Universidad Autónoma de Madrid

Departamento de Informática

Proyecto de Sistemas Informáticos
Práctica - 2.1

Roberto Marabini
Alejandro Bellogín
José A. Macías
Alejandro Moya

Registro de Cambios

Versión[1]	**Fecha**	**Autor**	**Descripción**
3.0	13.01.2025	AB	Actualización sintaxis a Composition API, script setup.
3.1	17.02.2025	JAMI	Actualización sobre librerías Vue.js.
4.0	01.09.2025	AB	Adaptación para el curso 2025-26
4.1	09.02.2026	AM	Modificación del texto con aclaraciones menores

[1]La 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

Aviso

El proyecto descrito a continuación está 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/`). *Vue.js* es un framework de JavaScript para la construcción de interfaces de usuario, por lo que es recomendable tener una base previa en dicho lenguaje. En la plataforma Moodle de la asignatura se proporciona un tutorial de JavaScript (Introduccion javascript - `https://moodle.uam.es/mod/resource/view.php?id=3246580`) que los estudiantes pueden utilizar para reforzar y profundizar en los conceptos necesarios para seguir este material con mayor facilidad. En los tutoriales originales se usa la API de *Vue.js* conocida como "Options API" mientras que en el material que os proporcionamos usamos la "Composition API" que ha sido introducida en *Vue.js* 3 y está llamada a convertirse en un standard.

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 un ejemplo:

Listado 1:

```
<!--index.html-->
<!DOCTYPE html>
```

```
<html lang="en">
  <head>
    <script
        src="https://unpkg.com/vue@3.4/dist/vue.global.prod.js">
    </script>
    <title>Basic application with Vue</title>
  </head>

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

Código fuente: 

Tal y como ves, hemos incluido la versión 3.4 de *Vue.js*. Para incluir otras versiones, basta con que cambies la porción `@3.4` 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@3.4/dist/vue.global.js">
</script>
```

En general es mejor usar la versión de desarrollo durante la creación de la aplicación web porque los mensajes de error son más 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:

Listado 2:

```
...
<script>
  // Importa las funciones necesarias de Vue, como createApp y ref
```

```
  const { createApp, ref } = Vue;

  // Crea una aplicacion Vue
  const app = createApp({
    // La funcion setup es parte de la API de composicion
    setup() {
      // Define una variable reactiva llamada greeting con un valor inicial
         ↪ 'Hello world'
      const greeting = ref('Hello world');

      // Devuelve un objeto que contiene las variables que se haran
         ↪ accesibles en el componente
      return {
        greeting
      };
    },
  });

  // Monta la aplicacion en el elemento HTML con el id 'app'
  app.mount('#app');
</script>
```

Nótese que únicamente los objetos devueltos en `return` son visibles por el código HTML[2]. 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');
```

[2]Esto es así porque estamos dentro de un bloque `<script></script>`. Si se hubiera definido un bloque del tipo `<script setup></script>`, entonces todas las variables definidas con `ref` serían accesibles sin usar return. Más adelante usaremos esta convención para producir un código más compacto.

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 3: Código completo de la primera aplicación.

```
<!DOCTYPE html>
<html lang="es">

<head>
  <!-- Incluye el script de Vue.js -->
  <script src="https://unpkg.com/vue@3.4/dist/vue.global.prod.js"></script>
  <title>Basic application with Vue</title>
</head>

<body>
  <!-- El elemento con el id 'app' sera controlado por Vue -->
  <div id="app">{{ greeting }}</div>

  <script>
    // Importa las funciones necesarias de Vue, como createApp y ref
    const { createApp, ref } = Vue;

    // Crea una instancia de la aplicacion Vue
    const app = createApp({
      // La funcion setup es parte de la API de composicion
      setup() {
        // Define una variable reactiva llamada greeting con al valor
            ↪ inicial 'Hello world'
        const greeting = ref('Hello world');

        // Devuelve un objeto que contiene las variables que se haran
            ↪ accesibles en el componente
```

```
      return {
        greeting
      };
    },
  });

  // Monta la aplicacion en el elemento HTML con el id 'app'
  app.mount('#app');
</script>
</body>

</html>
```

Código fuente: 

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

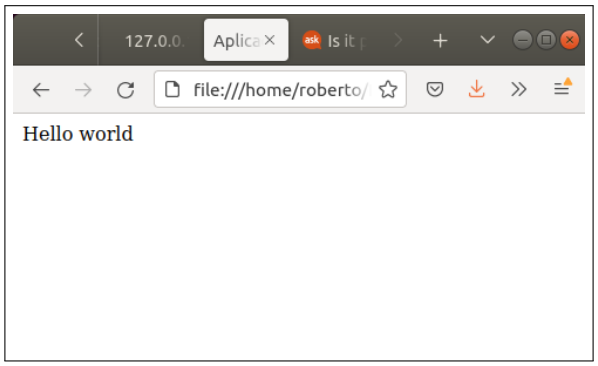


Figura 1: Página web creada por el código 3.

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

```
npm install vue@3.4
```

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

```
npm init vue@3.4 tutorial-vue
```

y responde no a todas las preguntas, excepto en las opciones de Vue Router, Pinia, Cypress, ESLint y Prettier (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 an End-to-End Testing Solution?
    No
    Cypress**
    Playwright
  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 **5173**:

```
npm install
npm run dev
```

Una vez creada la aplicación, abre tu navegador y accede a `http://localhost:5173/` para ver la aplicación, que incluye por defecto una página similar a la figura 2:

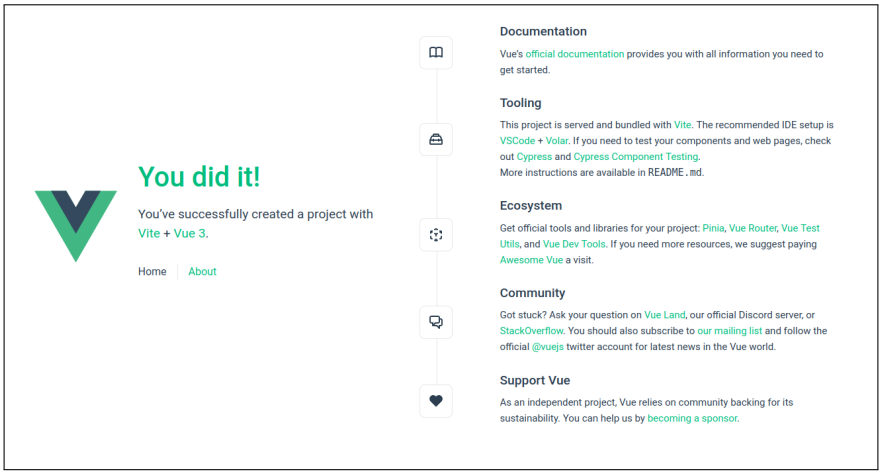


Figura 2: Página web creada por defecto por *Vue.js* 3.4.

En la raíz del projecto 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 4) 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 4: Ejemplo de fichero main.js.

```
// Importa la funcion createApp de la biblioteca Vue
import { createApp } from 'vue'

// Importa el componente principal App desde el archivo App.vue
import App from './App.vue'

// IMPORTANTE: Puedes querer limpiar el archivo main.css por defecto
// ya que las opciones por defecto pueden no satisfacerte.
import './assets/main.css'

// Crea una instancia de la aplicacion Vue y monta el
// componente App en el elemento con el ID 'app'
createApp(App).mount('#app')

// Las dos lineas siguientes haran que Bootstrap este disponible para tu
// aplicacion si Bootstrap ha sido instalado.
// Las instrucciones de instalacion estan disponibles mas adelante en esta
    ↪ guia.

// Importa el archivo JavaScript de Bootstrap desde node_modules
import "../node_modules/bootstrap/dist/js/bootstrap.js"

// Importa el archivo CSS de Bootstrap desde node_modules
import "../node_modules/bootstrap/dist/css/bootstrap.min.css"
```

Nótese que pueden existir otras líneas en el fichero generado. Se recomienda *comentarlas* hasta que sean necesarias (en la siguiente parte de la práctica), en concreto, las relacionadas con `router` (el *import* y *app.use*).

Si ahora accedes al archivo `App.vue` verás que es relativamente complicado. Conceptualmente podéis simplificarlo a las líneas que generan una página vacía:

Listado 5: Fichero App.vue simplificado.

```
<template>
  <!-- El componente principal tiene el ID 'app' y se aplica la clase '
    ↪ container' de Bootstrap -->
  <div id="app" class="container">
    <!-- Contenido del componente va aqui -->
  </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 `Volar`.

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

Vue Devtools es una herramienta esencial para depurar aplicaciones desarrolladas con el framework Vue.js. Permite inspeccionar el estado de los componentes, observar las propiedades reactivas y analizar el flujo de datos en tiempo real. Para activarla, es

necesario ejecutar la aplicación en modo desarrollo (`NODE_ENV=development`) y tener instalada la extensión correspondiente en el navegador. Esta herramienta facilita la identificación de errores.

El valor de la variable de entorno `NODE_ENV` debe establecerse en el momento de arrancar la aplicación. Por ejemplo, desde la línea de comandos puede ejecutarse:

```
export NODE_ENV=production
```

Por motivos de seguridad Devtools no está activa en aplicaciones desplegadas en modo producción. En esta práctica os vamos a pedir que la activéis en la aplicación web desplegada en *Render.com* para facilitar la corrección de la misma. Sed conscientes de que Devtools expone el estado interno de tu aplicación, lo que puede facilitar ataques si está en producción pública.

Para forzar la activación de devtools hay que añadir en `main.js` la línea:

```
app.config.devtools = true  // Enable devtools in production (use with caution)
```

y en `vite.config.js` las líneas

```
  define: {
    __VUE_PROD_DEVTOOLS__: true  // Enable Vue Devtools in production CAREFUL
 }
```

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 `https://getbootstrap.com/docs/5.3/getting-started/introduction/`.

Para incluir Bootstrap en el proyecto, abre el archivo `src/main.js` y agrega las siguientes líneas al final del mismo (véase listado 4)

```
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@5.3.3
npm install @popperjs/core@2.11.8
```

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**:

Listado 6:

```
<template>
  <!-- Aqui ira el marcado HTML del componente -->
</template>

<script>
  // Importamos las funciones necesarias de Vue
  const { createApp, ref } = Vue;

  // Creamos una aplicacion Vue
  const app = createApp({
    // Aqui podrias definir opciones para el
    // componente
  });
</script>

<style scoped>
  <!-- Aqui irian las reglas de estilo del componente,
```

```
  con el modificador scoped de forma que solo afecten a esta pagina -->
</style>
```

Seguramente estés habituado a separar el código HTML del código CSS y JavaScript. Además creerás que es una mala práctica unirlo 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>`.

En la sección `<style> ... </style>` se incluirá el código CSS del componente. Si se desea que el estilo afecte sólo al componente actual se usará `style scoped`.

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. También incluimos el div `<div id="tabla-personas">` antes de la tabla por claridad:

Listado 7: Componente tabla-personas en fichero TablaPersonas.vue.

```
<!-- src/components/TablaPersonas.vue -->

<template>
```

```
<!-- Contenedor principal del componente -->
<div id="tabla-personas">
  <!-- Tabla para mostrar informacion de personas -->
  <table class="table">
    <!-- Encabezado de la tabla -->
    <thead>
      <tr>
        <!-- Columnas del encabezado -->
        <th>Nombre</th>
        <th>Apellido</th>
        <th>Email</th>
      </tr>
    </thead>
    <!-- Cuerpo de la tabla -->
    <tbody>
      <!-- datos para Jon Nieve -->
      <tr>
        <td>Jon</td>
        <td>Nieve</td>
        <td>jon@email.com</td>
      </tr>
      <!-- datos para Tyrion Lannister -->
      <tr>
        <td>Tyrion</td>
        <td>Lannister</td>
        <td>tyrion@email.com</td>
      </tr>
      <!-- datos para Daenerys Targaryen -->
      <tr>
        <td>Daenerys</td>
        <td>Targaryen</td>
        <td>daenerys@email.com</td>
      </tr>
    </tbody>
```

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

<script setup>
// Definicion del componente Vue
defineOptions({
  // Nombre del componente
  name: 'tabla-personas',
});
</script>

<style scoped>
  /* Estilos especificos del componente con el modificador "scoped" */
</style>
```

Código fuente: 

Aunque el nombre del 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 8:

```
<!-- App.vue -->
<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">
        <!-- Inclusion del componente "TablaPersonas" -->
        <tabla-personas />
        </div>
    </div>
    </div>
</template>

<script setup>
// Importacion del componente "TablaPersonas"
import TablaPersonas from '@/components/TablaPersonas.vue'
</script>

<style>
    /* Estilos globales para todos los elementos button en la aplicacion */
    button {
    background: #009435;
    border: 1px solid #009435;
    }
</style>
```

Código fuente: 

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

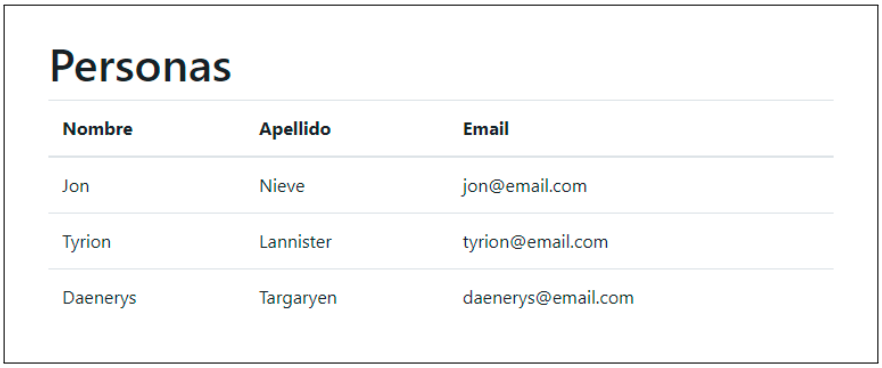


Figura 3: Página web creada por el código del listado 8.

A lo largo del resto del proyecto, 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 crear la variable `personas` dentro del método setup como una variable "reactiva" (`ref()`). De esta forma, y estando dentro de un bloque `script setup`, no hará falta hacer return de dicha variable.

Listado 9:

```
<script setup>
// Importacion del componente 'TablaPersonas' y el metodo 'ref' de Vue 3
import TablaPersonas from '@/components/TablaPersonas.vue'
import { ref } from 'vue';

// Declaracion de una variable reactiva "personas" usando "ref"
const personas = ref([
  {
```

```
    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',
  },
]);
</script>


<style>
    /* Estilos globales para todos los elementos button en la aplicacion */
    button {
    background: #009435;
    border: 1px solid #009435;
    }
</style>
```

Código fuente: 

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:

```
<!-- App.vue -->
<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, incluyendo una función que devuelva el valor por defecto cuando se definan objetos o arrays. En nuestro caso, la propiedad `personas` es un `Array`:

Listado 10:

```
// definicion del componente
defineOptions({
  // nombre del componente
  name: 'tabla-personas',
});

// declaramos y damos valor por defecto para la propiedad personas
const props = defineProps({
  personas: {type: Array, default: () => []},
});
```

Código fuente: 

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`:

Listado 11:

```
<template>
  <!-- Contenedor principal del componente -->
  <div id="tabla-personas">
    <!-- Tabla HTML para mostrar la informacion de personas -->
    <table class="table">
      <!-- Encabezado de la tabla -->
      <thead>
        <!-- nombres de columnas -->
        <tr>
          <th>Nombre</th>
          <th>Apellido</th>
          <th>Email</th>
        </tr>
      </thead>
      <!-- Cuerpo de la tabla con datos dinamicos -->
      <tbody>
        <!-- Iteracion sobre el array de personas utilizando v-for -->
        <tr v-for="persona in personas" :key="persona.id">
          <!-- Celda de datos para el nombre de la persona -->
          <td>{{ persona.nombre }}</td>
          <!-- Celda de datos para el apellido de la persona -->
          <td>{{ persona.apellido }}</td>
          <!-- Celda de datos para el correo electronico de la persona -->
          <td>{{ persona.email }}</td>
        </tr>
      </tbody>
    </table>
  </div>
</template>
```

Código fuente: 

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:

Listado 12:

```
<!-- FormularioPersona.vue -->
<template>
<!-- Contenedor principal del componente -->
<div id="formulario-persona">
    <!-- Formulario con campos para ingresar informacion de una persona -->
    <form>
    <div class="container">
        <!-- Primera fila con campos de nombre, apellido y email -->
        <div class="row">
        <div class="col-md-4">
            <div class="form-group">
            <!-- Etiqueta y campo de entrada para el nombre -->
            <label>Nombre</label>
```

```
            <input type="text" class="form-control" />
            </div>
        </div>
        <div class="col-md-4">
            <div class="form-group">
            <!-- Etiqueta y campo de entrada para el apellido -->
            <label>Apellido</label>
            <input type="text" class="form-control" />
            </div>
        </div>
        <div class="col-md-4">
            <div class="form-group">
            <!-- Etiqueta y campo de entrada para el correo electronico -->
            <label>Email</label>
            <input type="email" class="form-control" />
            </div>
        </div>
        </div>
        <!-- Segunda fila con un boton para agregar persona -->
        <div class="row">
        <div class="col-md-4">
            <div class="form-group">
            <!-- Boton para agnadir persona -->
            <br />
            <button class="btn btn-primary">Agnadir persona</button>
            </div>
        </div>
        </div>
    </div>
    </form>
</div>
</template>

<script setup>
```

```
// Importacion de la funcion "ref" de Vue 3
import { ref } from 'vue';

// definicion del componente
defineOptions({
    // nombre del componente
    name: 'formulario-persona',
});

// Declaracion de una variable reactiva "persona" con propiedades nombre,
    ↪ apellido y email
const persona = ref({
    nombre: '',
    apellido: '',
    email: '',
});
</script>

<style scoped>
/* Estilos especificos del componente con el modificador "scoped" */
form {
    margin-bottom: 2rem;
}
</style>
```

Código fuente: 

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:

Listado 13:

```
<!-- App.vue -->
```

```
<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 /> <!-- NUEVO -->
          <tabla-personas :personas="personas" />
        </div>
    </div>
    </div>
</template>

<script setup>
import TablaPersonas from '@/components/TablaPersonas.vue'
// NUEVO:
import FormularioPersona from '@/components/FormularioPersona.vue'
import { ref } from 'vue';
```

Código fuente: 

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

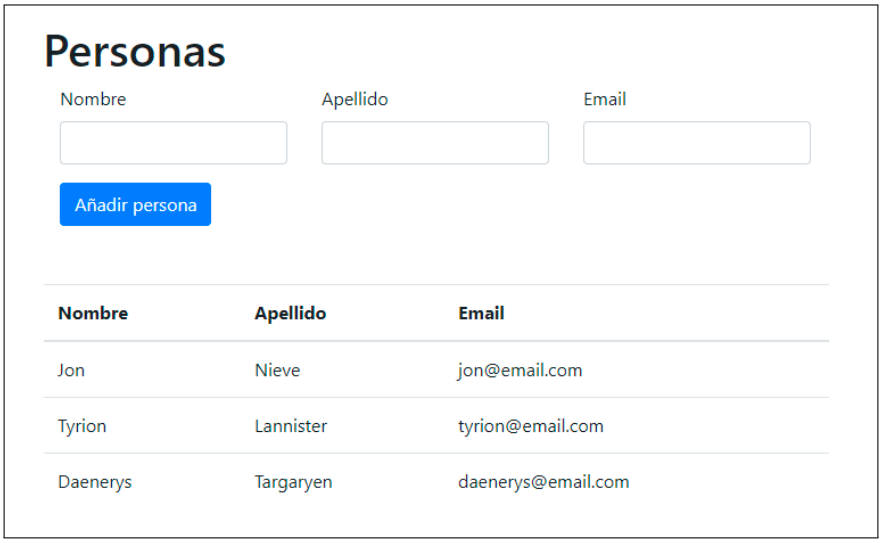


Figura 4: Página web creada por el listado 13.

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`, en *const persona* (véase `src/components/FormularioPersona.vue`):

Listado 14: 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.

```
<!-- FormularioPersona.vue -->
<template>
  <!-- Contenedor principal del componente -->
  <div id="formulario-persona">
    <!-- Formulario con campos para ingresar informacion de una persona -->
    <form>
      <div class="container">
        <!-- Primera fila con campos de nombre, apellido y email -->
```

```
<div class="row">
  <div class="col-md-4">
    <div class="form-group">
      <!-- Etiqueta y campo de entrada para el nombre con binding
          ↪ bidireccional v-model -->
      <label>Nombre</label>
      <input v-model="persona.nombre" type="text" class="form-control
          ↪ " />
    </div>
  </div>
  <div class="col-md-4">
    <div class="form-group">
      <!-- Etiqueta y campo de entrada para el apellido con binding
          ↪ bidireccional v-model -->
      <label>Apellido</label>
      <input v-model="persona.apellido" type="text" class="form-
          ↪ control" />
    </div>
  </div>
  <div class="col-md-4">
    <div class="form-group">
      <!-- Etiqueta y campo de entrada para el correo electronico con
          ↪  binding bidireccional v-model -->
      <label>Email</label>
      <input v-model="persona.email" type="email" class="form-control
          ↪ " />
    </div>
  </div>
</div>
<!-- Segunda fila con un boton para agregar persona -->
<div class="row">
  <div class="col-md-4">
    <div class="form-group">
      <!-- Boton para agnadir persona -->
```

```
            <button class="btn btn-primary">Agnadir persona</button>
          </div>
        </div>
      </div>
    </div>
  </form>
</div>
</template>
```

Código fuente: 

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.

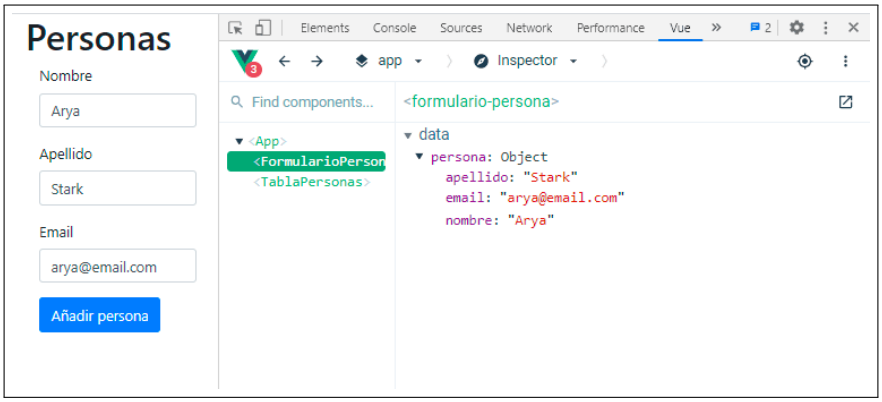


Figura 5: Visualizando el array `personas` con DevTools.

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:

Listado 15:

```
<!-- FormularioPersona.vue -->
<form @submit.prevent="enviarFormulario">
    ...
</form>
```

Ahora vamos a agregar el método `enviarFormulario` al componente. Los métodos de los componentes de *Vue.js* se incluyen como constantes dentro del bloque script, sin necesidad de incluirlos en el método return ya que estamos usando `script setup`:

Listado 16:

```
<script setup>
// Importacion de la funcion "ref" de Vue 3
import { ref } from 'vue';

// definicion del componente
defineOptions({
    // nombre del componente
    name: 'formulario-persona',
});

// Declaracion de una variable reactiva "persona" con propiedades nombre,
    ↪ apellido y email
const persona = ref({
    nombre: '',
    apellido: '',
    email: '',
});

const enviarFormulario = () => {
    console.log('Works!');
  };
</script>
```

Código fuente: 

Si pruebas el código y envías el formulario, verás que por la consola (accediendo a las herramientas para desarrolladores del navegador que se esté usando) se muestra el texto `Works!`.

1.6.5. Emitir 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 `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`.

Para ello, usaremos el método `defineEmits` que está disponible dentro del bloque `script setup` sin necesidad de importarlo[3]. De esta forma, declaramos los eventos que emitirá el componente, y obtenemos un método que podremos usar para emitir el evento cuando sea necesario.

Listado 17:

```
const emit = defineEmits(['add-persona']);

const enviarFormulario = () => {
    console.log('Works!');
    // Emite un evento llamado 'add-persona' con el valor de la variable
        ↪ persona
    emit('add-persona', persona.value);
  };
```

Código fuente: 

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 `setup` del archivo `App.vue`, que modificará el array `personas` que se incluye, agregando un

[3]Ver `https://vuejs.org/guide/components/events`.

nuevo objeto al mismo.

Listado 18:

```
const personas = ref([]);

// definimos una funcion de nombre agregarPersona que agrega una nueva
    ↪ persona al array
const agregarPersona = (persona) => {
  // actualizamos el valor del array creando un nuevo array con los valores
      ↪ existentes y agregando la nueva persona
  personas.value = [...personas.value, persona];
};
```

Código fuente: 

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:

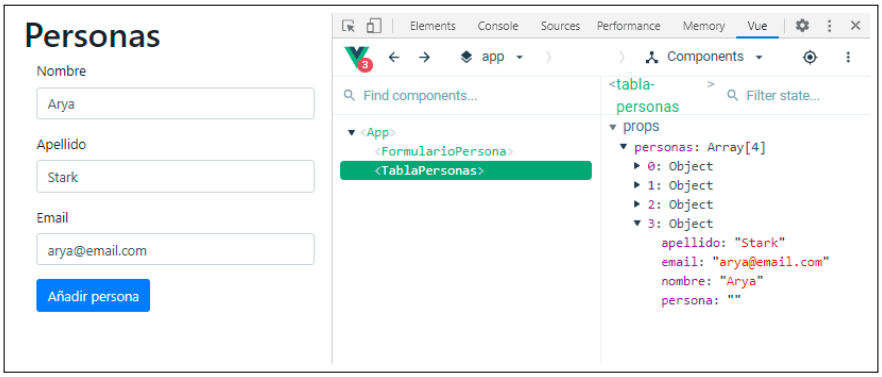


Figura 6: DevTools usado para ver el array `personas` en el navegador.

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 limi-

taremos a generar un ID basándonos en el ID del elemento inmediatamente anterior al actual:

Listado 19:

```
const agregarPersona = (persona) => {
  let id = 0;

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

Código fuente: 

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 del método `setup` del componente `FormularioPersona`. Para ello, debe importarse el módulo `computed`:

```
import { ref, computed } from 'vue';
// ...
 const nombreInvalido = computed(() => persona.value.nombre.length < 1);
 const apellidoInvalido = computed(() => persona.value.apellido.length < 1);
 const emailInvalido = computed(() => persona.value.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` son `False` (no se muestra ni el mensaje de error ni el de éxito).

```
    const procesando = ref(false);
    const correcto = ref(false);
    const error = ref(false);
// ...
```

Seguidamente, debemos modificar el método `enviarFormulario` para que establezca el valor de la variable de estado `procesando` = `true` cuando se esté enviado el formulario, e igual a `false` cuando se obtenga un resultado. En función del resultado 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:

```
const enviarFormulario = () => {
  procesando.value = true;
  resetEstado();

  // Comprobamos la presencia de errores
```

```
  if (nombreInvalido.value || apellidoInvalido.value || emailInvalido.value)
      ↪ {
    error.value = true;
    return;
  }

  emit('add-persona', persona.value);

  // Limpiamos el formulario
  persona.value = {
    nombre: '',
    apellido: '',
    email: '',
  };

  error.value = false;
  correcto.value = true;
  procesando.value = false;
};

const resetEstado = () => {
  correcto.value = false;
  error.value = false;
};
```

Código fuente: 

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 20: 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">
    <!-- Primera fila con campos de nombre, apellido y email -->
    <div class="row">
      <div class="col-md-4">
        <div class="form-group">
          <!-- Etiqueta y campo de entrada para el nombre con binding
              ↪ bidireccional v-model -->
          <label>Nombre</label>
          <input
            ref="nombre"
            v-model="persona.nombre"
            type="text"
            class="form-control"
            data-cy="name"
            :class="{ 'is-invalid': procesando && nombreInvalido }"
            @focus="resetEstado"
            @keypress="resetEstado"
          >
        </div>
      </div>
      <div class="col-md-4">
        <div class="form-group">
          <!-- Etiqueta y campo de entrada para el apellido con binding
              ↪ bidireccional v-model -->
```

```
          <label>Apellido</label>
          <input
            v-model="persona.apellido"
            type="text"
            class="form-control"
            data-cy="surname"
            :class="{ 'is-invalid': procesando && apellidoInvalido }"
            @focus="resetEstado"
          >
        </div>
      </div>
      <div class="col-md-4">
        <div class="form-group">
          <!-- Etiqueta y campo de entrada para el correo electronico con
          ↪ binding bidireccional v-model -->
          <label>Email</label>
          <input
            v-model="persona.email"
            type="email"
            class="form-control"
            data-cy="email"
            :class="{ 'is-invalid': procesando && emailInvalido }"
            @focus="resetEstado"
          >
        </div>
      </div>
    </div>
    <br />
    <!-- Segunda fila con un boton para agregar persona -->
    <div class="row">
      <div class="col-md-4">
        <div class="form-group">
          <!-- Boton para agnadir persona -->
          <button class="btn btn-primary" data-cy="add-button">Agnadir
```

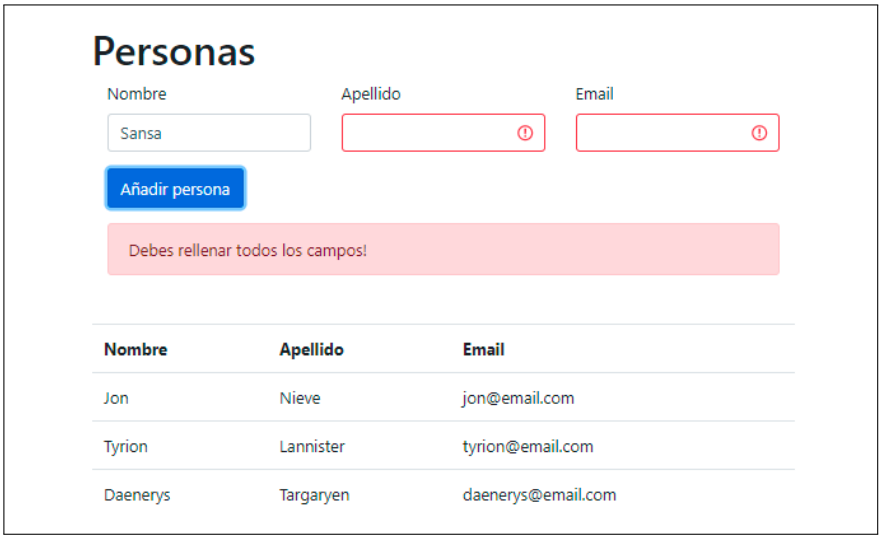
```
            ↪ persona</button>
        </div>
      </div>
    </div>
  </div>
  <br>
  <div class="container">
    <div class="row">
      <div class="col-md-12">
        <div
          v-if="error && procesando"
          class="alert alert-danger"
          role="alert"
        >
          Debes 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.

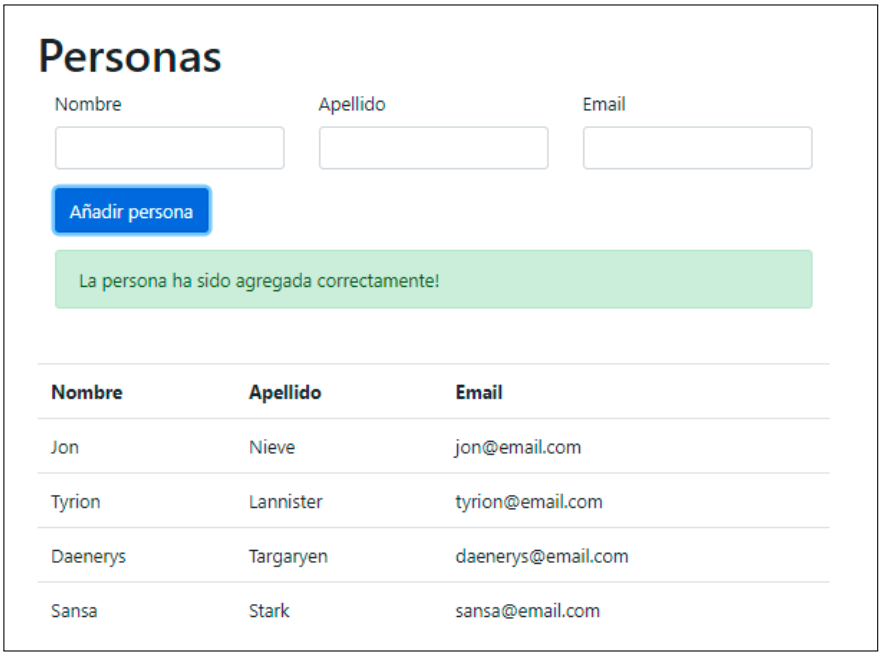
Para mostrar los mensajes de error hemos usado la **sentencia condicional** `v-if`

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 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 exactamente 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 olvidas rellenar algún campo se muestra un mensaje de error:



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



1.7.3. Referencias con *Vue.js*

Cuando se envía un formulario, por motivos de accesibilidad web, el comportamiento esperado es que el foco y el cursor se sitúen en el primer elemento del formulario, que en nuestro caso corresponde al campo `nombre`. Para lograrlo, podemos utilizar las referencias de *Vue.js* (`https://v3.vuejs.org/api/refs-api.html`), las cuales permiten acceder directamente a los elementos que las incluyen. Para agregar una referencia, basta con usar el atributo `ref`.

A continuación se muestra un ejemplo de cómo añadir esta funcionalidad. Primero, en el componente `FormularioPersona.vue`, añadimos `ref="nombre"` al elemento que debe recibir el foco:

```
<!-- FormularioPersona.vue -->
<input
  ref="nombre" <!-- Referencia al elemento input con nombre 'nombre' -->
  v-model="persona.nombre" <!-- Vinculacion bidireccional con la propiedad '
      ↪ nombre' de la variable reactiva 'persona' -->
  type="text" <!-- Tipo de input: texto -->
```

```
  class="form-control" <!-- Clase de Bootstrap para estilos de formulario --
      ↪ >
  data-cy="name" <!-- Atributo para pruebas automatizadas (Cypress) -->
  :class="{ 'is-invalid': procesando && nombreInvalido }" <!-- Clase
      ↪ condicional -->
  @focus="resetEstado" <!-- Evento al obtener el foco -->

/>
```

Una vez enviada la información, podemos usar el método `focus` sobre la referencia para situar el cursor en el campo `nombre`:

```
<script setup>
import { ref, getCurrentInstance, onMounted } from 'vue';

const instance = getCurrentInstance();
const nombre = ref(null);

const enviarFormulario = () => {
  emit('add-persona', persona.value);
  const input = instance.refs.nombre;
  input?.focus();
}
</script>
```

Además, es conveniente enfocar el campo también al montar el componente. Para ello redefinimos la función `onMounted`:

```
onMounted(() => {
  // Enfocamos el campo nombre al montar el componente
  const input = instance.refs.nombre;
  input?.focus();
});
```

Si ahora pruebas a agregar una persona, el cursor se situará automáticamente en el primer elemento del formulario.

Código fuente combinado: 

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:

```
<!-- src/components/TablaPersonas.vue -->
<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>
            <!-- &#x1F5D1; is the wastebasket icon. Icons available at https
                 ↪ ://codepoints.net -->
            <button class="btn btn-danger">&#x1F5D1; Eliminar</button>
          </td>
        </tr>
```

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

Código fuente: 

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`:

```
<!-- TablaPersonas.vue -->
<button class="btn btn-danger" @click="$emit('delete-persona', persona.id)"
    ↪ >&#x1F5D1; 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:

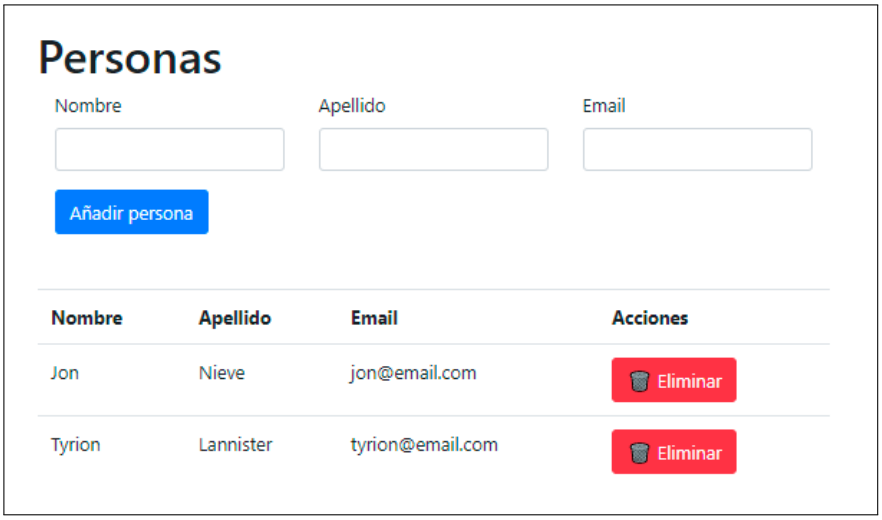
```
<!-- App.vue -->
<tabla-personas :personas="personas" @delete-persona="eliminarPersona" />
```

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

```
<!-- App.vue -->
const eliminarPersona = (id) => {
  try {
    personas.value = personas.value.filter(
      u => u.id !== id
    );
  }
  catch(error){
    console.error(error);
```

```
  }
};
```

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:

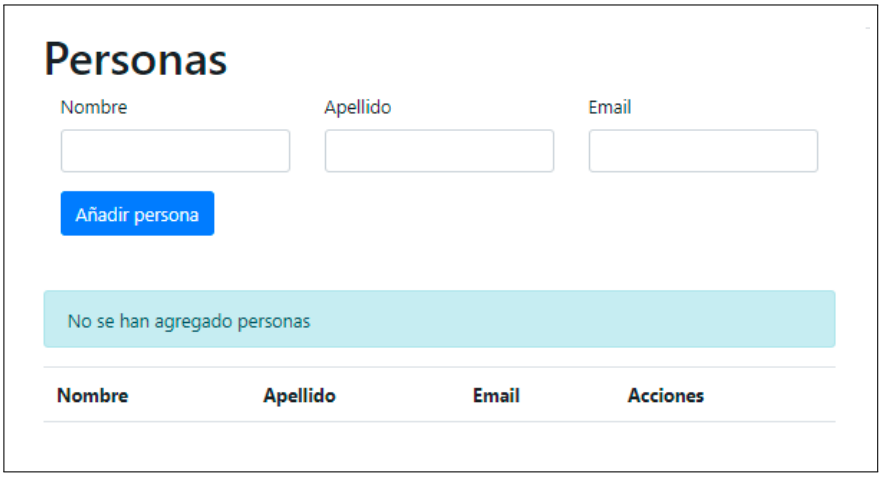


1.8.4. Agrega un mensaje informativo

Para que la aplicación sea más amigable, 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 encontrado personas
</div>
<div v-else>
  <table class="table">
```

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



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` *de la misma componente TablaPersonas*, 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`, incluyendo las variables `editando` y `personaEditada` para permitir la edición, tal como se muestra a continuación:

```
const editando = ref(null);
const personaEditada = ref(null);

const editarPersona = (persona) => {
  personaEditada.value = { ...persona };
  editando.value = persona.id;
};
```

Código fuente: 

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`.

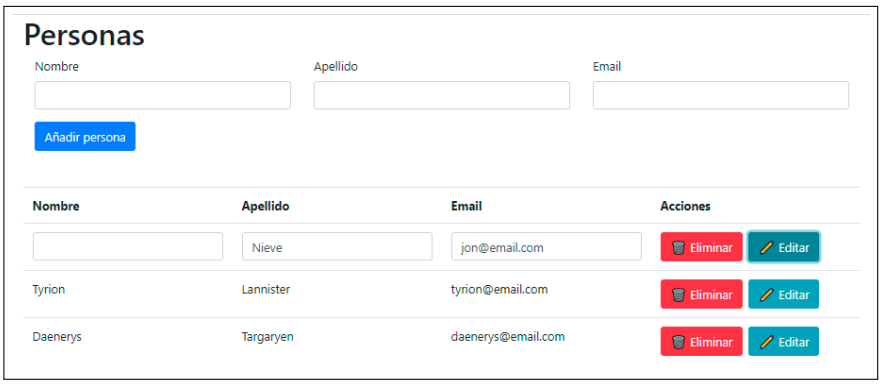
1.9.3. Agrega campos de edición a la tabla

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:

```
    <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 id="persona.nombre" v-model="persona.nombre" type="text"
                ↪  class="form-control" data-cy="persona-nombre">
```

```
</td>
<td v-else>
  {{ persona.nombre }}
</td>
<td v-if="editando === persona.id">
  <input v-model="persona.apellido" type="text" class="form-
      ↪ control">
</td>
<td v-else>
  {{ persona.apellido }}
</td>
<td v-if="editando === persona.id">
  <input v-model="persona.email" type="email" class="form-control
      ↪ ">
</td>
<td v-else>
  {{ persona.email }}
</td>
```

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:

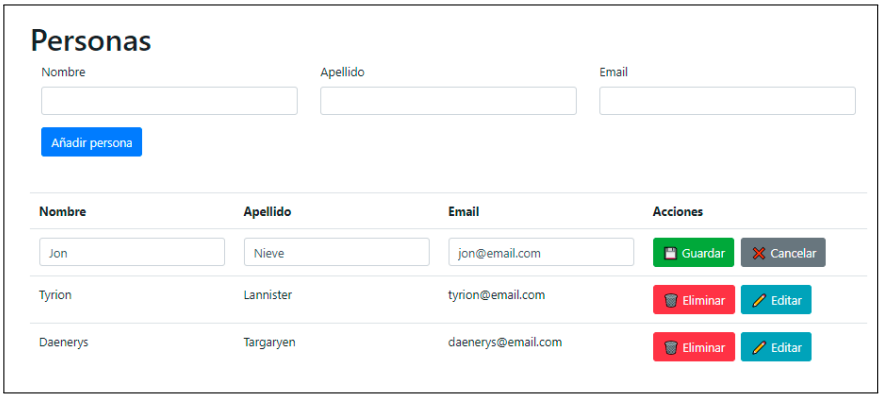


1.9.4. Agrega un botón de guardado a la tabla

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" data-cy="save-button" @click="
      ↪ guardarPersona(persona)">
    &#x1F5AB; Guardar
  </button>
  <button class="btn btn-secondary ml-2" data-cy="cancel-button"
      ↪ @click="cancelarEdicion(persona)">
    &#x1F5D9; Cancelar
  </button>
</td>
<td v-else>
  <button class="btn btn-info" data-cy="edit-button" @click="
      ↪ editarPersona(persona)">
    &#x1F58A; Editar
  </button>
  <button class="btn btn-danger ml-2" data-cy="delete-button"
      ↪ @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:



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:

```
const guardarPersona = (persona) => {
  if (!persona.nombre.length || !persona.apellido.length || !persona.email
      ↪ .length) {
    return;
  }
  emit('actualizar-persona', persona.id, persona);
  editando.value = 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:

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

Código fuente: 

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`:

```
<!-- App.vue -->
<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`:

```
<!-- App.vue -->
const actualizarPersona = (id, personaActualizada) => {
  try {
    personas.value = personas.value.map(persona =>
    persona.id === id ? personaActualizada : persona);
  }
  catch(error){
    console.error(error);
  }
}
```

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.

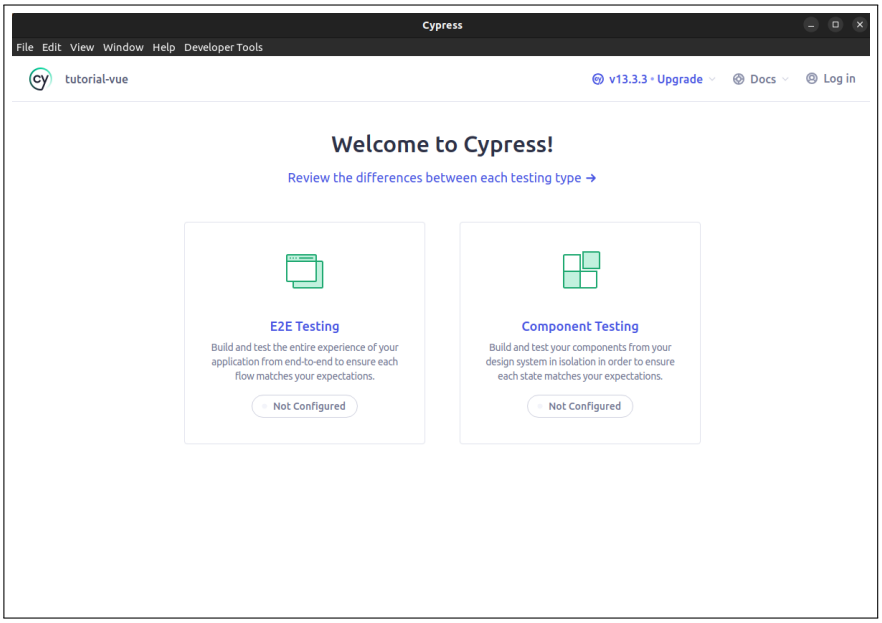
1.10. Testing de la aplicación *Vue.js*

El proceso de testing es ampliamente utilizado en la ingeniería del software para mejorar la calidad del software en sus distintas etapas. Por lo tanto, al igual que en la práctica anterior, es necesario ejecutar test unitarios. En este caso, vamos a realizar testing sobre *Vue.js*, por lo que específicamente para esta práctica vamos a utilizar `cypress`, como se configuró al inicio del proyecto[4].

Para abrir el entorno cypress, debes escribir el siguiente comando:

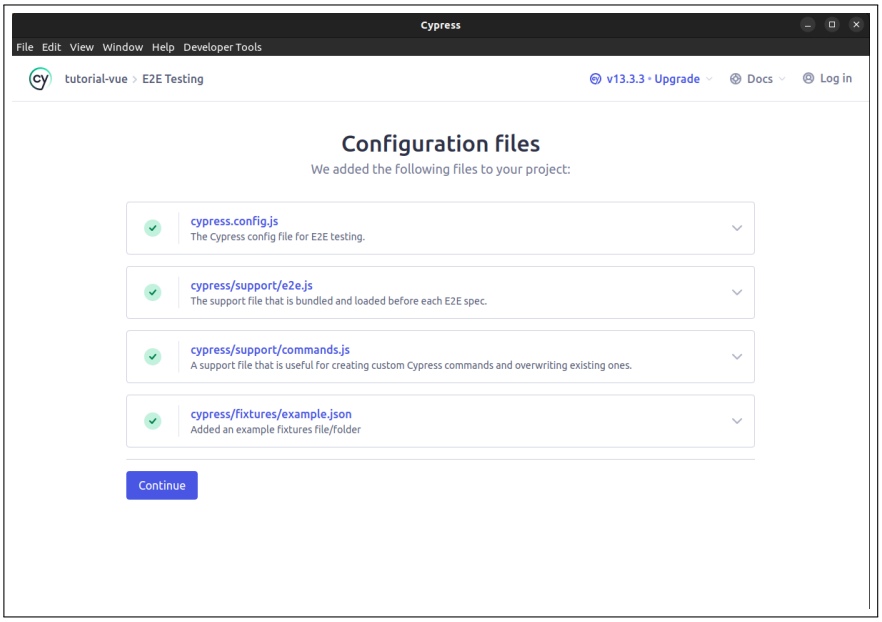
```
npx cypress open
```

En ese momento, arrancará el entorno en su modo gráfico con una pantalla similar a la mostrada a continuación:

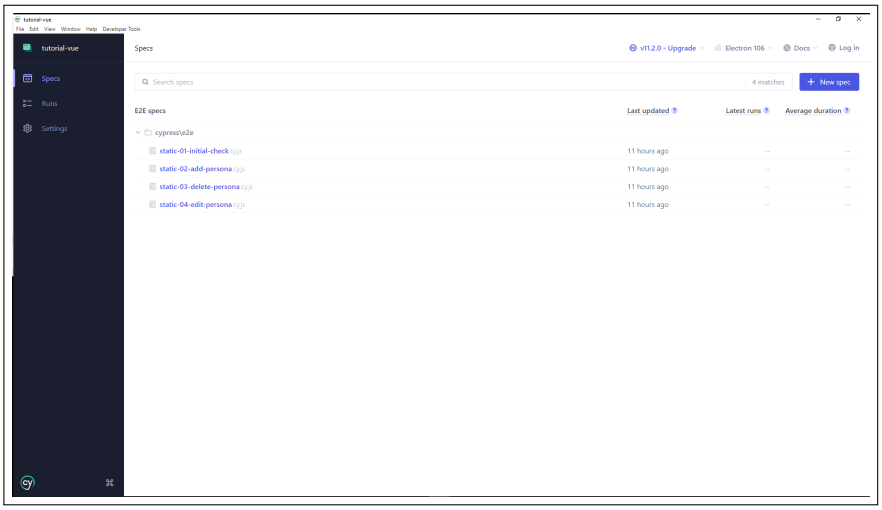


En este caso, debes seleccionar el modo E2E de modo que generemos los siguientes archivos:

[4]Si hiciera falta, se puede instalar usando la siguiente instrucción: `npm install cypress@13.3.3`



Después, se debe seleccionar el navegador a utilizar para realizar el proceso de testing. Una vez elegido, se deben copiar los tests proporcionados por el equipo docente en el directorio `cypress/e2e` (borrando cualquier test que ya existiera en dicha carpeta). De esta forma, los test aparecerán listados en el interfaz gráfico de `cypress`:



Si en algún momento recibes un error indicando que Cypress no se puede conectar al servidor (porque está usando el puerto 4173 en lugar del 5173), simplemente debes cambiar la variable `baseUrl` del fichero `cypress.config.js`.

En este momento, puedes ejecutar los tests en el modo gráfico seleccionando cualquiera de ellos, o ejecutarlos por línea de comandos a través de la siguiente instrucción:

```
npx cypress run
```

Cabe destacar que en esta primera parte de la práctica, únicamente hay que ejecutar los tests que empiezan por `static`, que asumen que la página inicia con una lista vacía y sin persistencia que permita mantener el estado al realizar las operaciones de creación de persona, edición, borrado, etc.

Nota: En los bloques de código que se han mostrado en secciones anteriores, aparecen **etiquetas** `data-cy` en HTML, hay que tener especial cuidado en mantener estas etiquetas, ya que son necesarias para la correcta ejecución de los tests.

1.11. Estilo de la aplicación *Vue.js*

Con el objetivo de satisfacer la guía de estilos para *Vue.js*, se va a validar el estilo de la práctica utilizando la librería `ESLint`. [5].

Además, se debe incluir en el archivo `package.json`, la siguiente entrada en la sección scripts:

```
"lint": "eslint . --ext .vue,.js,.jsx,.cjs,.mjs --fix --ignore-path .
    ↪ gitignore"
```

A continuación, creamos el archivo `.eslintrc.js` incluyendo la siguiente configuración:

```
module.exports = {
  extends: [
    'plugin:vue/vue3-recommended',
  ],
```

[5]Si hiciera falta, se puede instalar usando la siguiente instrucción: `npm install -save-dev eslint eslint-plugin-vue`

```
  rules: {
     'vue/no-unused-vars': 'error'
  }
}
```

Finalmente, para analizar el estilo, una vez la librería está bien configurada e instalada, ejecutamos el siguiente comando:

```
npm run lint
```

Prestad atención y corregid todos los errores y warnings que devuelva `lint`.

1.12. 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*.

1.12.1. 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
```

1.12.2. 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`), eligiendo

la subcarpeta correspondiente a este proyecto como *Root directory*. 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.

1.12.3. Usando cypress sobre la versión desplegada en Render

Ahora mismo tenemos dos aplicaciones completamente equivalentes, una corriendo en local (en *http://localhost:5173/*) y otra en Render (supongamos en *https://tutorial-vue-static.onrender.com/*).

Con el siguiente comando se pueden ejecutar los mismos tests que hemos lanzado antes (Sección 1.10) pero con la aplicación alojada en Render:

```
npx cypress run --config "baseUrl=https://tutorial-vue-static.onrender.com/
    ↪ "
```

También ten en cuenta que puedes lanzar solo un test concreto de la siguiente manera:

```
npx cypress run --spec cypress/e2e/static-02-add-persona.cy.js
```

1.13. 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 testear y desplegar una aplicación en producción.

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