

UNIVERSIDAD AUTÓNOMA DE MADRID

DEPARTAMENTO DE INFORMÁTICA

Computer Systems Project

Assignment 2.1

Roberto MARABINI
Alejandro BELLOGÍN

Changelog

Version ¹	Date	Author	Description
1.0	10.10.2022	RM	First version.
1.1	5.12.2022	RM	Renumbering assignment 3 -> 2
1.2	12.12.2022	AB	Translation to English
1.3	26.1.2023	RM	Review before uploading the document to moodle

¹Version control is made using 2 numbers $X.Y$. Changes in Y denote clarifications, more detailed descriptions of some aspect, or translations. Changes in X denote deeper modifications that either change the provided material or the content of the assignment.

Contents

1	Using <i>Vue.js</i>	3
1.1	Using <i>Vue.js</i> in a static file	3
1.2	Use of <i>Vue.js</i> creating a project	7
1.3	Development environment in <i>Vue.js</i>	10
1.3.1	Syntax highlighting in <i>Vue.js</i>	10
1.3.2	DevTools installation for <i>Vue.js</i>	10
1.3.3	Including a CSS framework	10
1.4	Structure of a <i>Vue.js</i> file	11
1.5	Creating components	12
1.5.1	Create a component with <i>Vue.js</i>	12
1.5.2	Add the component to the application	13
1.5.3	Add data to the component	15
1.5.4	Add a loop to the component	18
1.6	Create forms	19
1.6.1	Create a form with <i>Vue.js</i>	19
1.6.2	Add a form to the application	21
1.6.3	Link the fields of the form with their state	23
1.6.4	Add a sending method to the form	25
1.6.5	Send events from the form to the application	26
1.6.6	Receive events from the table in the application	27
1.7	Validations with <i>Vue.js</i>	28
1.7.1	Computed properties in <i>Vue.js</i>	28
1.7.2	Conditional sentences with <i>Vue.js</i>	31
1.7.3	References with <i>Vue.js</i>	35
1.8	Remove elements with <i>Vue.js</i>	36
1.8.1	Add a remove button to the table	36
1.8.2	Emit a remove event from the table	37
1.8.3	Receive the remove event in the application	37
1.8.4	Add an informative message	38
1.9	Edit elements with <i>Vue.js</i>	39

1.9.1	Add an edit button to the table	39
1.9.2	Add an edit method to the table	39
1.9.3	Add edit fields to the table	40
1.9.4	Add a save button to the table	42
1.9.5	Emit the save event	43
1.9.6	Add a method that cancels the edit	44
1.9.7	Receive the update event in the application	44
1.10	Build & Deploy the <i>Vue.js</i> application	45
1.10.1	Building the <i>Vue.js</i> application	45
1.10.2	Deploying the <i>Vue.js</i> application	46
1.11	Summary of the work done until now	46

Aviso

The project described next is based on the following tutorials: “Tutorial de introducción a *Vue.js* 3” (<https://www.neoguias.com/tutorial-vue/>) and “Cómo crear una aplicación REST con *Vue.js*” (<https://www.neoguias.com/tutorial-rest-vue/>).

In <https://worldline.github.io/vuejs-training/> you may find an introduction to *Vue.js* that complements the information included in this document.

1 Using *Vue.js*

You may *Vue.js* in different ways. First, let us see how to inject *Vue.js* code in an HTML file using a static file, later on we will create a more complex project.

1.1 Using *Vue.js* in a static file

This is the simplest option, as it is enough for us to load a javascript file in the head section of an HTML file and include the *Vue.js* directives inside the `div` that is identified through its `id` attribute, whose value will be `app`. In the following HTML file we see how to add it:

```
<!--index.html-->
<!DOCTYPE html>
<html lang="en">
  <head>
    <script
      src="https://unpkg.com/vue@3.2/dist/vue.global.prod.js">
    </script>

    <title>Basic application with Vue</title>
  </head>

  <body>
```

```
<div id="app"></div>
</body>
</html>
```

As you may see, we have included the version 3.2 of *Vue.js*. To include other versions, you may simply change the part @3.2 of the URL, and specify whatever version number you want to add instead. In case a development version of *Vue.js* wants to be included (instead of a production one), we should include the following code and replace the previous one:

```
<script src="https://unpkg.com/vue@3.2/dist/vue.global.js">
</script>
```

In general, it is better not to use a production version during development because the error messages will be less detailed. Next, we shall create the simplest application one can create with *Vue.js*, that is, a Hello World.

For this, we will include a script right before the closing `body` tag. In the script, we will create a new instance of the *Vue.js* application by running the sentence `app = Vue.createApp({ ... })`, that will receive an object as parameter:

```
...
<script>
const app = Vue.createApp({
  data() {
    return {
      # if you copy and paste this code be careful
      # because the quote in a pdf file are typographical
      # therefore you will need to delete and re-type them.
      greeting: 'Hello world'
    }
  },
});
</script>
```

The object we passed to the *Vue.js* class constructor includes the `data` method, which must return the data we want to initialize our application with. In our case, it will only be the `greeting` variable.

To initialize the *Vue.js* application we just created, we must use the `mount` method, that will receive the element identifier in which we want to render the application, which in our case it is the `#app` div:

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

Next, we render the `greeting` property inside of the `app` div using the syntax `{{ greeting }}`:

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

This would be the complete code for this first part of the introductory application:

Listing 1: Complete code of the first application.

```
<html lang="es">
  <head>

    <script src="https://unpkg.com/vue@3.2/dist/vue.global.prod.js"></
      ↪ script>

    <title>Basic application with Vue</title>
  </head>

  <body>
    <div id="app">{{ greeting }}</div>

    <script>
      const app = Vue.createApp({
        data() {
          return {
            greeting: 'Hello world'
          }
        }
      })
      app.mount('#app')
    </script>
  </body>
</html>
```

```
    }  
  },  
});  
  
app.mount('#app');  
</script>  
</body>  
</html>
```

As you may see, we have used something similar to JavaScript with a strange syntax that, in a normal situation, would raise errors. However, when the code is executed, you may check it results in Figure 1:

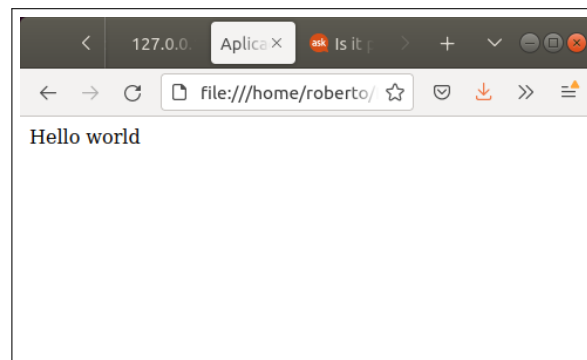


Figure 1: Web page created by the code 1.

there are not errors because the *Vue.js* code is translated into JavaScript code through a compiler called Babel (<https://babeljs.io>), although you may not notice it. This will happen every time the application runs. However, it is not common to include *Vue.js* in this way, because of this, the compilation, as you will see, only needs to occur when you develop the application and not when your users use it in production. We have used the word “compiler” in an imprecise way since Babel does not produce machine code, but it uses a code as input *Vue.js* and as output JavaScript, which means that the input and output are languages of the same level, but one of them is interpretable directly by the browsers. The correct term for this operation is not “compiling” but “transpiling”.

It is completely normal to think all of this is too complicated when it is possible to show the sentence “hello world” with pure HTML more easily. Do not worry, when you finish this tutorial you will see the possibilities that *Vue.js* opens up.

1.2 Use of *Vue.js* creating a project

Vue.js code is not often embedded directly into an HTML page because of the inconvenience of transpiling the code everytime the user loads the application. Usually, what we want to do is to create a *Vue.js* project and transpile the *Vue.js* code into JavaScript before uploading it to the server, in such a way that the browsers could understand it without any further extra operation.

Vue.js includes a command line interface that will help you develop the applications. Let us see now how to install it. For this, you should open a terminal and run the following command:

```
npm install vue@3.2.27
```

To create a new application, locate yourself in the folder where you want to create your application and run the following command:

```
npm init vue@3.2 tutorial-vue
```

and answer no to all the questions (in the next project we will discuss some of these options in detail)

```
Add TypeScript? No** / Yes
Add JSX Support? No** / Yes
Add Vue Router for Single Page Application development? No** / Yes
Add Pinia for state management? No** / Yes
Add Vitest for Unit Testing? No** / Yes
Add Cypress for End-to-End testing? No** / Yes
Add ESLint for code quality? No** / Yes
Add Prettier for code formatting? No** / Yes
```

After a few seconds, the application will be created. Next, move to the folder that was just created:

```
cd tutorial-vue
```

Then, run the following command to start the **development server**, which by default will be located in port **3000**:

```
npm install  
npm run dev
```

Once the application is created, open your browser and go to `http://localhost:3000/` to see the application, which includes by default a page similar to figure 2:

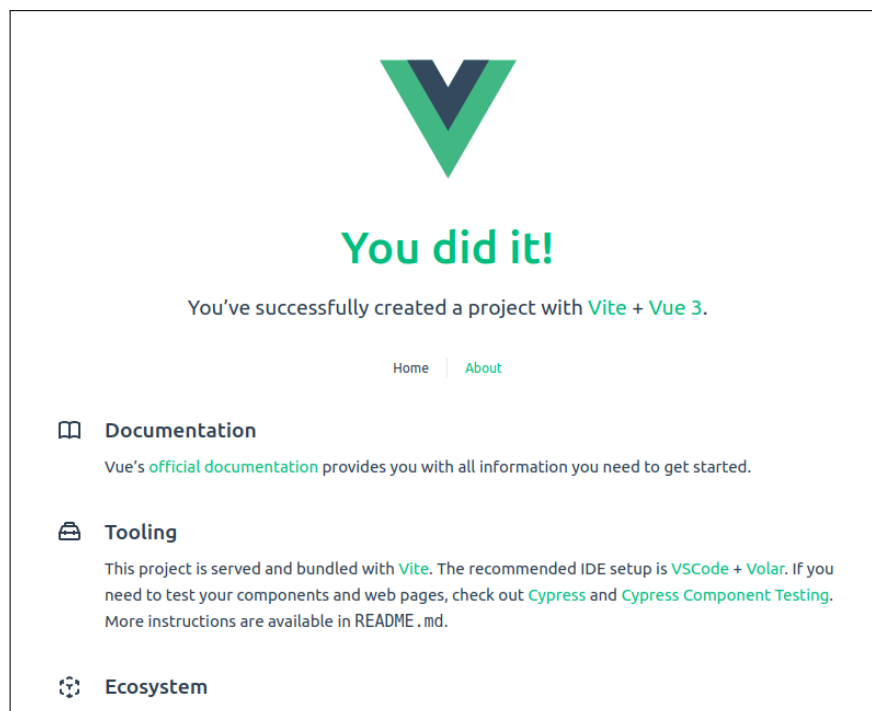


Figure 2: Web page created by default by *Vue.js* 3.2.

The folder where your code will be located is `src`, where you will find the file `main.js`, which is the entry point of the application.

The content of file `main.js` (see listing 2) depends on the selected options and should always import: (a) the `createApp` method and (b) the main code of the

application, located in file `App.vue`. The application is created with the command `createApp(App)` and is “rendered” in an HTML element of our page, which in this case is the `#app` div, with the command `mount('#app')`:

Listing 2: Example of file `main.js`.

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

// you may want to clean default main.css file
// since default options may no satisfy you.
import './assets/main.css'

createApp(App).mount('#app')
// the two next lines will make bootstrap available to your
// application if bootstrap has been installed.
// Instalation instructions are available later in this guide
import './node_modules/bootstrap/dist/js/bootstrap.js'
import './node_modules/bootstrap/dist/css/bootstrap.min.css'
```

If you now go to file `App.vue`, you will see it is relatively complex. The only important lines are:

Listing 3: Extract of `App.vue` file.

```
<template>
  <div id="app" class="container">
    </div>
</template>
```

as in the previous case, only the code that is inside `div id=app` will have access to the variables defined by `Vue.js`.

1.3 Development environment in *Vue.js*

After installing *Vue.js*, we could start programming the application. However, let us first configure a set of tools that will be helpful during the process.

1.3.1 Syntax highlighting in *Vue.js*

Although you may use any IDE, we recommend you to use some plugin to **highlight the syntax** which, in addition, could also format the code. If you use **VS Code**, simply install the plugin **Vetur**.

1.3.2 DevTools installation for *Vue.js*

Before continuing, it is recommendable that you install the `vue.js` DevTools plugin in your browser. This plugin will give you access to the *Vue.js* variables that the browser stores locally.

1.3.3 Including a CSS framework

We recommend you to use some CSS framework like **Bootstrap**. If you have never used Bootstrap, you may check the tutorial in `/bootstrap/introduction` to Bootstrap.

To include Bootstrap in the project, open the file `src/main.js` and include the following lines at the end (see listing 2)

```
import "../node_modules/bootstrap/dist/js/bootstrap.js";  
import '../node_modules/bootstrap/dist/css/bootstrap.min.css'
```

you will also need to install bootstrap using `node` by typing in the terminal

```
# If you are not in the project directory  
# cd to it before executing these commands  
npm install bootstrap@5.1.3  
npm install @popperjs/core@2.11.5
```

Please, install the versions we suggest to avoid incompatibility problems.

1.4 Structure of a *Vue.js* file

Vue.js files are divided in three very different sections. First, we have the **template** section, where we add the **HTML** code of the application. Second, we have the **script** section, where we add the **JavaScript** code. Finally, we have the **style** section, where we add the **CSS** code:

```
<template></template>

<script>
  export default {
    name: 'nombre-componente',
  }
</script>

<style scoped></style>
```

You are probably familiar with separating HTML from CSS and JavaScript code. Moreover, you may think it is a bad practice to put everything together. However, *Vue.js* applications are divided into **reusable components**, because of this, in each of them we will only include the HTML, JavaScript, and CSS code related to each component. This will avoid us to navigate through a large number of files and will make the applications easier to support and maintain.

In fact, it is not correct to call HTML code to the code in the **template** section, as it is really *Vue.js* code.

The logical part and the component data are added in the section `<script> ... </script>`. The **export** sentence allows to export our components, so that they could be included in other files.

In section `<style> ... </style>` the CSS code of the component will be included. Thanks to the **scoped** attribute, the CSS code will only affect the current component.

1.5 Creating components

Let us create an application to manage people data, so, the first thing we need to do is to **create a component** that shows a **list of people** in the folder `src/components`. We will call this file `TablaPersonas.vue`, since the convention in *Vue.js* is that names follow the “Pascal Case” (XxxxYyyy).

1.5.1 Create a component with *Vue.js*

We will next add the following code, where we will include an HTML table. Besides, you may see we include the div `<div id="tabla-personas">` before the table. The rationale for this is that all the **components** in *Vue.js* must contain a **unique element as root**:

```
<!-- src/components/TablaPersonas.vue -->
<template>
  <div id="tabla-personas">
    <table class="table">
      <thead>
        <tr>
          <th>Nombre</th>
          <th>Apellido</th>
          <th>Email</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Jon</td>
          <td>Nieve</td>
          <td>jon@email.com</td>
        </tr>
        <tr>
          <td>Tyrion</td>
          <td>Lannister</td>
        </tr>
      </tbody>
    </table>
  </div>
</template>
```

```
        <td>tyrion@email.com</td>
      </tr>
      <tr>
        <td>A</td>
        <td>Daenerys</td>
        <td>Targaryen</td>
        <td>daenerys@email.com</td>
      </tr>
    </tbody>
  </table>
</div>
</template>

<script>
  export default {
    name: 'tabla-personas',
  }
</script>

<style scoped></style>
```

Even though the file name is in Pascal Case (`TablaPersonas.vue`), it is common in *Vue.js* to use the name of the component inside of its own file in Kebab Case (`tabla-personas`), so that the same convention as in HTML is maintained.

1.5.2 Add the component to the application

Once we have created and exported the component `TablaPersonas`, let us import it in file `App.vue`. To import a file, we shall write the following sentence in the upper part of the `script` section in file `App.vue`:

```
import TablaPersonas from '@components/TablaPersonas.vue'
```

As you may see, we could use the character `@` to make a reference to folder `src`.

Next, we shall include the component `TablaPersonas` to the property `components`, in such a way that `Vue.js` knows it can use this component. You must include every component you create to this property.

To render the `TablaPersonas` component, it is enough adding `<tabla-personas />`, in Kebab case, to the HTML code. You may see next the complete code you must include in file `App.vue`, replacing the existing code:

Listing 4:

```
<template>
  <div id="app" class="container">
    <div class="row">
      <div class="col-md-12">
        <h1>Personas</h1>
      </div>
    </div>
    <div class="row">
      <div class="col-md-12">
        <tabla-personas />
      </div>
    </div>
  </div>
</template>

<script>
  import TablaPersonas from '@components/TablaPersonas.vue'

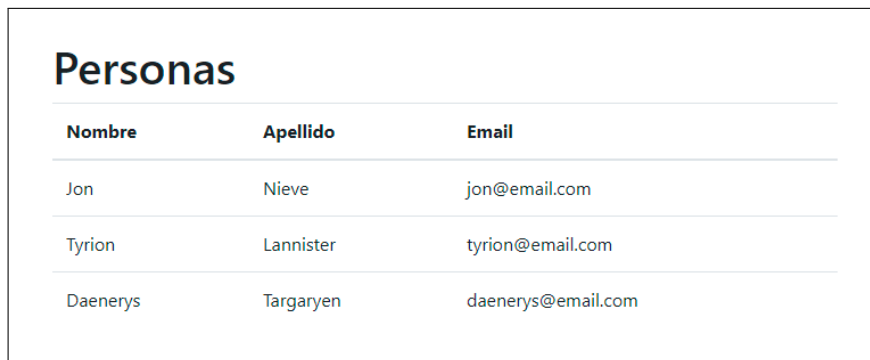
  export default {
    name: 'app',
    components: {
      TablaPersonas,
    },
  }
}
```



```
</script>

<style>
  button {
    background: #009435;
    border: 1px solid #009435;
  }
</style>
```

If you access the project in your browser, you should see the result shown in figure 3.



Nombre	Apellido	Email
Jon	Nieve	jon@email.com
Tyrion	Lannister	tyrion@email.com
Daenerys	Targaryen	daenerys@email.com

Figure 3: Web page created with code 4.

We will replace next the static data for the table with dynamic data.

1.5.3 Add data to the component

In the table of our component we only have static text. Let us replace the data from people by an array of objects that include the data of people. For this, let us add a `data` method to our application in file `App.vue`. This method will return the data from the users. Besides, each user will have an identifier:

```
// ...
import TablaPersonas from '@components/TablaPersonas.vue'
```

```
export default {
  name: 'app',
  components: {
    TablaPersonas,
  },
  data() {
    return {
      personas: [
        {
          id: 1,
          nombre: 'Jon',
          apellido: 'Nieve',
          email: 'jon@email.com',
        },
        {
          id: 2,
          nombre: 'Tyrion',
          apellido: 'Lannister',
          email: 'tyrion@email.com',
        },
        {
          id: 3,
          nombre: 'Daenerys',
          apellido: 'Targaryen',
          email: 'daenerys@email.com',
        },
      ],
    }
  },
}
// ...
```

Once we have included the data in the `App.vue` component, we must pass them to the `TablaPersonas` component. The data will be transferred as properties, and they are included with the syntax `:nombre="datos"`. This means they are dealt in the same way as an HTML **attribute**, except for the fact of having a **colon** `:` before of their name:

```
<tabla-personas :personas="personas" />
```

Alternatively, you may use `v-bind:` instead of `:`, which is the long form to add properties:

```
<tabla-personas v-bind:personas="personas" />
```

We next must accept data for people in the `TablaPersonas` component. For this, we must define the **property** `personas` in the object `props`. The object `props` must include all those properties the component will receive, by setting pairs that include the **name of the property** and their **type**. In our case, the property `personas` is an `Array`:

```
// src/components/TablaPersonas.vue
export default {
  name: 'tabla-personas',
  props: {
    personas: Array,
  },
}
...
```

It is also possible to add properties as a string array, although this is less common:

```
...
export default {
  name: 'tabla-personas',
  props: ['empleados'],
}
// ...
```

1.5.4 Add a loop to the component

Once we have added data to the `TablaPersonas` component, let us create a loop that goes through the data of the people, showing a row of the table in each iteration. For this, we will use the `v-for` attribute, that will let us go through the data of a property, which in our case is the `personas` property:

```
<template>
  <div id="tabla-personas">
    <table class="table">
      <thead>
        <tr>
          <th>Nombre</th>
          <th>Apellido</th>
          <th>Email</th>
        </tr>
      </thead>
      <tbody>
        <tr v-for="persona in personas" :key="persona.id">
          <td>{{ persona.nombre }}</td>
          <td>{{ persona.apellido }}</td>
          <td>{{ persona.email }}</td>
        </tr>
      </tbody>
    </table>
  </div>
</template>
```

If you know check the application in your browser, you will see its appearance has not changed (figure 3), although now it is more dynamic, being even able to load several tables using the same component:

In the next section we will see how to add more people (“personas”) to the table.

1.6 Create forms

Nex, we shall see how you can add new people to the `TablaPersonas` component. For this, we shall create another component that includes a small form.

1.6.1 Create a form with *Vue.js*

Let us create a file `FormularioPersona.vue` in folder `src/componentes`, where we will add a form with an `input` field for the name, another for the surname, and another for the email of the person to be added. Finally, we will also add a button with type `submit` that will allow us to send the data.

With respect to the JavaScript code, we will create the `persona` property as a property that will be returned by the component, and that will include the `name`, the `surname`, and the `email` the person that was added. This would be the code of the component:

```
<template>
  <div id="formulario-persona">
    <form>
      <div class="container">
        <div class="row">
          <div class="col-md-4">
            <div class="form-group">
              <label>Nombre</label>
              <input type="text" class="form-control" />
            </div>
          </div>
          <div class="col-md-4">
            <div class="form-group">
              <label>Apellido</label>
              <input type="text" class="form-control" />
            </div>
          </div>
        </div>
      </div>
    </form>
  </div>
</template>
```

```
<div class="col-md-4">
  <div class="form-group">
    <label>Email</label>
    <input type="email" class="form-control" />
  </div>
</div>
</div>
<div class="row">
  <div class="col-md-4">
    <div class="form-group">
      <button class="btn btn-primary">Agnadir persona</button>
      ↪ >
    </div>
  </div>
</div>
</div>
</form>
</div>
</template>

<script>
export default {
  name: 'formulario-persona',
  data() {
    return {
      persona: {
        nombre: '',
        email: '',
        apellido: '',
      },
    }
  },
}
```

```
    }  
</script>  
  
<style scoped>  
  form {  
    margin-bottom: 2rem;  
  }  
</style>
```

Note that we have also added a margin in the form with CSS in the `style` section.

1.6.2 Add a form to the application

We next are going to edit the file `App.vue` and add the component `FormularioPersona` we just created:

```
<template>  
  <div id="app" class="container">  
    <div class="row">  
      <div class="col-md-12">  
        <h1>Personas</h1>  
      </div>  
    </div>  
    <div class="row">  
      <div class="col-md-12">  
        <formulario-persona /> <!-- <<<<<< -->  
        <tabla-personas :personas="personas" />  
      </div>  
    </div>  
  </div>  
</template>  
<script>  
  import TablaPersonas from '@components/TablaPersonas.vue'
```

```
import FormularioPersona from '@components/FormularioPersona.  
  ↪ vue' // <<<<<<<  
  
export default {  
  name: 'app',  
  components: {  
    TablaPersonas,  
    FormularioPersona, // <<<<<<<  
  },  
  data: {  
    // ...  
  },  
}  
</script>
```

If you now access the project in your browser, you should see the form is shown above the table:

Personas

Nombre

Apellido

Email

Añadir persona

Nombre	Apellido	Email
Jon	Nieve	jon@email.com
Tyrion	Lannister	tyrion@email.com
Daenerys	Targaryen	daenerys@email.com

Figure 4: Web page created with the code 1.8.1.

1.6.3 Link the fields of the form with their state

Next we should obtain the values that will be introduced in the form through JavaScript, so that we could assign their values to the state of the component.

For this, we shall use the attribute `v-model`, that will link the value of the fields with their respective state variables, which are defined in the `persona` property of the `return` sentence in the component (see `src/components/FormularioPersona.vue`):

Listing 5: Code that implements a form in *Vue.js*. Used classes (`class`) are defined in `bootstrap` and improve the overall aesthetic of the application, even if they do not add any functionality by themselves.

```
<template>
  <div id="formulario-persona">
    <form>
      <div class="container">
        <div class="row">
          <div class="col-md-4">
            <div class="form-group">
              <label>Nombre</label>
              <input v-model="persona.nombre" type="text" class="
                ↪ form-control" />
            </div>
          </div>
          <div class="col-md-4">
            <div class="form-group">
              <label>Apellido</label>
              <input v-model="persona.apellido" type="text" class="
                ↪ form-control" />
            </div>
          </div>
          <div class="col-md-4">
            <div class="form-group">
```

```
        <label>Email</label>
        <input v-model="persona.email" type="email" class="
            ↪ form-control" />
    </div>
</div>
</div>
<div class="row">
    <div class="col-md-4">
        <div class="form-group">
            <button class="btn btn-primary">Añadir persona</
            ↪ button>
        </div>
    </div>
</div>
</div>
</div>
</form>
</div>
</template>
```

If you check the result in your browser and go to the **DevTools** of *Vue.js*, you will see how the state of the component changes every time you modify a field from the form.

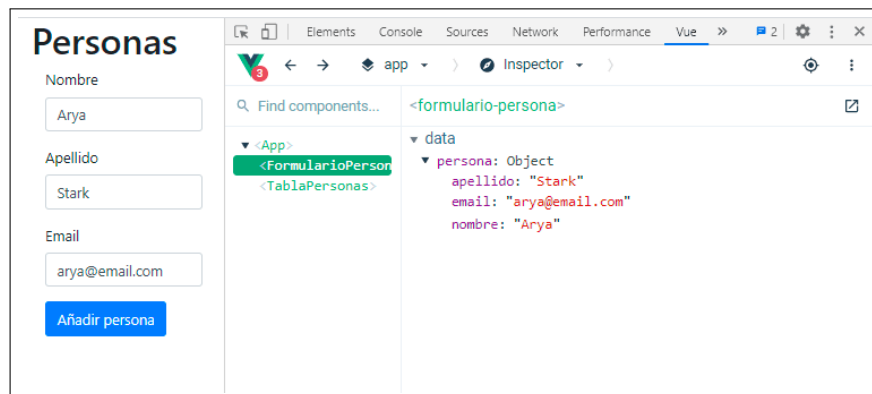


Figure 5: Visualizing the `personas` array with DevTools.

However, we still need to send the data to our main component, which is the **App** application, so that its state is also modified, by adding the data of each new person to the list of people.

1.6.4 Add a sending method to the form

Let us add an **event**, also known as an **event listener**, to the form. In particular, we will add an **onSubmit** event so that a method is executed whenever a click is made in the submit button. For this, we will use the **@submit** attribute, which is the short form of the **v-on:submit** attribute, being both of them equivalent.

In general, all these event listeners in *Vue.js* are included with prefixes **@** or **v-on:**. Because of this, we could detect a click with **@click** or **v-on:click**, and a hover event with **@mouseover** or **v-on:mouseover**.

Moreover, we do not want the server being responsible to update the page with the form, but it should be *Vue.js* the one doing this, so we must run the **event.preventDefault** method that disables the default behavior which is to connect to the server. For this, the event **@submit** provides a modifier **prevent**, that is equivalent to run the **event.preventDefault()** method inside of the associated function to the submit event.

Let us bind the **enviarFormulario** method to the **@submit** event of the form:

```
<form @submit.prevent="enviarFormulario">
  ...
</form>
```

Now we shall include the method **enviarFormulario** to the component. Methods from *Vue.js* components are added inside the **methods** property, that we will also create as follows:

```
export default {
  name: 'formulario-persona',
  data() {
    return {
      persona: {
```

```
        nombre: '',
        email: '',
        apellido: '',
      },
    },
    methods: {
      enviarFormulario() {
        console.log('Works!');
      },
    },
  }
}
```

If you test the code and send the form, you will see the following text in the console **Works!**.

1.6.5 Send events from the form to the application

We now need to send the data of the person we have added to our application **App**, for this, we will use the **\$emit** method in the **enviarFormulario** function. The method **\$emit** sends the **name of the event** we define and the **data** we want to the component where the current component has been rendered.

In our case, we shall send the **persona** property and an event we will call **add-persona**:

```
// ...
enviarFormulario() {
  this.$emit('add-persona', this.persona);
}
// ...
```

It is important that you consider that the **name of the events** must always follow the **kebab-case syntax**. The emitted event will allow to start the method that receives the data in the **App** application.

1.6.6 Receive events from the table in the application

The `FormularioPersona` component sends data through the `add-persona` event. Now we must capture data in the application. For this, we shall include the `@add-persona` property in the `formulario-persona` tag through which we include the component. In it, we will associate a new method to the event, that we will call as `agregarPersona`:

```
<formulario-persona @add-persona="agregarPersona" />
```

Next, we create a method called `agregarPersona` in the `methods` property of the file `App.vue`, that will modify the `personas` array that is included, adding a new object to itself:

```
methods: {  
  agregarPersona(persona) {  
    this.personas = [...this.personas, persona];  
  }  
}
```

We have used the propagation operator `...`, useful to combine objects and arrays, so that a new array is created including the old elements from `personas` arrays together with the new person introduced using the form.

If you now run the DevTools and send the form, you will see a new element is added in the `personas` array:

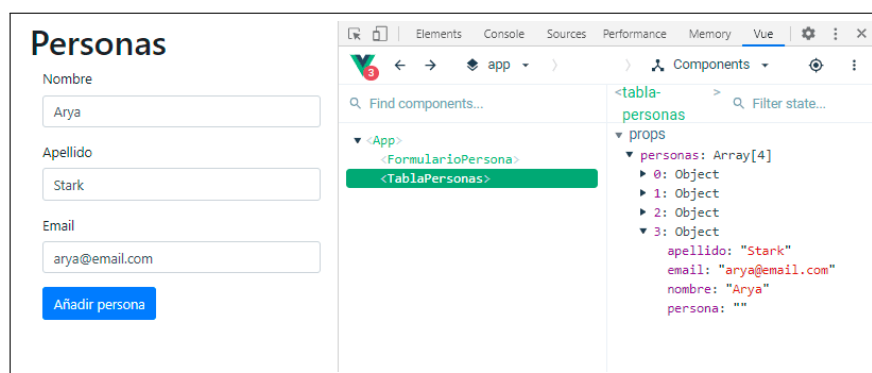


Figure 6: DevTools used to see the `personas` array in the browser.

However, it is important to assign a unique ID to the element we just created. Usually, we would add the created person in a database, which would return the person together with its new ID. But right now we will limit ourselves to generate an ID based on the ID of the previous created element:

```
methods: {
  agregarPersona(persona) {
    let id = 0;

    if (this.personas.length > 0) {
      id = this.personas[this.personas.length - 1].id + 1;
    }
    this.personas= [...this.personas, { ...persona, id}];
  }
}
```

What we have done is to increment the value of the ID of the last created element in one unit, or leave it in 0 if there are no elements. We then add the person in the array, and assign it the generated id.

1.7 Validations with *Vue.js*

Our form works, and it may be even necessary to enter a valid email address thanks to HTML validation from many browsers. However, we still need to **show a notification** when a user is added correctly, **reset the focus** on the first element of the form and **empty the data fields**. Besides, we must be sure all the fields have been filled with **valid data**, showing an **error message** otherwise.

1.7.1 Computed properties in *Vue.js*

In *Vue.js*, data is usually validated through **computed properties**, which are functions that are automatically executed when the state of a property is modified. In this way we avoid overloading the HTML code of the component. These computed

properties are added inside the `computed` property, that we will add right after the `methods` property of the `FormularioPersona` component:

```
// ...
computed: {
  nombreInvalido() {
    return this.persona.nombre.length < 1;
  },
  apellidoInvalido() {
    return this.persona.apellido.length < 1;
  },
  emailInvalido() {
    return this.persona.email.length < 1;
  },
},
// ...
```

We have added a very simple validation that checks there is something introduced in each field.

Next, we will add a state `estado` in the `FormularioPersona` component that will be called `procesando`. This variable will check if the form is being submitted currently or not.

We will also add the `error` and `correcto` variables. By default both variables are set to `false` and no message is shown but the became `true` is an error occurs (`error`) or the data is sent successfully (`procesado`)

```
// ...
data() {
  return {
    procesando: false,
    correcto: false,
    error: false,
    persona: {
      nombre: '',
```

```
        apellido: '',
        email: '',
    }
}
}
// ...
```

Next, we should modify the `enviarFormulario` method to establish the value of the estate variable called `procesando` as `true` when the form is submitted, and as `false` when the result is obtained. Depending on the obtained result, the value of the `error` variable will be set as `true` if there was any error, at the same time, the value of the `correcto` variable would be set to `true` if the data was submitted correctly:

```
// ...
methods: {
    enviarFormulario() {
        this.procesando = true;
        this.resetEstado();

        // Comprobamos la presencia de errores
        if (this.nombreInvalido || this.apellidoInvalido || this.
            ↪ emailInvalido) {
            this.error = true;
            return;
        }

        this.$emit('add-persona', this.persona);

        this.error = false;
        this.correcto = true;
        this.procesando = false;
    }
}
```



```
// Restablecemos el valor de la variables
this.persona= {
  nombre: '',
  apellido: '',
  email: '',
}
},
resetEstado() {
  this.correcto = false;
  this.error = false;
}
}
// ...
```

As you may see, we have also include the `resetEstado` method to reset some state variables.

1.7.2 Conditional sentences with *Vue.js*

We are going to modify next the HTML code in our form. As you could checked before when using the DevTools, *Vue.js* renders again each component every time the state of a component is modified. In this way, the events are managed more easily.

Having said this, let us configure the form in such a way that a CSS class called `has-error` is added to its fields depending on whether they failed or not. We shall also add a potential error message at the end of the form.

Listing 6: Code implementing the form to add people. Note the following sentence `:class="{ 'is-invalid': procesando && nombreInvalido }"` adds the class `is-valid` if `procesando` and `nombreInvalido` are true. `is-valid` is defined in bootstrap.

```
<form @submit.prevent="enviarFormulario">
  <div class="container">
```

```
<div class="row">
  <div class="col-md-4">
    <div class="form-group">
      <label>Nombre</label>
      <input
        v-model="persona.nombre"
        type="text"
        class="form-control"
        :class="{ 'is-invalid': procesando && nombreInvalido }"
        @focus="resetEstado"
      />
    </div>
  </div>
  <div class="col-md-4">
    <div class="form-group">
      <label>Apellido</label>
      <input
        v-model="persona.apellido"
        type="text"
        class="form-control"
        :class="{ 'is-invalid': procesando && apellidoInvalido
          ↪ }"
        @focus="resetEstado"
      />
    </div>
  </div>
  <div class="col-md-4">
    <div class="form-group">
      <label>Email</label>
      <input
        v-model="persona.email"
        type="email"
```

```

        class="form-control"
        :class="{ 'is-invalid': procesando && emailInvalido }"
        @focus="resetEstado" />
    </div>
</div>
</div>
<div class="row">
    <div class="col-md-4">
        <div class="form-group">
            <button class="btn btn-primary">Agnadir persona</button>
        </div>
    </div>
</div>
</div>
<div class="container">
    <div class="row">
        <div class="col-md-12">
            <div v-if="error && procesando" class="alert alert-danger"
                ↪ role="alert">
                Debes rellenar todos los campos!
            </div>
            <div v-if="correcto" class="alert alert-success" role="
                ↪ alert">
                La persona ha sido agregada correctamente!
            </div>
        </div>
    </div>
</div>
</form>

```

As you have seen, we have used the `:class` attribute to define the classes, as we cannot use attributes that exist in HTML. The reason behind not using the `class`

attribute is that it only accepts text strings as value.

We have also included the `@focus` event to the `input` fields so they are reset every time they are selected.

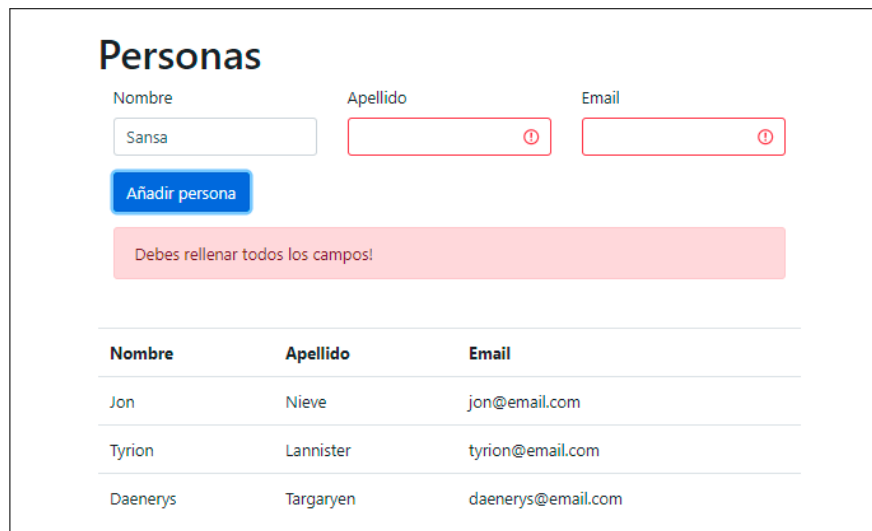
To show the error messages we have used the **conditional sentence** `v-if` from *Vue.js*, which will make the element in which it is included visible only if the specified condition is evaluated as `true`.

The error message will only be shown if the value of the `error` variable is `true` and the form is being processed, that is, `procesando` is `true`. The success message will only be shown when the value of the `correcto` variable is `true`.

It is also possible to use `v-else` or `v-else-if` sentences, which work exactly the same as `else` and `else if` sentences from JavaScript, respectively.

For more information about the conditional rendering, check the *Vue.js* documentation (<https://v3.vuejs.org/guide/conditional.html>).

After adding the error messages, check again the browser to see the result. You will see that, when you forget to fill any field, an error message is shown:



Personas

Nombre: Apellido: Email:

Debes rellenar todos los campos!

Nombre	Apellido	Email
Jon	Nieve	jon@email.com
Tyrion	Lannister	tyrion@email.com
Daenerys	Targaryen	daenerys@email.com

In the same way, a success message is shown when the person is correctly added to the list:

Personas

Nombre

Apellido

Email

Añadir persona

La persona ha sido agregada correctamente!

Nombre	Apellido	Email
Jon	Nieve	jon@email.com
Tyrion	Lannister	tyrion@email.com
Daenerys	Targaryen	daenerys@email.com
Sansa	Stark	sansa@email.com

1.7.3 References with *Vue.js*

When you submit a form, usually both the cursor and the focus, because of web accessibility, are located in the first element of the form, which in our case is the `nombre` field. For this, we could use the references of *Vue.js* (<https://v3.vuejs.org/api/refs-api.html>), since they allow to make reference to the elements that include them. To add a reference you should use the `ref` attribute.

In the following example, we will add a reference to the `nombre` field:

```
<input
  ref="nombre"
  v-model="persona.nombre"
  type="text"
  class="form-control"
  :class="{ 'is-invalid': procesando && nombreInvalido }"
  @focus="resetEstado"
  @keypress="resetEstado"
/>
```

Finally, after submitting the form, we will use the `focus` method that includes the references so that the cursor is located in the `nombre` field:

```
this.$emit('add-persona', this.persona);  
this.$refs.nombre.focus();
```

We have also added a `@keypress` event to the `nombre` field so that the state is reset whenever a key is pressed, as the focus will be in another field then.

If you now try to add a person, you will see the cursor is located in the first element.

1.8 Remove elements with *Vue.js*

The form is already working correctly, but let us add also the option to remove people.

1.8.1 Add a remove button to the table

For this, let us add one more column to the table of the `TablaPersonas` component, that will include the button that will allow to remove each row:

```
<template>  
  <div id="tabla-personas">  
    <table class="table">  
      <thead>  
        <tr>  
          <th>Nombre</th>  
          <th>Apellido</th>  
          <th>Email</th>  
          <th>Acciones</th>  
        </tr>  
      </thead>  
      <tbody>  
        <tr v-for="persona in personas" :key="persona.id">  
          <td>{{ persona.nombre }}</td>
```

```

        <td>{{ persona.apellido }}</td>
        <td>{{ persona.email }}</td>
        <td>
        <!-- 🗑️ is the wastebasket icon. Icons available
            ↪ at https://codepoints.net -->
            <button class="btn btn-danger">🗑️ Eliminar</
            ↪ button>
        </td>
    </tr>
</tbody>
</table>
</div>
</template>

```

1.8.2 Emit a remove event from the table

Now, as we did for the form, we must emit an event that we will call `eliminarPersona`. This event will send the `id` of the person to be removed to the parent component, which in this case is the `App.vue` application:

```

<button class="btn btn-danger" @click="$emit('delete-persona',
    ↪ persona.id)">🗑️ Eliminar</button>

```

1.8.3 Receive the remove event in the application

Now, in the `App.vue` application you must add an action that will execute a method that will remove the corresponding `persona` with respect to the given `id`:

```

<tabla-personas :personas="personas" @delete-persona="eliminarPersona
    ↪ " />

```

We now define the `eliminarPersona` method right below the `agregarPersona` method we created previously:

```
eliminarPersona(id) {  
  this.personas = this.personas.filter(  
    persona => persona.id !== id  
  );  
}
```

We have used the `filter` method, that will keep those elements in the `personas` array whose `id` do not match the provided one. With this, if you check the browser, you will see people are removed from the table when clicking on the button of their respective row:

Nombre	Apellido	Email	Acciones
Jon	Nieve	jon@email.com	Eliminar
Tyrion	Lannister	tyrion@email.com	Eliminar

1.8.4 Add an informative message

To make the application more usable, let us add a message that will be shown right before the `table` opening label whenever it does not contain any person:

```
<div v-if="!personas.length" class="alert alert-info" role="alert">  
  No se han agregado personas  
</div>
```

This is the message that will be shown when there are no more users:

Personas

Nombre Apellido Email

Añadir persona

No se han agregado personas

Nombre	Apellido	Email	Acciones
--------	----------	-------	----------

1.9 Edit elements with *Vue.js*

Now that we can remove elements, it would be good if we could edit them, which is what we will address in this section. Let us add the possibility to edit the elements in the table itself, so that we do not need any additional components.

1.9.1 Add an edit button to the table

Let us start adding an edit button to the table of the `TablaPersonas` component, right next to the remove button:

```
<button class="btn btn-info ml-2" @click="editarPersona(persona)">&#x1F58A; Editar</button>
```

The button will call the `editarPersona` method, to which the selected person will be passed.

1.9.2 Add an edit method to the table

Now, we shall add the `editarPersona` method to our list of methods, inside the `methods` property of our component:

```
//TablaPersonas.vue
methods: {
```

```
editarPersona(persona) {  
  this.personaEditada = Object.assign({}, persona);  
  this.editando = persona.id;  
},  
}
```

What we have done was to store the original data of the person that is being edited currently in the `personaEditada` object, in such a way that we could recover the data if the edit is canceled. Moreover, we have changed the value of the state variable `editando` which, obviously, we must add to our component:

```
data() {  
  return {  
    editando: null,  
  }  
},
```

1.9.3 Add edit fields to the table

Now it is necessary to check the value of the `editando` variable in each row, showing input fields instead of the values of the person in those rows where the variable is active:

```
<template>  
  <div id="tabla-personas">  
    <div v-if="!personas.length" class="alert alert-info" role="alert">  
      No se han agregado personas  
    </div>  
    <table class="table">  
      <thead>  
        <tr>  
          <th>Nombre</th>  
          <th>Apellido</th>
```

```
<th>Email</th>
<th>Acciones</th>
</tr>
</thead>
<tbody>
<tr v-for="persona in personas" :key="persona.id">
  <td v-if="editando === persona.id">
    <input type="text" class="form-control" v-model="
      ↪ persona.nombre" />
  </td>
  <td v-else>
    {{ persona.nombre}}
  </td>
  <td v-if="editando === persona.id">
    <input type="text" class="form-control" v-model="
      ↪ persona.apellido" />
  </td>
  <td v-else>
    {{ persona.apellido}}
  </td>
  <td v-if="editando === persona.id">
    <input type="email" class="form-control" v-model="
      ↪ persona.email" />
  </td>
  <td v-else>
    {{ persona.email}}
  </td>
  <td>
    <button class="btn btn-info" @click="editarPersona(
      ↪ persona)">&#x1F58A; Editar</button>
    <button class="btn btn-danger ml-2" @click="$emit('
      ↪ delete-persona', persona.id)">&#x1F5D1;
```

```

        ↪ Eliminar</button>

      </td>
    </tr>
  </tbody>
</table>
</div>
</template>

```

Now, if you test the application and edit a row, you will see the edit fields are shown in the corresponding row to the person you are editing:

Personas

Nombre
Apellido
Email

Añadir persona

Nombre	Apellido	Email	Acciones
<input type="text"/>	Nieve	jon@email.com	<div>Eliminar</div> <div>Editar</div>
Tyion	Lannister	tyion@email.com	<div>Eliminar</div> <div>Editar</div>
Daenerys	Targaryen	daenerys@email.com	<div>Eliminar</div> <div>Editar</div>

1.9.4 Add a save button to the table

However, we still need a **save button** and another that allows to **cancel the edit state**. These buttons will only be shown in the row that is being edited:

```

<td v-if="editando === persona.id">
  <button class="btn btn-success" @click="guardarPersona(persona
    ↪ )">&#x1F5AB; Guardar</button>
  <button class="btn btn-secondary ml-2" @click="cancelarEdicion
    ↪ (persona)">&#x1F5D9; Cancelar</button>
</td>
<td v-else>
  <button class="btn btn-info" @click="editarPersona(persona)">&#
    ↪ x1F58A; Editar</button>

```

```
<button class="btn btn-danger ml-2" @click="$emit('delete-
  ↪ persona', persona.id)">&#x1F5D1; Eliminar</button>
</td>
```

As you observe, we make reference to the `guardarPersona` method, that we still need to add. At the moment, this would be the result when you click on the edit button in any row:

Nombre	Apellido	Email	Acciones
Jon	Nieve	jon@email.com	Guardar Cancelar
Tyron	Lannister	tyron@email.com	Eliminar Editar
Daenerys	Targaryen	daenerys@email.com	Eliminar Editar

1.9.5 Emit the save event

Let us add the `guardarPersona` method to the list of methods. This method will send a save event to the `App.vue` application, that will be in charge of **updating** the data of the person we have edited:

```
guardarPersona(persona) {
  if (!persona.nombre.length || !persona.apellido.length || !
    ↪ persona.email.length) {
    return;
  }
  this.$emit('actualizar-persona', persona.id, persona);
  this.editando = null;
}
```

1.9.6 Add a method that cancels the edit

Let us add also the `cancelarEdicion` method that we have also used in the previous section, and which will let us **cancel** the edit status of a person:

```
cancelarEdicion(persona) {  
    Object.assign(persona, this.personaEditada);  
    this.editando = null;  
}
```

As you can see, when we cancel the edit of a person, we recover its original value, stored in the `personaEditada` object.

1.9.7 Receive the update event in the application

We still need to modify the code of the `App.vue` application so that it receives the save event and updates the data of the corresponding person. We first add the `actualizar-persona` event:

```
<tabla-personas  
  :personas="personas"  
  @delete-persona="eliminarPersona"  
  @actualizar-persona="actualizarPersona"  
>
```

Now we add the `actualizarPersona` method to the list of methods of the application, right after the `eliminarPersona` method:

```
actualizarPersona(id, personaActualizada) {  
    this.personas = this.personas.map(persona =>  
        persona.id === id ? personaActualizada : persona  
    )  
}
```

In the previous method we used the loop “=>” to go through the `personas` array, updating those whose `id` match with the one from the person we want to update. If you now test the application, you will see the changes are stored correctly

1.10 Build & Deploy the *Vue.js* application

We have created an application that works in our computer, but ideally it should work in an external server, so that anyone could access it from anywhere in the world. To do this, we will deploy the application in **render.com**.

If you have your code in a **GitHub** repository jump to the subsection 1.10.1, otherwise continue reading.

Initialize a repository in the project root folder by running `git init`. Then, go to your GitHub account and create a new repository. It will be possible to access this repository through an identifier similar to `git@github.com:usuario/repositorio`, where `usuario` is your GitHub username and `repositorio` the name of the repository.

Next, run the following commands from the root directory of the project to upload your code to GitHub, replacing `usuario` by your username:

```
git add .
git remote add origin https://github.com/usuario/tutorial-vue
git commit -m "first commit"
git branch -M main
git push -u origin main
# where "main" is the name of the git branch
# in most cases this name will be either "main" or "master"
```

With this, our code should be in GitHub.

1.10.1 Building the *Vue.js* application

The application you are using is compiled on-the-fly in memory. As it is a development version, some error checks and other tests are made which provoke this is not a proper version for production. To compile the files and create a production version, you must close the current node process by pressing `CTRL+C` or `CMD+C` and run the command:

```
npm run build
```

1.10.2 Deploying the *Vue.js* application

Let us deploy the application in **render.com**. First check that it is possible to create a “compiled” version of the application by running the command:

```
npm run build
```

Then, create a “Static Site” in **render.com** and give it permission to access your repository in **GitHub** (<https://render.com/docs/github>). Use the following values to create the repository:

```
Build Command: npm run build  
Publish Directory: dist
```

If everything worked as expected, you already have your application deployed.

1.11 Summary of the work done until now

In this tutorial we have created a CRUD application with *Vue.js*. You have learned to create components, methods, and forms. In addition, you have managed states and events, together with conditional expressions. Finally, you have also learned to put an application in production.

Currently, this application only works in your browser. If you refresh your browser, the data will be lost. What we will do next is to connect the application with an external server, so that the data will be stored persistently.

In the next tutorial, we will create an application that supports viewing (GET), creating (POST), updating (PUT), and removing (DELETE) the data of different entities.