>
export default {
  data() {
```

```
    return {
      elements : [
        "Element 1",
        "Element 2",
        "Element 3",
        "Element 4",
        "Element 5"
      ]
    }
  },
  created() {
    console.log("Elements created");
    this.displayNameInCity();
  }
}
</script>
```

The Elements component uses the data defined in the mixin, as if it were its own. All other components used in the application could do the same.



We can clearly see in the Vue tab that the Elements

component perfectly integrates the reactive variables defined in the mixin, as well as the calculated properties, by integrating them into its own.

Moreover, it would be useful, in a mixin, to indicate names which do not risk overlapping the names of the data already defined in the components (for example if the city variable was also defined in the Elements component, it would be in conflict with that defined in the mixin).

Adding a mixin to a component

A mixin can be used in one or more components (apart from all the components of the application, otherwise it must be done as indicated in the previous section by integrating it into the Vue module).

> This way of using a mixin is the most common in Vue programs.

To use a mixin in a component, you must:

- Import the mixin file using the import instruction (as before),
- Indicate that the component uses the mixin, by means of the mixins section in the component. We indicate in the mixins section, in array form, the list of all the mixins used by the component.

The mixin remains identical to that defined previously.

It is the way of integrating it into the components that differs.

To use the mixin directly in the App component:

Use a mixin in the App component (src/App.vue file)

```
<template>
  <Elements />
</template>

<script>

import Vue from "vue";
import Elements from "@/components/Elements.vue";
import MyMixin from "@/mixins/MyMixin.js";

export default {
  created() {
    console.log("App created");
    this.displayNameInCity();
  },
  components : {
    Elements
  },
  methods : {
  },
  mixins : [
    MyMixin
  ]
}

</script>

<style>

</style>
```

We import the mixin file and declare it in the mixins

section. This allows us to use all the data defined in the mixin (here we use the displayNameInCity() method which displays the name and the city in the console).

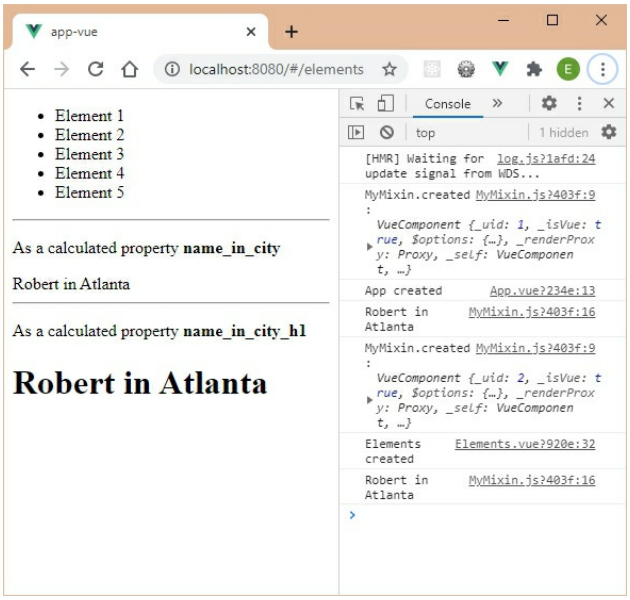
The Elements component must do the same in order to continue using the mixin.

Use a mixin in the Elements component (src/components/Elements.vue file)

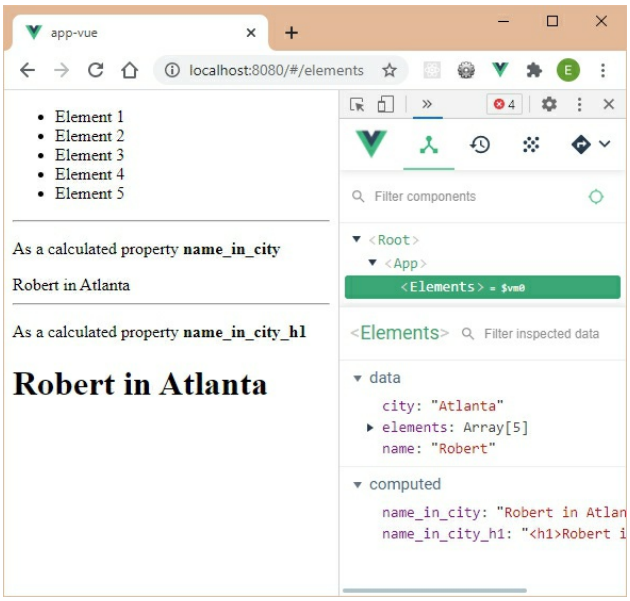
```
<template>
  <div>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
{{element}}</li>
    </ul>
    <hr>
    <p>As a calculated property <b>name_in_city</b></p>
    {{name_in_city}}
    <hr>
    <p>As a calculated property <b>name_in_city_h1</b></p>
    <div v-html="name_in_city_h1"></div>
  </div>
</template>
<script>
import MyMixin from "@/mixins/MyMixin.js";
export default {
  data() {
    return {
      elements : [
        "Element 1",
        "Element 2",
        "Element 3",
        "Element 4",
        "Element 5"
```

```
    ]
   }
  },
  created() {
   console.log("Elements created");
   this.displayNameInCity();
  },
  mixins : [
   MyMixin
  ]
}
</script>
```

The Console tab shows that the created() methods are executed correctly (in the mixin and components):



The Vue tab shows the integration of reactive variables and calculated properties in the components (here for the Elements component):



Use plugins

We have seen that with a mixin we could transmit data to users that will be accessible in the component that integrates the mixin.

Now suppose we want to pass components or directives, and make them easily accessible to other Vue applications? We then integrate them into a plugin.

This is what we did, but in a different way, when we had created the directories directives, components (which already existed), etc. in the src directory, and that we import each file into the component that uses it.

With a plugin, everything will be grouped in the plugin file, and we will only have to import the plugin file to use what it contains: components, directives, mixins,

even filters.

Creating a plugin

A plugin is a JavaScript module that exports an object having the install(Vue, options) method as a property.

For example, let's create the MyPlugin.js file which contains this minimum structure. It will be enriched thereafter. This file is created in the src/plugins directory which will contain our plugins.

src/plugins/MyPlugin.js file

```
export default {
  install(Vue, options) {
    console.log("MyPlugin install");
  }
}
```

Note that as the Vue module is indicated as a parameter of the install() method, this module does not need to be imported into the plugin file.

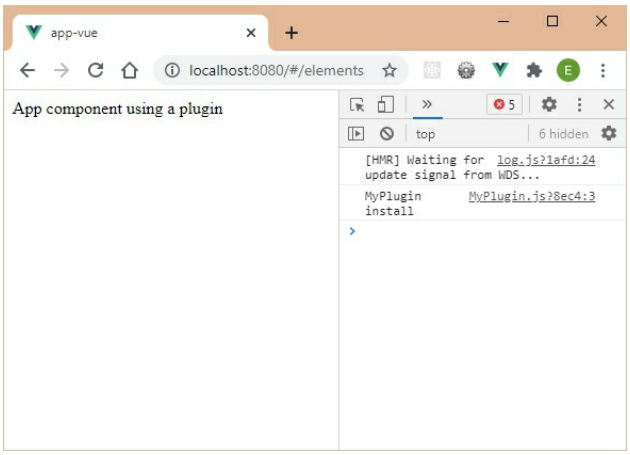
Using a plugin

To install this plugin in our Vue application, all you have to do is import the plugin file and then use the Vue.use(plugin) method. This must be done before the main Vue object of the application is created. Since this Vue object is created in the main.js file which imports the src/App.vue file first, these instructions can be inserted into the App.vue file.

Insertion and use of the plugin (src/App.vue file)

```
<template>
  <div>
    App component using a plugin
  </div>
</template>

<script>

import Vue from "vue";
import MyPlugin from "@/plugins/MyPlugin.js";

Vue.use(MyPlugin);

export default {
  data() {
    return {
    }
  },
  components : {
  },
  methods : {
  }
}

</script>

<style>

</style>
```

The display in the console shows that the plugin's install() method is correctly executed after calling the Vue.use(MyPlugin) method.



Now that we've seen how to create and use the plugin, let's enrich it with things like components, directives, etc.

Adding a component in a plugin

Let's start by adding a component in the MyPlugin.js plugin. For example let's add the Elements component. This component can be defined in an external file (eg Elements.vue) or be defined directly in the plugin file.

Even if the first way of defining the component is the most used, we show the two ways of doing it below.

Define the component in an external file

The Elements component file is for example the following:

src/components/Elements.vue file

```
<template>
  <div>
    <ul>
```

```
      <li v-for="(element, index) in elements" v-bind:key="index">
        {{element}}
      </li>
    </ul>
  </div>
</template>
<script>
export default {
  data() {
    return {
      elements : [
        "Element 1",
        "Element 2",
        "Element 3",
        "Element 4",
        "Element 5"
      ]
    }
  },
  created() {
  },
  components : {
  }
}
</script>
```

The Elements component is defined in the traditional way. It is here in its simplest form, namely that it does not depend on any other component (for example it does not currently include the Element component which describes a list item).

The Elements component is included in the plugin as follows:

src/plugins/MyPlugin.js file

```
import Elements from "@/components/Elements.vue";
export default {
  install(Vue, options) {
    console.log("MyPlugin install");
    Vue.component("Elements", Elements);
    console.log("Elements install");
  }
}
```

The component being defined in an external file, this file is first imported into the plugin file. Then the component is installed in the plugin using Vue.component().

This means that when the plugin will install via Vue.use(MyPlugin), the Elements component will be accessible everywhere (without having to import Elements).

For example let's use the Elements component in the App component.

Use the Elements component in the App component (src/App.vue file)

```
<template>
  <div>
    App component using a plugin
    <Elements />
  </div>
</template>

<script>
```

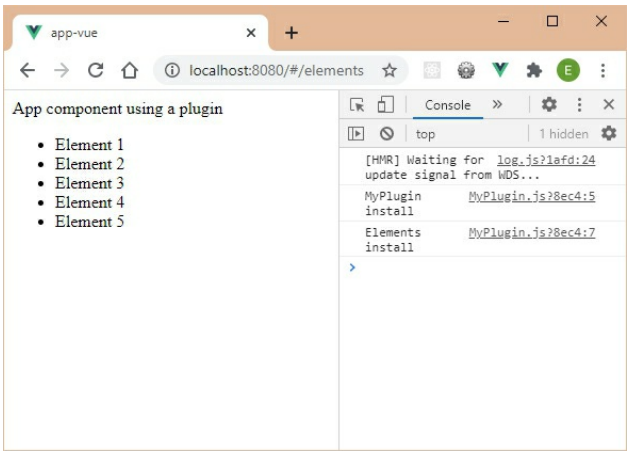
```
import Vue from "vue";
import MyPlugin from "@/plugins/MyPlugin.js";

Vue.use(MyPlugin);

export default {
  data() {
    return {
    }
  },
  components : {
  },
  methods : {
  }
}

</script>

<style>

</style>
```

The App component uses the Elements component in its template, and yet the App component:

- Does not include Elements module,
- Do not declare it in its components section.

We see that the use of the plugin simplifies the use of the components that are included in it.

Let's check that the list of elements is displayed:



Define the component in the plugin

This way of proceeding is less common because it writes the component code into the plugin code, which is less maintainable.

src/plugins/MyPlugin.js file

```
export default {
  install(Vue, options) {
    console.log("MyPlugin install");
    Vue.component("Elements", {
      data() {
        return {
          elements : [
            "Element 1",
            "Element 2",
            "Element 3",
            "Element 4",
            "Element 5"
          ]
        }
      },
      created() {
      },
      components : {
```

```
    },
    template : `
      <div>
        <ul>
          <li v-for="(element, index) in elements" v-bind:key="index">
            {{element}}
          </li>
        </ul>
      </div>
    `
  });
  console.log("Elements install");
 }
}
```

The code of the Elements component is directly registered in the plugin, by creating the template section in the component.

The Elements.vue file is no longer useful in this case. The App component file remains the same.

Adding other components in the plugin

By using a .vue file for each component, you will need in the install() method of the plugin:

- Import the component file,
- Use the Vue.component() method to define it in the plugin.

Each component thus defined becomes usable in the entire Vue application.

What if one component of the plugin uses another component?

Suppose the Elements component internally uses the Element component (which allows you to define a single list item).

We can of course declare this new component in the plugin (as we did for the Elements component), but very often this new Element component is already declared in its parent Elements. It is therefore not necessarily useful to also declare it in the plugin (if it is not used elsewhere).

Here's how to do it. The Elements component is modified to introduce the Element component that it uses.

src/components/Elements.vue file

```
<template>
  <div>
    <ul>
      <li v-for="(element, index) in elements" v-bind:key="index">
        <Element v-bind:text="element" />
      </li>
    </ul>
  </div>
</template>
<script>
import Element from "@/components/Element.vue";
export default {
  data() {
```

```
    return {
      elements : [
        "Element 1",
        "Element 2",
        "Element 3",
        "Element 4",
        "Element 5"
      ]
    }
  },
  created() {
  },
  components : {
    Element
  }
}
</script>
```

The Element component is not included in the plugin, to use it in another component, import it and declare it in the components section.

The Element component is as follows:

src/components/Element.vue file

```
<template>
  <span style="font-size:20px;">
    {{text}}
  </span>
</template>
<script>
export default {
  props : [
    "text"
  ],
```

```
  data() {
    return {
    }
  },
  created() {
  },
  updated() {
  },
  methods : {
  },
  components: {
  }
}
</script>
<style>
</style>
```

We use a 20px font to check that the Element component is taken into account in the display.

The plugin file and the App component file are as follows:

src/plugins/MyPlugin.js file

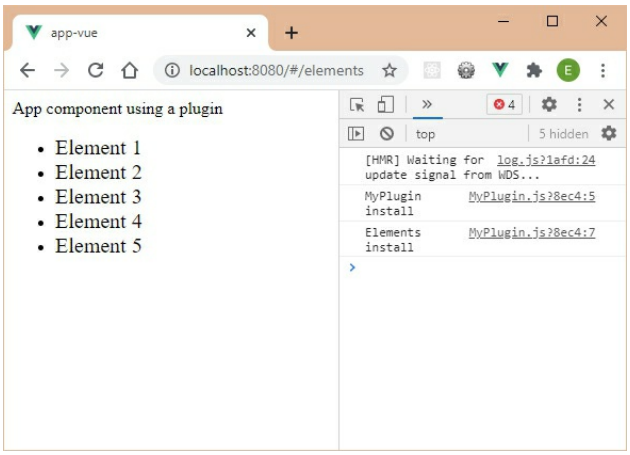
```
import Elements from "@/components/Elements.vue";
export default {
  install(Vue, options) {
    console.log("MyPlugin install");
    Vue.component("Elements", Elements);
    console.log("Elements install");
  }
}
```

src/App.vue file

```
<template>
```

```
  <div>
    App component using a plugin
    <Elements />
  </div>
</template>

<script>

import Vue from "vue";
import MyPlugin from "@/plugins/MyPlugin.js";

Vue.use(MyPlugin);

export default {
  data() {
    return {
    }
  },
  components : {
  },
  methods : {
  }
}

</script>

<style>

</style>
```

Let's check that everything is working:



Adding a directive in a plugin

Assume that the directive is written to an external file, for example located in the src/directives directory.

We previously created two directives in this directory (see chapter 4 "*Creating directives*"):

- The v-log directive (in the src/directives/log.js file) displays in the console information about the element to which it applies,
- The v-hide directive (in the src/directives/hide.js file) allows you to hide (if true) or display (if false) the element to which it applies.

The guidelines were described as follows. They use the definition by Vue.directive().

src/directives/log.js file

```
import Vue from "vue"
import { alertMsg1, alertMsg2 } from "@/assets/helpers.js"
```

```
Vue.directive("log", {
  inserted(el) {
    alertMsg1("v-log directive applied on el element : ", el);
    alertMsg2("Whose parent is:", el.parentElement);
  }
});
```

src/directives/hide.js file

```
import Vue from "vue"
Vue.directive("hide", function(el, binding) {
  var hide = binding.value;
  console.log("hide = " + hide);
  if (hide) el.style.display = "none";
});
```

Let's integrate these two directives into the MyPlugin.js plugin. All you have to do is import the corresponding JavaScript files into the plugin.

src/plugins/MyPlugin.js file

```
import Elements from "@/components/Elements.vue"
import hide from "@/directives/hide.js"
import log from "@/directives/log.js"
export default {
  install(Vue, options) {
    Vue.component("Elements", Elements);
  }
}
```

The directive files are simply included, and no further processing is done in the install() method of the plugin.

> Of course, if the directives were not defined in external files like here, we could indicate their treatment in the install() method of the plugin (also

using Vue.directive()).

Let's use these guidelines in our components. We use the v-log directive in the App component, and the v-hide="true" directive to hide the first item in the list in the Elements component.

These directives having been associated with the plugin, are accessible everywhere (as for the components declared in the plugin).

src/App.vue file

```
<template>
  <div>
    App component using a plugin
    <Elements v-log />
  </div>
</template>

<script>

import Vue from "vue";
import MyPlugin from "@/plugins/MyPlugin.js";

Vue.use(MyPlugin);

export default {
  data() {
    return {
    }
  },
  components : {
  },
  methods : {
  }
}
```

```
</script>

<style>

</style>
```

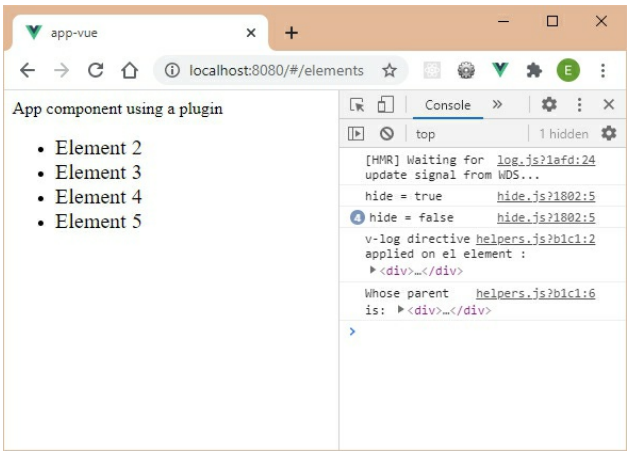
The v-log directive has been positioned on the Elements component displaying the set of all the elements in the list.

src/components/Elements.vue file

```
<template>
  <div>
    <ul>
      <li v-for="(element,index) in elements" v-bind:key="index" v-hide="index==0" >
        <Element v-bind:text="element"/>
      </li>
    </ul>
  </div>
</template>
<script>
import Element from "@/components/Element.vue"
export default {
  data() {
    return {
      elements : [
        "Element 1",
        "Element 2",
        "Element 3",
        "Element 4",
        "Element 5"
      ]
    }
  },
  created() {
```

```
  },
  components : {
    Element
  }
}
</script>
```

The v-hide directive is set to true on the first item in the list, and false for the other items.



Adding a filter in a plugin

The principle for adding a filter in a plugin is the same as for adding a directive. We import the JavaScript file associated with the filter into the plugin file.

Consider the upper filter that we defined in the src/filters/upper.js file. Its content was as follows:

src/filters/upper.js file

```
import Vue from "vue"
Vue.filter("upper", function(value) {
  return value.toUpperCase();
});
```

Importing the file into the plugin is as follows:

src/plugins/MyPlugin.js file

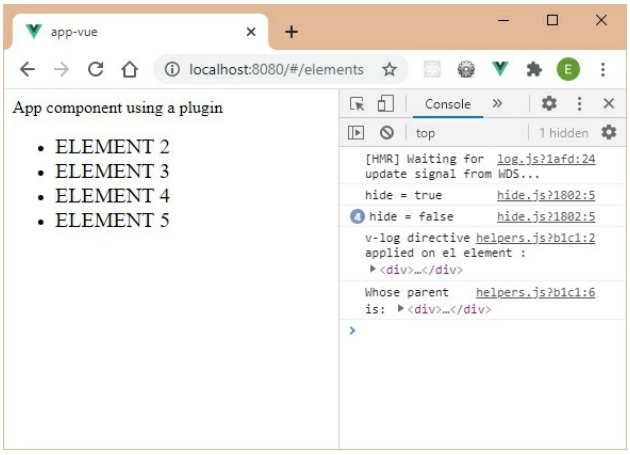
```
import Elements from "@/components/Elements.vue"
import hide from "@/directives/hide.js"
import log from "@/directives/log.js"
import upper from "@/filters/upper.js"
export default {
  install(Vue, options) {
    Vue.component("Elements", Elements);
  }
}
```

The use of the filter is done here in the Element component, in order to capitalize the displayed list items.

Using the upper filter in the Element component (src/components/Element.vue file)

```
<template>
  <span style="font-size:20px;">
    {{text|upper}}
  </span>
</template>
<script>
export default {
  props : [
    "text"
  ],
  data() {
    return {
    }
  },
  created() {
```

```
  },
  updated() {
  },
  methods : {
  },
  components: {
  }
}
</script>
<style>
</style>
```



Adding a mixin in a plugin

Adding a mixin is similar. Let's take the previous mixin located in src/mixins/MyMixin.js.

src/mixins/MyMixin.js file

```
export default {
  data() {
    return {
      name : "Robert",
      city : "Atlanta"
    }
```

```
  },
  created : function() {
    console.log("MyMixin created");
  },
  methods : {
    nameInCity() {
      return this.name + " in " + this.city;
    },
    displayNameInCity : function() {
      console.log(this.nameInCity());
    }
  },
  computed : {
    name_in_city() {
      return this.nameInCity();
    },
    name_in_city_h1() {
      return "<h1>" + this.nameInCity() + "<h1>";
    }
  }
};
```

The use of the mixin in the plugin is done by importing the file, and by using the Vue.mixin() method in the install() method of the plugin. This is because the imported file only contains one object that must be declared as a mixin, hence the Vue.mixin() statement.

The plugin file is modified as follows:

src/plugins/MyPlugin.js file

```
import Elements from "@/components/Elements.vue"
import hide from "@/directives/hide.js"
import log from "@/directives/log.js"
import upper from "@/filters/upper.js"
```

```
import MyMixin from "@/mixins/MyMixin.js"
export default {
  install(Vue, options) {
    Vue.component("Elements", Elements);
    Vue.mixin(MyMixin);
  }
}
```

> Be careful because the mixin being here associated with the Vue module, this means that its data will be accessible throughout the Vue application (as we explained previously in the section on mixins).

Let's use the data from the mixin in the App component. In particular let's use the computed name_in_city property displayed below the list of items.

src/App.vue file

```
<template>
  <div>
    App component using a plugin
    <Elements v-log />
    {{name_in_city}}
  </div>
</template>

<script>

import Vue from "vue";
import MyPlugin from "@/plugins/MyPlugin.js";

Vue.use(MyPlugin);

export default {
  data() {
```

```
    return {
    }
  },
  components : {
  },
  methods : {
  }
}

</script>

<style>

</style>
```