

UNIVERSIDAD AUTÓNOMA DE MADRID

DEPARTAMENTO DE INFORMÁTICA

Proyecto de Sistemas Informáticos

Práctica - 2.1

Roberto MARABINI
Alejandro BELLOGÍN

Registro de Cambios

Versión ¹	Fecha	Autor	Descripción
1.0	10.10.2022	RM	Primera versión.
1.1	5.12.2022	RM	Renumerar práctica 3 ->2
1.2	12.12.2022	AB	Traducción al inglés
1.3	26.1.2023	RM	Revisión previa a subir la guía a moodle
1.4	22.2.2023	RM	Clarificar la descripción de los diferentes ficheros que crea un proyecto <i>Vue.js</i>

¹La asignación de versiones se realizan mediante 2 números *X.Y*. Cambios en *Y* indican aclaraciones, descripciones más detalladas de algún punto o traducciones. Cambios en *X* indican modificaciones más profundas que o bien varían el material suministrado o el contenido de la práctica.

Índice

1. Usando <i>Vue.js</i>	4
1.1. Usando <i>Vue.js</i> en un fichero estático	4
1.2. Uso de <i>Vue.js</i> creando un proyecto	8
1.3. Entorno de desarrollo de <i>Vue.js</i>	11
1.3.1. Resaltado de sintaxis en <i>Vue.js</i>	11
1.3.2. Instalación de las DevTools de <i>Vue.js</i>	11
1.3.3. Incluye un framework CSS	11
1.4. Estructura de un archivo <i>Vue.js</i>	12
1.5. Creación de componentes	13
1.5.1. Crea un componente con <i>Vue.js</i>	13
1.5.2. Agrega el componente a la aplicación	14
1.5.3. Agrega datos al componente	16
1.5.4. Agrega un bucle al componente	19
1.6. Creación de formularios	20
1.6.1. Crea un formulario con <i>Vue.js</i>	20
1.6.2. Agrega el formulario a la aplicación	22
1.6.3. Enlaza los campos del formulario con su estado	24
1.6.4. Agrega un método de envío al formulario	26
1.6.5. Emite eventos del formulario a la aplicación	28
1.6.6. Recibe eventos de la tabla en la aplicación	28
1.7. Validaciones con <i>Vue.js</i>	30
1.7.1. Propiedades computadas de <i>Vue.js</i>	30
1.7.2. Sentencias condicionales con <i>Vue.js</i>	33
1.7.3. Referencias con <i>Vue.js</i>	37
1.8. Elimina elementos con <i>Vue.js</i>	38
1.8.1. Agrega un botón de borrado a la tabla	38
1.8.2. Emite un evento de borrado desde la tabla	39
1.8.3. Recibe el evento de borrado en la aplicación	39
1.8.4. Agrega un mensaje informativo	40
1.9. Edita elementos con <i>Vue.js</i>	41

1.9.1. Agrega un botón de edición a la tabla	41
1.9.2. Agrega un método de edición a la tabla	41
1.9.3. Agrega campos de edición a la tabla	42
1.9.4. Agrega un botón de guardado a la tabla	44
1.9.5. Emite el evento de guardado	45
1.9.6. Agrega un método que cancele la edición	46
1.9.7. Recibe el evento de actualización en la aplicación	46
1.10. Build & Deploy de la aplicación <i>Vue.js</i>	47
1.10.1. Build de la aplicación <i>Vue.js</i>	48
1.10.2. Deploy de la aplicación <i>Vue.js</i>	48
1.11. Resumen del trabajo realizado hasta el momento	48

Aviso

El proyecto descrito a continuación esta basado en los tutoriales: “Tutorial de introducción a *Vue.js* 3” (<https://www.neoguias.com/tutorial-vue/>) y “Cómo crear una aplicación REST con *Vue.js*” (<https://www.neoguias.com/tutorial-rest-vue/>).

En <https://worldline.github.io/vuejs-training/> podéis encontrar una introducción a *Vue.js* que complementa la información contenida en este documento.

1. Usando *Vue.js*

Puedes usar *Vue.js* de diversas formas. Primero veremos cómo inyectar código *Vue.js* en un archivo HTML usando un fichero estático y más adelante crearemos un proyecto.

1.1. Usando *Vue.js* en un fichero estático

Esta es la opción más sencilla, ya que basta con cargar un fichero javascript en la sección **head** de un archivo HTML y colocar las directivas *Vue.js* dentro del **div** que se identifica mediante el atributo **id**, cuyo valor será **app**. En el siguiente archivo HTML vemos cómo agregarlo:

```
<!--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>
```

Tal y como ves, hemos incluido la versión 3.2 de *Vue.js*. Para incluir otras versiones, basta con que cambies la porción `@3.2` de la URL y especifiques el número de versión que quieres agregar en su lugar. En caso de querer incluir una versión de desarrollo de *Vue.js* y no de producción, tendríamos que incluir este código en su lugar:

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

En general es mejor no usar la versión de producción durante el desarrollo por que los mensajes de error son menos detallados. A continuación vamos a crear la aplicación más sencilla posible que puedes crear con *Vue.js*, que no es otra cosa que un Hola Mundo.

Para ello agregaremos un script justo antes del cierre de la etiqueta `body`. En el script crearemos una nueva instancia de una aplicación *Vue.js* ejecutando la sentencia `app = Vue.createApp({ ... })`, que recibirá un objeto como parámetro:

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

```
</script>
```

El objeto que hemos pasado al constructor de la clase *Vue.js* incluye el método `data`, que debe devolver aquellos datos con los que queramos inicializar nuestra aplicación. En nuestro caso será únicamente la variable `greeting`.

Para inicializar la aplicación *Vue.js* que hemos creado, debemos usar el método `mount`, que recibirá el identificador del elemento en el que queremos renderizar la aplicación, que en nuestro caso es el `div #app`:

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

Seguidamente, renderizaremos la propiedad `greeting` en el interior del `div app` usando la sintaxis `{{ greeting }}`:

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

Este sería el código completo de esta primera aplicación introductoria:

Listado 1: Código completo de la primera aplicación.

```
<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>
```

Tal y como ves, hemos usado algo similar a JavaScript con una sintaxis un poco peculiar que, en condiciones normales, provocarían errores. Sin embargo, cuando ejecutas el código, puedes comprobar que el resultado es la figura 1:

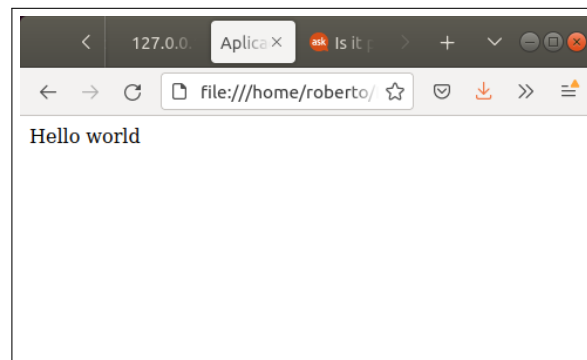


Figura 1: Página web creada por el código 1.

no existen errores porque el código *Vue.js* se traduce a código JavaScript mediante un compilador llamado Babel (<https://babeljs.io>) aunque no lo percibas. Esto ocurrirá cada vez que se ejecute la aplicación. Sin embargo, no es habitual agregar *Vue.js* de esta forma, por lo que la compilación, tal y como verás, solamente se suele realizar cuando desarrollas la aplicación y no cuando la utilizan los usuarios en producción. Hemos utilizado la palabra “compilador” de forma imprecisa puesto que Babel no produce código máquina sino que usa como entrada código *Vue.js* y como salida JavaScript, esto es, la entrada y la salida son lenguajes del mismo nivel

pero uno de ellos (**JavaScript**) es interpretable directamente por los navegadores. El termino correcto para esta operación no es “compilar” sino “transpilar”.

Es totalmente normal pensar que todo esto es excesivamente complicado cuando con HTML puro se puede mostrar la frase “hello world” fácilmente. No te preocupes, cuando finalices este tutorial verás las posibilidades que te abre *Vue.js*.

1.2. Uso de *Vue.js* creando un proyecto

No es habitual embeber código *Vue.js* directamente en una página HTML por el inconveniente de que el código *Vue.js* se debe transpilar cada vez que un usuario carga la aplicación. Lo que se suele hacer es crear un proyecto *Vue.js* y transpilar el código *Vue.js* en JavaScript antes de subirlo al servidor, de forma que los navegadores puedan entenderlo sin realizar ninguna operación extra.

Vue.js cuenta con un interfaz de línea de comandos que te ayudará a desarrollar las aplicaciones. Vamos a ver cómo instalarla. Para ello debes abrir un terminal y ejecutar el comando que ves a continuación:

```
npm install vue@3.2.27
```

Para crear una nueva aplicación, desplázate al directorio en el que quieres crear tu aplicación y ejecuta el siguiente comando:

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

y responde no a todas las preguntas (en el siguiente proyecto discutiremos alguna de las opciones en detalle)

```
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
```

Tras unos segundos, la aplicación estará ya creada. Seguidamente, desplázate al directorio que se ha creado:

```
cd tutorial-vue
```

Luego ejecuta el comando siguiente para iniciar el **servidor de desarrollo**, que por defecto estará en el puerto **3000**:

```
npm install  
npm run dev
```

Una vez creada la aplicación, abre tu navegador y accede a <http://localhost:3000/> para ver la aplicación, que incluye por defecto una página similar a la figura 2:

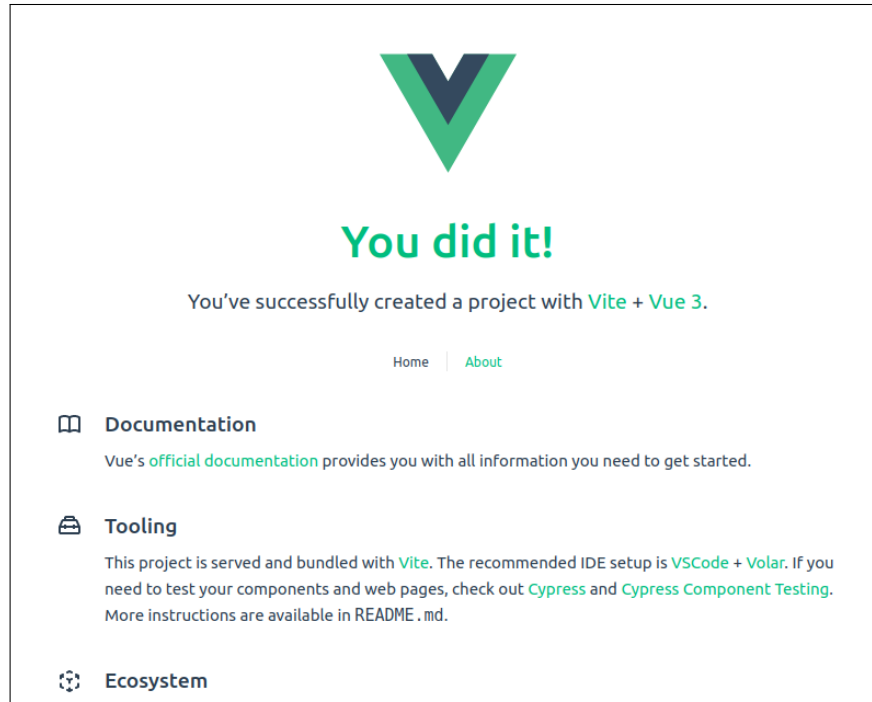


Figura 2: Página web creada por defecto por *Vue.js* 3.2.

En la raíz del proyecto se encuentra el fichero `index.html` que es la entrada a la aplicación. El directorio en donde estará tu código será el directorio `src`, en

donde encontrarás el archivo `main.js`, que es llamado desde `index.html` y monta las diferentes aplicaciones de las que consta tu proyecto de *Vue.js*.

El contenido del fichero `main.js` (vease listado 2) depende de las opciones elegidas y siempre debe importar: (a) el método `createApp` y (b) el código principal de la aplicación, localizado en el archivo `App.vue`. La aplicación se crea con el comando `createApp(App)` y se “renderiza” en un elemento HTML de nuestra página, que en este caso es el `div #app`, con el comando `mount('#app')`:

Listado 2: Ejemplo de fichero `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"
```

Si ahora accedes al archivo `App.vue` verás que es relativamente complicado. Conceptualmente lo podéis simplificar a las líneas:

Listado 3: Extracto del fichero `App.vue`.

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

como en el caso anterior sólo el código que se encuentre dentro de la `div id=app` tendrá acceso a las variables definidas por *Vue.js*.

1.3. Entorno de desarrollo de *Vue.js*

Tras instalar *Vue.js*, ya podríamos comenzar a programar la aplicación. Sin embargo, vamos a configurar primero una serie de herramientas que nos serán de gran ayuda durante el proceso.

1.3.1. Resaltado de sintaxis en *Vue.js*

Puedes usar cualquier IDE, aunque te recomendamos que uses algún plugin de **resaltado de sintaxis** que además también pueda formatear el código. Si usas **VS Code**, instala el plugin **Vetur**.

1.3.2. Instalación de las DevTools de *Vue.js*

Antes de continuar, es recomendable que instales el plugin **vue.js DevTools** en tu navegador. Este plugin te da acceso a las variables *Vue.js* que el navegador almacena localmente.

1.3.3. Incluye un framework CSS

Te recomendamos que uses algún framework de CSS como pueda ser **Bootstrap**. Si nunca has usado Bootstrap, puedes consultar el tutorial de `/bootstrap/introducción` a Bootstrap.

Para incluir Bootstrap en el proyecto, abre el archivo `src/main.js` y agrega las siguientes líneas al final del mismo (véase listado 2)

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

también tendrás que instalar bootstrap usando **node** tecleando en la 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
```

Por favor, instala las versiones que te sugerimos para que no aparezcan problemas de compatibilidad.

1.4. Estructura de un archivo *Vue.js*

Los archivos *Vue.js* se dividen en tres secciones muy diferenciadas. Por un lado tenemos la sección **template**, en donde agregaremos el código **HTML** de la aplicación. Por otro lado, tenemos la sección **script**, en donde agregaremos el código **JavaScript**. Finalmente, tenemos la sección **style**, en donde agregaremos el código **CSS**:

```
<template></template>

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

<style scoped></style>
```

Seguramente estés habituado a separar el código HTML del código CSS y JavaScript. Además crearás que es una mala práctica unirlos todo. Sin embargo, las aplicaciones *Vue.js* se dividen en **componentes reutilizables**, por lo que en cada uno de ellos incluiremos únicamente el código HTML, JavaScript y CSS inherente a dicho componente. Esto evitará que tengamos que navegar por una gran cantidad de archivos y hará que las aplicaciones sean más fáciles de mantener.

Lo cierto, es que no es correcto llamar HTML al código de la sección **template**, puesto que se trata de código *Vue.js*.

La parte lógica y los datos del componente se agregan en la sección `<script> ... </script>`. La sentencia **export** permite exportar nuestros componentes, de forma que pueda incluirse en otros archivos.

En la sección `<style> ... </style>` se incluirá el código CSS del componente.

Gracias al atributo `scoped`, el código CSS solamente afectará al componente actual.

1.5. Creación de componentes

Vamos a crear una aplicación para gestionar datos de personas, por lo que, antes de nada, **crearemos un componente** que muestre una **lista de personas** en el directorio `src/components`. Al archivo le llamaremos `TablaPersonas.vue`, ya que la convención en *Vue.js* es que los nombres de archivo estén en “Pascal Case” (XxxxYyyy).

1.5.1. Crea un componente con *Vue.js*

A continuación agregaremos el siguiente código donde incluiremos una tabla HTML. Además, verás que incluimos el div `<div id="tabla-personas">` antes de la tabla. El motivo es que en todos los **componentes** en *Vue.js* deben contener un **único elemento como raíz**:

```
<!-- 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>
      </tbody>
    </table>
  </div>
</template>
```

```
<tr>
  <td>Tyrion</td>
  <td>Lannister</td>
  <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>
```

Aunque el nombre de archivo esté en Pascal Case (`TablaPersonas.vue`), lo habitual en *Vue.js* es que el nombre del componente en el interior del archivo que lo contiene esté en Kebab Case (`tabla-personas`), de modo que se respete la convención que se usa en HTML.

1.5.2. Agrega el componente a la aplicación

Una vez hemos creado y exportado el componente `TablaPersonas`, vamos a importarlo en el archivo `App.vue`. Para importar el archivo escribiremos la siguiente

sentencia en la parte superior de la sección `script` del archivo `App.vue`:

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

Tal y como ves, podemos usar el carácter `@` para referenciar al directorio `src`.

Seguidamente, vamos a agregar el componente `TablaPersonas` a la propiedad `components`, de modo que *Vue.js* sepa que puede usar este componente. Debes agregar todos los componentes que crees a esta propiedad.

Para renderizar el componente `TablaPersonas`, basta con agregar `<tabla-personas />`, en Kebab case, al código HTML. A continuación puedes ver el código completo que debes incluir en el archivo `App.vue`, reemplazando al código existente:

Listado 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'

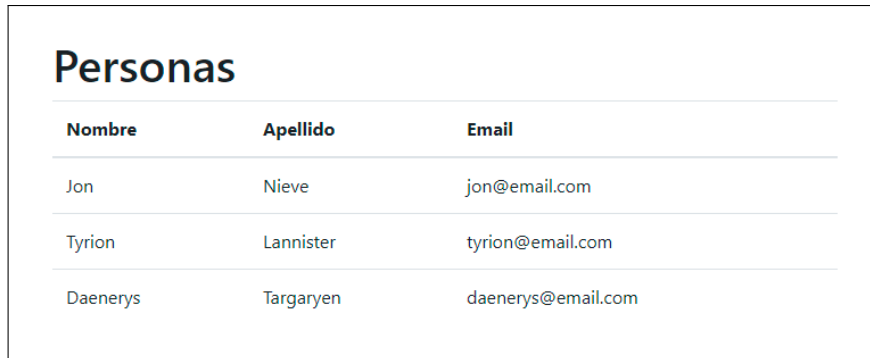
  export default {
    name: 'app',
    components: {
```



```
        TablaPersonas,
      },
    }
  </script>

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

Si accedes al proyecto en tu navegador, deberías ver el resultado que se muestra en la figura 3.



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

Figura 3: Página web creada por el código 4.

A continuación reemplazaremos los datos estáticos de la tabla por datos dinámicos.

1.5.3. Agrega datos al componente

En la tabla de nuestro componente sólo tenemos texto estático. Vamos a reemplazar los datos de las personas por un array de objetos que contengan los datos de las personas. Para ello, vamos a agregar el método `data` a nuestra aplicación en

el archivo `App.vue`. Este método devolverá los datos de los usuarios. Además, cada usuario tendrá un identificador:

```
// ...
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',
        },
      ],
    }
  },
}
```

```
    }  
  },  
}  
// ...
```

Una vez hemos agregado los datos en el componente `App.vue`, debemos pasárselos al componente `TablaPersonas`. Los datos se transfieren como propiedades, y se incluyen con la sintaxis `:nombre="datos"`. Es decir, que se tratan igual que un **atributo** HTML, salvo por el hecho de tener **dos puntos** `:` delante:

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

Alternativamente también puedes usar `v-bind:` en lugar de `:`, que es la forma larga de agregar propiedades:

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

A continuación debemos aceptar los datos de las personas en el componente `TablaPersonas`. Para ello debemos definir la **propiedad** `personas` en el objeto `props`. El objeto `props` debe contener todas aquellas propiedades que va a recibir el componente, conteniendo pares que incluyan el **nombre de la propiedad** y el **tipo** de las mismas. En nuestro caso, la propiedad `personas` es un `Array`:

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

También es posible agregar las propiedades como un array de cadenas, aunque es menos habitual:

```
...
```

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

1.5.4. Agrega un bucle al componente

Una vez hemos agregado los datos al componente `TablaPersonas`, vamos a crear un bucle que recorra los datos de las personas, mostrando una fila de la tabla en cada iteración. Para ello usaremos el atributo `v-for`, que nos permitirá recorrer los datos de una propiedad, que en nuestro caso es la propiedad `personas`:

```
<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>
```

Si ahora ves la aplicación en tu navegador, verás que en apariencia no ha cambiado (figura 3), aunque ahora es mucho más dinámica, pudiendo incluso agregar varias tablas usando el mismo componente:

En el siguiente apartado veremos cómo agregar personas dinámicamente a la tabla.

1.6. Creación de formularios

A continuación vamos a ver cómo puedes agregar nuevas personas al componente `TablaPersonas`. Para ello crearemos otro componente que incluya un pequeño formulario.

1.6.1. Crea un formulario con *Vue.js*

Vamos a crear el archivo `FormularioPersona.vue` en la carpeta `src/componentes`, en el que agregaremos un formulario con un campo `input` para el nombre, otro para el apellido y otro para el email de la persona a agregar. Finalmente también agregaremos un botón de tipo `submit` que nos permita enviar los datos.

En cuanto al código JavaScript, crearemos la propiedad `persona` como una propiedad que será devuelta por el componente, que incluirá el `nombre`, el `apellido` y el `email` de la persona que se agregue. Este sería el código del componente:

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

<script>
export default {
  name: 'formulario-persona',
  data() {
```

```
    return {
      persona: {
        nombre: '',
        email: '',
        apellido: '',
      },
    }
  },
}
</script>

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

También hemos agregado un margen al formulario con CSS en la sección **style**.

1.6.2. Agrega el formulario a la aplicación

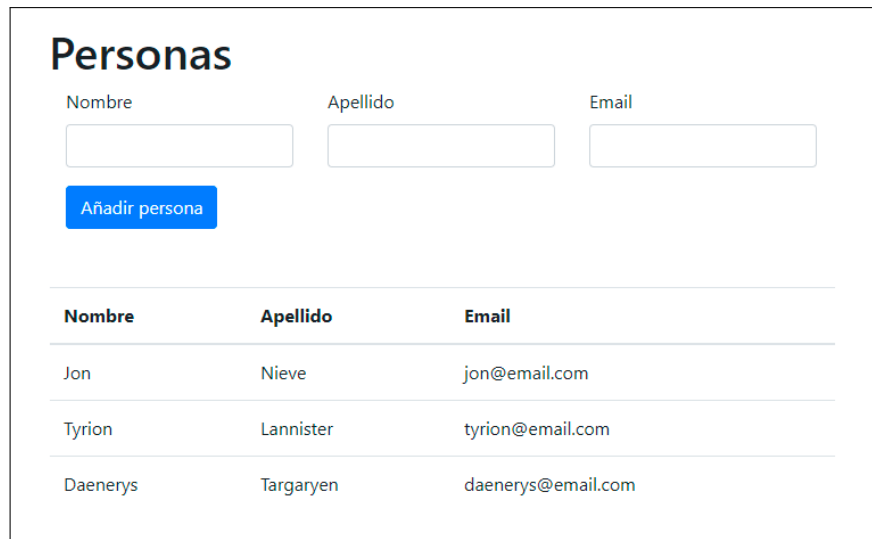
A continuación vamos a editar el archivo `App.vue` y a agregar el componente `FormularioPersona` que hemos creado:

```
<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 /> <!-- <<<<<< -->
      </div>
    </div>
  </div>
</template>
```

```
        <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>
```

Si ahora accedes al proyecto en tu navegador, verás que se muestra el formulario sobre de la tabla:



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

Figura 4: Página web creada por el código 1.8.1.

1.6.3. Enlaza los campos del formulario con su estado

A continuación debemos obtener los valores que se introduzcan en el formulario mediante *JavaScript*, de modo que podamos asignar sus valores al estado del componente. Para ello usamos el atributo `v-model`, que enlazará el valor de los campos con sus respectivas variables de estado, que son las definidas en la propiedad `persona` de la sentencia `return` del componente (véase `src/components/FormularioPersona.vue`):

Listado 5: Código que implementa un formulario en *Vue.js*. Las clases (`class`) usadas están definidas en `bootstrap` y mejoran estéticamente la aplicación aunque no añaden funcionalidad a la misma.

```
<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">Agnadir persona</
                ↪ button>
        </div>
    </div>
</div>
</div>
</form>
</div>
</template>
```

Si consultas el resultado en tu navegador y accedes a las **DevTools** de *Vue.js*, podrás ver cómo cambia el estado del componente cada vez que modificas un campo del formulario.

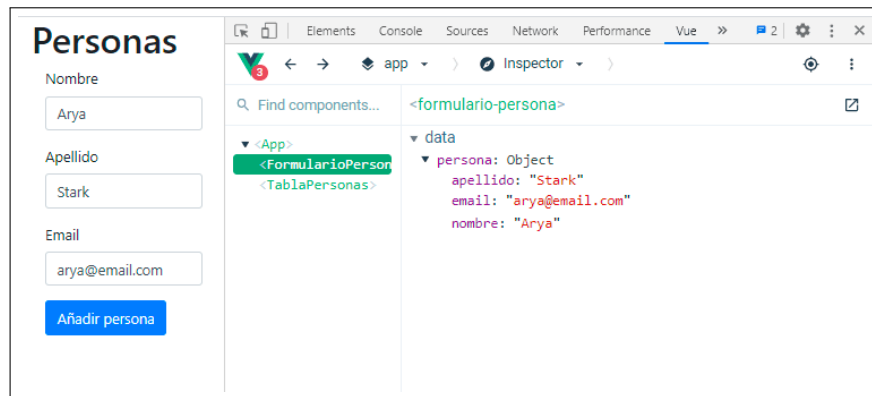


Figura 5: Visualizando el array **personas** con DevTools.

Sin embargo, todavía tenemos que enviar los datos a nuestro componente principal, que es la aplicación **App** en sí misma, de modo que también se modifique su estado, agregando los datos de una nueva persona a la lista de personas.

1.6.4. Agrega un método de envío al formulario

Vamos a agregar un **evento**, también conocido como **event listener**, al formulario. En concreto, agregaremos un evento **onSubmit** para que se ejecute un método cuando se haga click en el botón de envío. Para ello usaremos al atributo **@submit**, que es la forma corta del atributo **v-on:submit**, siendo ambos equivalentes.

En general todos los listeners de eventos en *Vue.js* se agregan con los prefijos **@** o **v-on:**. Por ello, podríamos detectar un click con **@click** o **v-on:click**, y un evento hover con **@mouseover** o **v-on:mouseover**.

Además, no queremos que el servidor se encargue de actualizar la página con el formulario si no que lo haga *Vue.js*, por lo que debemos ejecutar el método **event.preventDefault** que deshabilita el comportamiento por defecto, esto es, conectarse al servidor. Para ello, el evento **@submit** cuenta con el modificador **prevent**,

que es equivalente a ejecutar el método `event.preventDefault()` en el interior de la función asociada al evento `submit`.

Vamos a asociar el método `enviarFormulario` al evento `@submit` del formulario:

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

Ahora vamos a agregar el método `enviarFormulario` al componente. Los métodos de los componentes de *Vue.js* se incluyen en el interior de la propiedad `methods`, que también vamos a crear:

```
export default {
  name: 'formulario-persona',
  data() {
    return {
      persona: {
        nombre: '',
        email: '',
        apellido: '',
      },
    }
  },
  methods: {
    enviarFormulario() {
      console.log('Works!');
    },
  },
}
```

Si pruebas el código y envías el formulario, verás que por la consola se muestra el texto `Works!`.

1.6.5. Emite eventos del formulario a la aplicación

Ahora necesitamos enviar los datos de la persona que hemos agregado a nuestra aplicación `App`, para ello usaremos el método `$emit` en el método `enviarFormulario`. El método `$emit` envía el **nombre del evento** que definamos y los **datos** que deseemos al componente en el que se ha renderizado el componente actual.

En nuestro caso, enviaremos la propiedad `persona` y un evento al que llamaremos `add-persona`:

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

Es importante que tengas en cuenta que el **nombre de los eventos** debe seguir siempre la **sintaxis kebab-case**. El evento emitido permitirá iniciar el método que reciba los datos en la aplicación `App`.

1.6.6. Recibe eventos de la tabla en la aplicación

El componente `FormularioPersona` envía los datos a través del evento `add-persona`. Ahora debemos capturar los datos en la aplicación. Para ello, agregaremos la propiedad `@add-persona` en la etiqueta `formulario-persona` mediante la cual incluimos el componente. En ella, asociaremos un nuevo método al evento, al que llamaremos `agregarPersona`:

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

Seguidamente, creamos el método `agregarPersona` en la propiedad `methods` del archivo `App.vue`, que modificará el array `personas` que se incluye, agregando un nuevo objeto al mismo:

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

```

    }
  }

```

Hemos usado el operador de propagación `...`, útil para combinar objetos y arrays, para crear un nuevo array que contenga los elementos antiguos del array `personas` junto con la nueva persona introducida usando el formulario.

Si ahora ejecutas las DevTools y envías el formulario, verás que se agrega un nuevo elemento al array de personas:

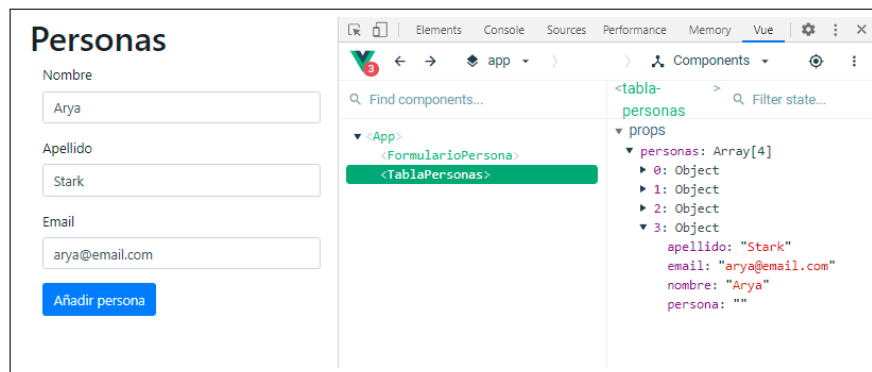


Figura 6: DevTools usado para ver el array `personas` en el navegador.

Sin embargo, es importante que asignemos un ID único al elemento que acabamos de crear. Habitualmente, insertaríamos la persona creada en una base de datos, que nos devolvería la persona junto con un nuevo ID. Sin embargo, por ahora nos limitaremos a generar un ID basándonos en el ID del elemento inmediatamente anterior al actual:

```

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}];
  }
}

```

```
}  
}
```

Lo que hemos hecho es aumentar el valor del ID del último elemento agregado en una unidad, o dejarlo en 0 si no hay elementos. Luego insertamos la persona en el array, a la que agregamos el id generado.

1.7. Validaciones con *Vue.js*

Nuestro formulario funciona, y hasta es necesario introducir una dirección de email válida gracias a la validación HTML de muchos navegadores. Sin embargo, todavía tenemos que **mostrar una notificación** cuando un usuario se inserte correctamente, **reestablecer el foco** en el primer elemento del formulario y **vaciar los campos de datos**. Además, debemos asegurarnos de que se han rellenado todos los campos con **datos válidos**, mostrando un **mensaje de error** en caso contrario.

1.7.1. Propiedades computadas de *Vue.js*

En *Vue.js*, los datos se suelen validar mediante **propiedades computadas**, que son funciones que se ejecutan automáticamente cuando se modifica el estado de alguna propiedad. De este modo evitamos sobrecargar el código HTML del componente. Las propiedades computadas se agregan en el interior de la propiedad `computed`, que añadiremos justo después de la propiedad `methods` del componente `FormularioPersona`:

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

```
        return this.persona.email.length < 1;
    },
},
// ...
```

Hemos agregado una validación muy sencilla que simplemente comprueba que se haya introducido algo en los campos.

A continuación vamos a incluir una variable de estado en el componente **FormularioPersona** a la que llamaremos **procesando**, que comprobará si el formulario se ha enviado o no.

También agregaremos las variables **error** y **correcto**. La variable **error** se usa para saber si debemos mostrar un mensaje de error y la variable **correcto** se usa para decidir si hay que mostrar un mensaje de éxito. Nótese que inicialmente tanto **error** como **correcto** valen **False** (no se muestra ni el mensaje de error ni el de éxito)

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

Seguidamente, debemos modificar el método **enviarFormulario** para que establezca el valor de la variable de estado **procesando** como **true** cuando se esté enviado el formulario, y como **false** cuando se obtenga un resultado. En función del resul-

tado obtenido, se establecerá el valor de la variable `error` como `true` si ha habido algún error, igualmente el valor de la variable `correcto` como `true` si se han enviado los datos correctamente:

```
// ...
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;
  }
}
```

```
    }  
  }  
  // ...
```

Como ves, hemos agregado también el método `resetEstado` para resetear algunas variables de estado.

1.7.2. Sentencias condicionales con *Vue.js*

A continuación vamos a modificar el código HTML de nuestro formulario. Tal y como has podido comprobar ya al observar las DevTools, *Vue.js* renderiza de nuevo cada componente cada vez que se modifica el estado de un componente. De este modo, los eventos se gestionan con mayor facilidad.

Dicho esto, vamos a configurar el formulario de modo que se agregue la clase CSS `has-error` a los campos del mismo en función de si han fallado o no. También agregaremos el posible mensaje de error al final del formulario.

Listado 6: Código que implementa el formulario para añadir personas. Nótese que la construcción `:class="{ 'is-invalid': procesando && nombreInvalido }"` añade la clase `is-valid` si `procesando` y `nombreInvalido` valen `true`. `is-valid` se define en 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>
```

Tal y como has visto, hemos usado el atributo `:class` para definir las clases, ya que no podemos usar atributos que existan en HTML. El motivo de no usar el atributo `class` es que acepta únicamente una cadena de texto como valor.

También hemos agregado el evento `@focus` a los campos `input` para que se reseteen los estados cada vez que se seleccionen.

Para mostrar los mensajes de error hemos usado la **sentencia condicional** `v-if` de *Vue.js*, que hará que el elemento en el que se incluya solamente se muestre si la condición especificada se evalúa como `true`.

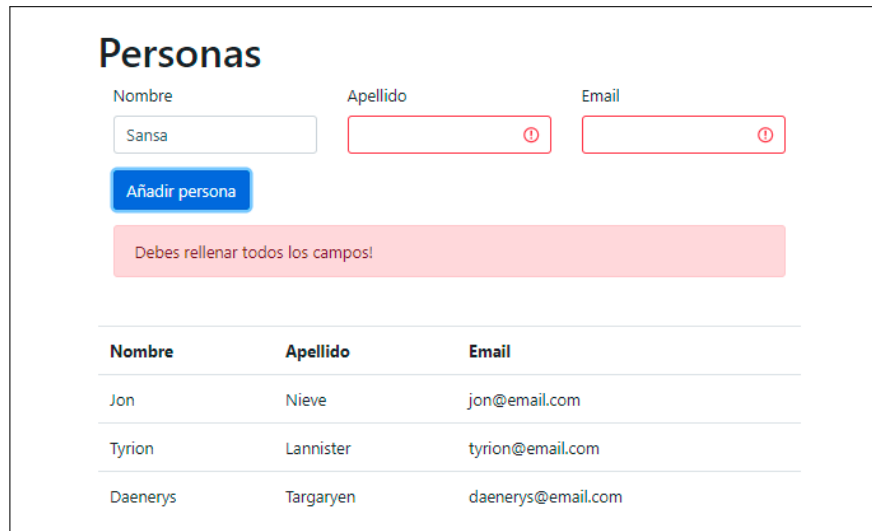
El mensaje de error solamente se mostrará si el valor de la variable `error` es `true` y el formulario se está `procesando`. El mensaje de éxito se mostrará únicamente cuando el valor de la variable `correcto` sea `true`.

También es posible usar sentencias `v-else` o `v-else-if`, que funcionan exacta-

mente igual que las sentencias `else` y `else if` de JavaScript respectivamente.

Para más información acerca del renderizado condicional, consulta la documentación de *Vue.js* (<https://v3.vuejs.org/guide/conditional.html>).

Tras agregar los mensajes de error, echa de nuevo un ojo al navegador para ver el resultado. Verás que cuando de olvidas de rellenar algún campo se muestra un mensaje de error:



Personas

Nombre: Sansa

Apellido:

Email:

Añadir persona

Debes rellenar todos los campos!

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

Del mismo modo, se mostrará el mensaje de éxito cuando la persona se agregue correctamente a la lista:

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. Referencias con *Vue.js*

Cuando envías un formulario, lo normal es que tanto el cursor como el foco, por temas de accesibilidad web, se sitúen en el primer elemento del formulario, que en nuestro caso es el campo **nombre**. Para ello podemos usar las referencias de *Vue.js* (<https://v3.vuejs.org/api/refs-api.html>), ya que permiten hacer referencia a los elementos que las incluyen. Para agregar una referencia debes usar el atributo **ref**.

A continuación vamos a agregar una referencia al campo **nombre**:

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

```
/>
```

Finalmente, tras enviar el formulario, vamos a usar el método `focus` que incluyen las referencias para que el cursor se sitúe en el campo `nombre`:

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

También hemos agregado un evento `@keypress` al campo `nombre` para que el estado se resetee cuando se pulse una tecla, puesto que el foco ya estará en dicho campo.

Si ahora pruebas a agregar una persona, verás que el cursor se sitúa en el primer elemento.

1.8. Elimina elementos con *Vue.js*

El formulario ya funciona correctamente, pero vamos a agregar también la opción de poder borrar personas.

1.8.1. Agrega un botón de borrado a la tabla

Para ello, vamos a agregar una columna más a la tabla del componente `TablaPersonas`, que contendrá un botón que permita borrar cada fila:

```
<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. Emite un evento de borrado desde la tabla

Ahora, al igual que hemos hecho en el formulario, debemos emitir un evento al que llamaremos `eliminarPersona`. Este evento enviará el `id` de la persona a eliminar al componente padre, que en este caso es la aplicación `App.vue`:

```

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

```

1.8.3. Recibe el evento de borrado en la aplicación

Ahora, en la aplicación `App.vue` debes agregar una acción que ejecute un método que elimina a la persona correspondiente con el `id` recibido:

```

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

```


Ahora define el método `eliminarPersona` justo debajo del método `agregarPersona` que hemos creado anteriormente:

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

Hemos usado el método `filter`, que conservará aquellos elementos del array `personas` cuyo `id` no sea el indicado. Y con esto, si consultas el navegador, podrás comprobar que las personas se eliminan de la tabla al pulsar el botón de su respectiva fila:

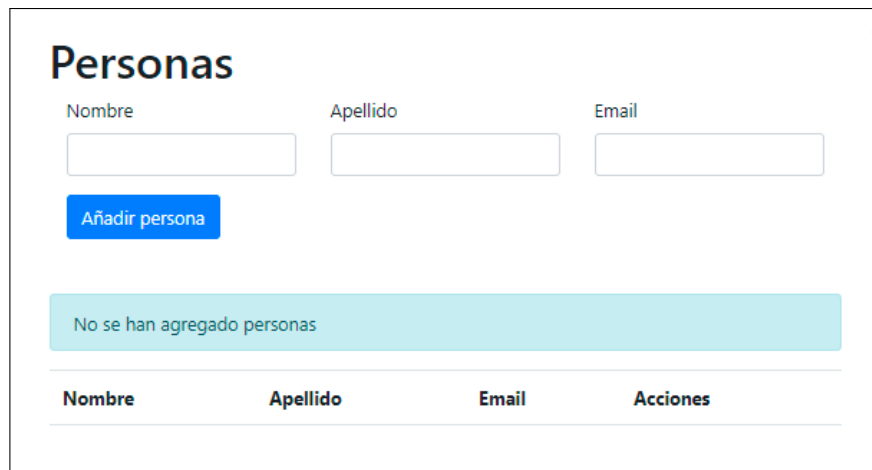
Nombre	Apellido	Email	Acciones
Jon	Nieve	jon@email.com	Eliminar
Tyrion	Lannister	tyrion@email.com	Eliminar

1.8.4. Agrega un mensaje informativo

Para que la aplicación sea más usable, vamos a agregar un mensaje que se mostrará justo antes de la etiqueta `table` de apertura cuando ésta no contenga personas:

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

Este es el mensaje que se mostrará cuando no queden usuarios:



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

1.9. Edita elementos con *Vue.js*

Ahora que podemos eliminar elementos, tampoco estaría mal poder editarlos, que es lo que haremos en este apartado. Vamos a agregar la posibilidad de editar los elementos en la propia tabla, por lo que no necesitaremos componentes adicionales.

1.9.1. Agrega un botón de edición a la tabla

Comenzaremos agregando un botón de edición a la tabla del componente `TablaPersonas`, justo al lado del botón de borrado:

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

El botón ejecutará el método `editarPersona`, al que pasará la persona que seleccionada.

1.9.2. Agrega un método de edición a la tabla

A continuación vamos a agregar el método `editarPersona` a nuestra lista de métodos, en el interior de la propiedad `methods` de nuestro componente:

```
//TablaPersonas.vue
methods: {
  editarPersona(persona) {
    this.personaEditada = Object.assign({}, persona);
    this.editando = persona.id;
  },
}
```

Lo que hemos hecho ha sido almacenar los datos originales de la persona que se está editando actualmente en el objeto `personaEditada`, de forma que podemos recuperar los datos si se cancela la edición. Además, hemos cambiado el valor de la variable de estado `editando`, que como no podría ser de otro modo, debemos agregar a nuestro componente:

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

1.9.3. Agrega campos de edición a la tabla

Ahora es necesario comprobar el valor de la variable `editando` en cada fila, mostrando campos `input` en lugar de los valores de cada persona en la fila que se esté editando:

```
<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>

```

Ahora, si pruebas la aplicación y editas una fila, podrás ver que se muestran campos de edición en la fila correspondiente a la persona que estás editando:

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. Agrega un botón de guardado a la tabla

Sin embargo, todavía necesitamos un **botón de guardado** y otro que permita **cancelar el estado de edición**. Estos botones se mostrarán únicamente en la fila que se está editando:

```

<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>

```

Como ves, hacemos referencia al método `guardarPersona`, que todavía tenemos que agregar. Por ahora, este sería el resultado cuando haces click en el botón de edición de alguna fila:

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

1.9.5. Emite el evento de guardado

Agregaremos el método `guardarPersona` a la lista de métodos. Este método enviará un evento de guardado a la aplicación `App.vue`, que se encargará de **actualizar** los datos de la persona que hemos editado:

```

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. Agrega un método que cancele la edición

Vamos a agregar también el método `cancelarEdicion` que también hemos utilizado en el apartado anterior y que permite **cancelar** el estado de edición de una persona:

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

Tal y como ves, cuando cancelamos la edición de una persona, recuperamos su valor original, almacenado en el objeto `personaEditada`.

1.9.7. Recibe el evento de actualización en la aplicación

Todavía tenemos que modificar el código de la aplicación `App.vue` para que reciba el evento de guardado y actualice los datos de la persona indicada. Primero añadiremos el evento `actualizar-persona`:

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

Ahora vamos a añadir el método `actualizarPersona` a la lista de métodos de la aplicación, justo después del método `eliminarPersona`:

```
actualizarPersona(id, personaActualizada) {
```

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

En el método anterior hemos usado el loop “=>” para recorrer el array de **personas**, actualizando aquella que coincida con el **id** de la persona que queremos actualizar. Si ahora pruebas la aplicación, verás que los cambios se guardan correctamente

1.10. Build & Deploy de la aplicación *Vue.js*

Hemos creado una aplicación que funciona en nuestro ordenador, pero lo ideal sería que funcionase en un servidor externo, de forma que se pueda acceder a ella desde cualquier parte del mundo. Para ello vamos a desplegar la aplicación en **render.com**.

Si tienes tu código en un repositorio de **GitHub** salta a la subsección 1.10.1 en caso contrario continúa leyendo.

Inicializa un repositorio en el directorio raíz del proyecto ejecutando `git init`. Luego accede a tu cuenta de GitHub y crea un nuevo repositorio. Este repositorio se podrá acceder mediante un identificador parecido a `git@github.com:usuario/repositorio`, siendo **usuario** tu nombre de usuario de GitHub y **repositorio** el nombre del repositorio.

Seguidamente ejecuta estos comandos desde el directorio raíz del proyecto para subir tu código a GitHub, reemplazando **usuario** por tu nombre de usuario:

```
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"
```

Con esto ya debería estar nuestro código en GitHub.

1.10.1. Build de la aplicación *Vue.js*

La aplicación que estás usando se compila al vuelo en memoria. Al tratarse de una versión en desarrollo, se realizan comprobaciones de errores y diversos tests que provocan que no sea una versión apta para usar en producción. Para compilar los archivos y crear una versión para producción debes cerrar el proceso actual de node pulsando **CTRL+C** o **CMD+C** y ejecutar el comando:

```
npm run build
```

1.10.2. Deploy de la aplicación *Vue.js*

Vamos a desplegar la aplicación en **render.com**. Primero comprueba que es posible crear una versión “compilada” de la aplicación ejecutando el comando:

```
npm run build
```

A continuación crea un “Static Site” en **render.com** y dale permiso para que acceda a tu repositorio en **GitHub** (<https://render.com/docs/github>). Usa los siguientes valores para crear el repositorio:

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

Si todo ha funcionado correctamente ya tienes tu aplicación desplegada.

1.11. Resumen del trabajo realizado hasta el momento

En este tutorial hemos creado una aplicación CRUD con *Vue.js*. Has aprendido a crear componentes, métodos y formularios. Así mismo has manejado estados y eventos así como expresiones condicionales. Finalmente, también has aprendido a poner una aplicación en producción.

Por ahora, esta aplicación solamente funciona en tu navegador. Si refrescas el navegador, los datos se perderán. Lo que haremos a continuación es conectar la aplicación con un servidor externo, de modo que los datos se guarden permanentemente.

En el siguiente tutorial crearemos una aplicación que soporte la vista (GET), creación (POST), actualización (PUT) y eliminación (DELETE) de los datos de diversas entidades.