DESENVOLVEMENTO DE INTERFACES WEB 22_23_Ord

1. Introducción a Vue.js

VueJS es una **librería javascript** pensada para tener un **framework** con el que **desarrollar páginas web con javascript**. Con Vue puedes **crear todas las vistas de tu página web**,
puedes hacerlas dinámicas, puedes conectarla a un servidor para tener datos dinámicos de una base de datos, etc. Creada por Evan You ex-trabajador de Google.
Sus características principales es la **modularidad**, se basa en los **componentes**, creación de un **Virtual DOM,** responde a **eventos y transiciones**.
Presenta funciones mixims o funciones reutilizables, entre otras características que iremos viendo.

- **Instalación en Webstorm**

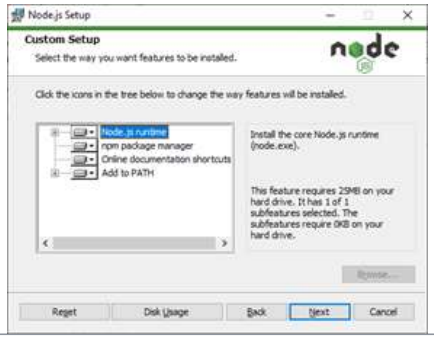
Antes de empezar debe estar **node.js** en el equipo. La mejor manera de saber si ya tenemos Node instalado es **lanzar el comando en la consola "node -v"** y
si está instalado nos debería informar la versión que tenemos.
En caso contrario:

A. Instalación de Node.js.

Entrar en ***https://nodejs.org/es/download/*** y descargar el instalador de Node.js

B. Ejecutar el instalador que acabamos de descargar.

Simplemente debemos avanzar en el proceso de instalación.



C. Una vez finalizado el proceso de instalación, podemos comprobar si se ha instalado.
Para ello, vamos al intérprete de comandos de nuestro ordenador

En la ventana de comandos, escribir *node -v* y pulsar la tecla *Enter*.

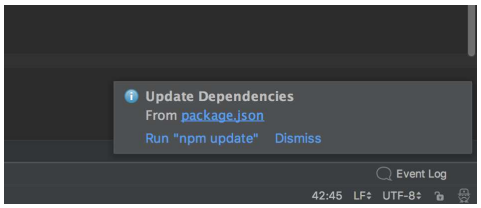
D. Para comprobar que se nos ha instalado también NPM, escribiremos *n -v* y pulsaremos de nuevo *Enter*.
Nos debería aparecer también en este caso la versión del *Node Package Manager*

Finalmente, se recomienta en Webstorm para crear un proyecto con Vue utilizar **Vue CLI**, el cual puede descargarse con **ejecutando:**

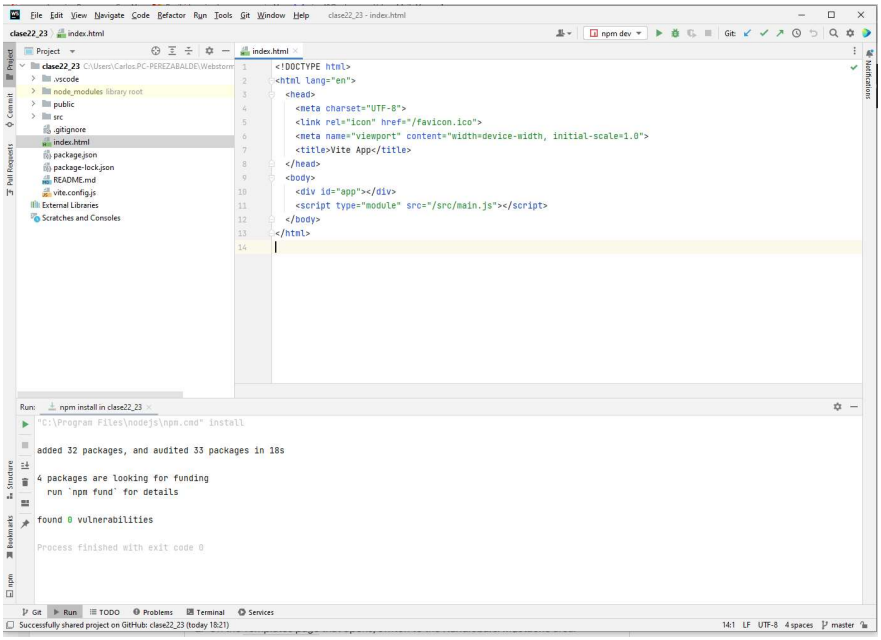
```
npm install vue
```

Conviene también hacer un update:

```
Terminal:  Local  +
PS C:\Users\jcarlos\WebstormProjects\projectintro> npm update

added 11 packages, removed 13 packages, changed 6 packages, and audited 957 packages in 25s

102 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```



Seguramente nos pide instalar dependencias.



Resultado Final

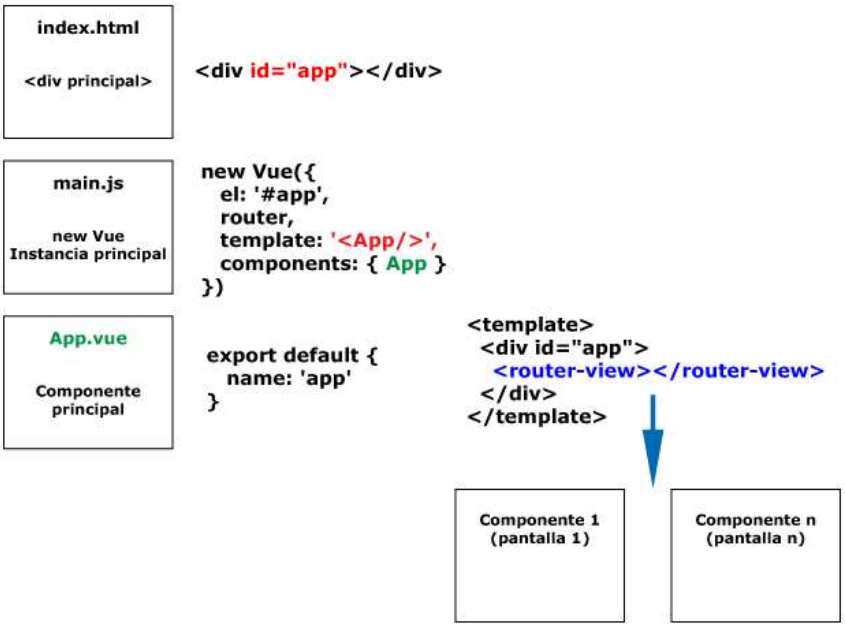
Enlace para configurar VueJS en VisualCode. Aquí
Finalmente, conviene instalar las **DevTools de VUE** en el navegador que se use de forma habitual. Permiten inspeccionar el código, componentes, método o estad en tiempo de ejecución.

- **Google Chrome:** Instala el plugin Vue.js devtools para Chrome.
- **Mozilla Firefox:** Instala el plugin Vue.js devtools para Firefox.

Expliquemos algunas de las carpetas y ficheros que aparecen en un proyecto ***vue.js***

- El fichero `.gitignore` es un archivo donde podemos escribir, linea por linea, los **nombres de archivos (*o patrones o comodines*) que queremos que** `Git`, nuestro sistema de control de versiones**, ignore y no tenga en cuenta** a la hora de guardar cambios en el repositorio. Es **obligatorio** en cualquier proyecto aunque no lo usemos.
- `README.md`. este fichero servirá como **primero punto de información** sobre el repositorio.
- Los ficheros `package.json` y `package-lock.json` son los que utiliza NPM para **crear y gestionar una aplicación o proyecto y sus dependencias.** Es como **un registro.**
- El fichero `vue.config.js` permite incluir detalles de **configuración adicionales de Vue,** como por ejemplo, modificaciones sobre el comportamiento estándar de **Webpack**, opciones de configuración de **plugins Vue** instalados en el proyecto, etc...
- La carpeta denominada `public/` en la carpeta raíz del proyecto. Esta carpeta se utiliza para **almacenar ficheros estáticos que no serán procesados por el framework.**
- La carpeta `src` es una de las **más importantes**. En ella se almacena el código fuente (*source*) de nuestro proyecto, **el que estaremos modificando** desde nuestro editor de texto o IDE. Es muy importante tener presente que dentro de `src` siempre vamos a encontrar los **archivos originales sin procesar**:
- fichero `main.ts` (*TypeScript*) o `main.js` (*Javascript*). Se trata del fichero principal que arranca el proyecto Vue y **que se insertará en la plantilla** `index.html` que se incluye en la carpeta `public/`.

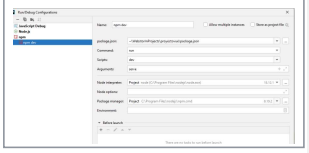
- El fichero ***app.vue*** se trata de un archivo especial de **Vue**, muy similar a un `.html` que incluye **3 etiquetas HTML especiales:** `<template>`, `<script>` y `<style>`. A primera vista, parecen ficheros `.html`, por lo que la curva de aprendizaje es muy sencilla, sin embargo, hay varias diferencias.
- En el interior de la carpeta `src` normalmente podemos encontrar una carpeta `assets`. Dicha carpeta se utiliza para guardar archivos estáticos como imágenes, audio, tipografías, video, etc... La diferencia con la carpeta `assets` o `img` que existe en la carpeta `public` es que, generalmente, los archivos estáticos que tenemos en `public` son para permitir un acceso directo a la URL. La carpeta `src/assets` se utilizan en nuestro código de la aplicación, importándolos, y muchas veces no queremos que se pueda acceder directamente a ellos mediante una URL concreta
- También en el interior de la carpeta `src` podremos encontrar `components`, **probablemente una de las carpetas más importantes de nuestro proyecto Vue**. En ella colocaremos los componentes `.vue` que iremos creando durante nuestro proyecto. Los componentes `.vue` son archivos que contienen el **HTML**, **CSS** y **Javascript** que está relacionado con una determinada parte de la página, como podría ser un botón, un panel desplegable o un comparador de imágenes.

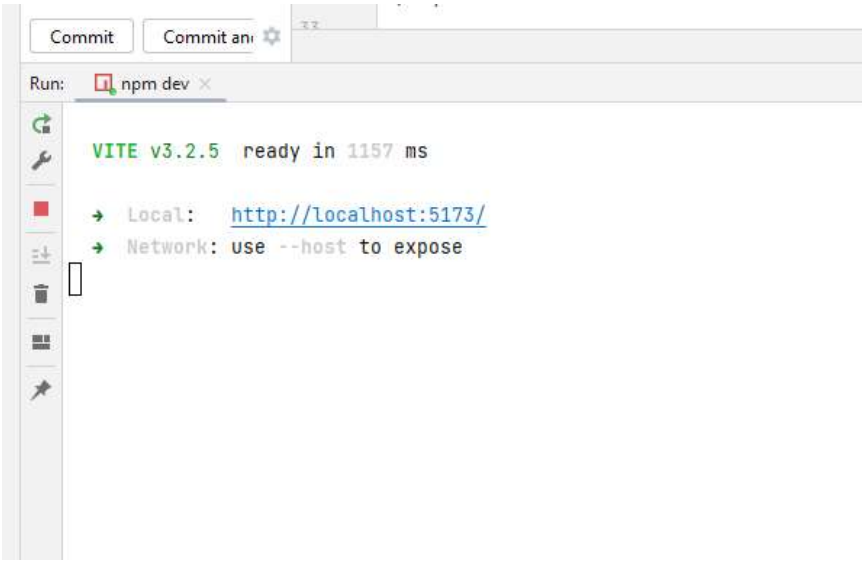


Para lanzar el servidor y ver que todo funciona debemos configurar **Run** en **Webstorm**

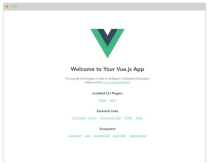
Clickeamos Run y luego en la dirección que se indica en la consola

Editamos Run/configuration y añadimos lo siguiente





Funciona correctamente si aparece la página de inicio.



- **Primeros Pasos**

Vamos a empezar con un **proyecto sencillo.**

Si nos fijamos en el fichero index.html aparece lo siguiente:

```
</head>
<body>
  <div id="app"></div>
  <script type="module" src="/src/main.js"></script>
</body>
```

En ***div id="app"*** sera el punto de inicio de la aplicación. Este fichero index no se tocará mucho. Realmente **donde vamos a trabajar será en la carpeta src.**

Pero antes vamos a comentar un elemento fundamental que son los componentes.

Los componentes son **instancias reutilizables,** que permiten encapsular código dentro de etiquetas HTML. Veamos el siguiente código:

```
Vue.component('saludo',{
    template: '<h2> Hola a todos</h2>'
});
```

Tal como se vió en la imagen anterior, los componentes deben estar situados en la carpeta *components*.

Si nos fijamos dentro de compenent está la propiedad **template,** que es donde enviamos la estructura HTML del componente, y todo luego **dentro una etiqueta div.**

Para utilizar nuestro compoente dentro del HTML solo hace falta colocar el **nombre del componente como si de etiquetas de html** se tratase.

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

Otros elementos importantes son los **métodos y los datos.**

Los **datos** de un componente deben ser tratados desde la función data(). Siempre dentro del objeto de propiedades del componente.

Así mismo, los métodos también son tratados desde esta función. Veamos con el siguiente código.

```
Vue.component('saludo',{
     template: '
        <div>
           <h2> Ejemplo del profesor</h2>
          <h2> {{ saludo }}</h2>
         <h3> Esto es una suma de dos números: {{sumar(2,2)}}</h3>
        </div>
    ' ,
    data (){
      return {
        saludo: 'Hola mundo',
        sumar:  function(num1,num2){
              return num1 + num2
                  }
          }
       }
    });
```

Además, en Vue se suelen usar métodos y computed properties (computadas). A veces de confunden.
Los **métodos tienes que llamarlos como un método normal de cualquier lenguaje de programación,** mientras que **las computed se usan como variables.**
Otras diferencias son que las computadas**, NO** pueden recibir **parámetros de entrada**, mientas que los métodos **SÍ.**
Las computadas **siempre** tienen que hace ***return*** **de un valor**, mientras que los métodos **puede que no devuelvan nada.**

Además, las computadas son **reactivas** quiere decir que cuando **el valor del data cambie, la computada automáticamente se va a ejecutar** y va a devolver el nuevo valor.
Otra diferencia es que las propiedades **computadas tienen caché,** es decir, utilizar propiedades computadas es **más óptimo** porque si Vue detecta que la computada va a devolver el mismo valor,
no ejecutará la computada ahorrando cálculos.

Los **métodos se ejecutan siempre** cada vez aunque el resultado sea el mismo.
Finalmente, es **normal llamar a las computadas desde dentro de los métodos,** mientras que **ejecutar métodos dentro de computadas no se recomienda,** por lo de la caché.

Veamos el siguiente ejemplo:

```
var app = new Vue({
 el: "#app",
 data: {
   n1: 0,
   n2: 0
 },
 computed: {
   sum2: function () {
     console.log('prop-> ' + this.n1 + '+' + this.n2);
     return this.n1 + this.n2;
   }
 },
 methods: {
   sum: function () {
     console.log('method-> ' + this.n1 + '+' + this.n2);
     return this.n1 + this.n2;
   }
 }
});
```

Hemos añadido un log para observar cuando se ejecutan ambas.

El código html sería:

```
<div id="app">
 Type the first number:
 <input type="number" v-model.number="n1"><br />
 Type the second number:
 <input type="number" v-model.number="n2"><br />
 The sum() is: {{ sum() }}<br />
 The sum2 is: {{ sum2 }}
</div>
```

En la consola vemos que si se realiza más de una invocación a la propiedad computed y al método ocurre que la propiedad "tira" de la caché, y el método vuelve a ejecutarse.

ACTIVIDAD.- Realizar el anterior ejercicio en Webstorm o VisualCode

- **Instalación de Bootstrap 5**

De los diferentes frameworks CSS elegiremos Bootstrap, aunque *TailwindCSS* está cogiendo fuerza. Lo primero es instalarlo.

Para ello se puede utilizar ***npm*** (hay otros instaladores como yarn).

Tutorial de **Bootstrap5**

Realizamos la instalación.

```
\WebstormProjects\proyectovue> npm install bootstrap bootstrap-vue
```

Luego se habilita en **main.js**

```
import { createApp } from 'vue'
import App from './App.vue'
import Vue from 'vue'
import BootstrapVue from "bootstrap-vue/
import 'bootstrap-vue/dist/bootstrap-vue
import 'bootstrap/dist/css/bootstrap.css

Vue.use(BootstrapVue);

createApp(App).mount( rootContainer: '#app')
```

IMPORTANTE: Sin embargo nosotros incluiremos un enlace a una versión de Bootstrap en **index.html,** ya que el motivo de este manual no es aprender CSS.

Sin embargo, **nosotros utilizaremos el link a la API de Bootstrap**

Link:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
```

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <link rel="icon" href="/favicon.ico">
      <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet"
            integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anc
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Empresa</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

- **Primera Aplicación**

Los archivos Vue 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 tal y como su nombre indica 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>
```

Vamos a **crear una aplicación para gestionar datos de clientes**, por lo que, antes de nada, creamos un **componente,** que muestre la lista de clientes en el directorio ***src/components***.

Le llamamos al archivo **TablaClientes.vue.** Importante ver la convención de Vue a la hora de nombrar los archivos. Usaremos la notacións **PascalCase,** aunque lo habitual es usar **Kebab Case.**

Añadiremos el siguiente código, concretamen una tabla HTML, con el ** ** antes de la tabla, ya que todos los **componentes** en Vue deben contener un **único elemento como raíz.**

App.vue | index.html | TablaClientes.vue

```
<template>
  <div id ='tabla-clientes'>
    <table class = 'table'>
      <thead>
      <tr>
        <th>NOMBRE</th>
        <th>APELLIDOS</th>
        <th>Email</th>
      </tr>
      </thead>
      <tbody>
      <tr>
        <td>Lou</td>
        <td>Reed</td>
        <td>lreed@email.com</td>
      </tr>
      <tr>
        <td>Patti</td>
        <td>Smith</td>
        <td>psmith@email.com</td>
      </tr>
      <tr>
        <td>Janis</td>
        <td>Joplin</td>
        <td>jjoplin@email.com</td>
      </tr>
      </tbody>
    </table>
  </div>
</template>

<script>
export default {
  name: "tabla-clientes",
}
</script>

<style scoped>

</style>
```

Creado el componente **TablaClientes**, vamos a **importarlo** desde **App.vue**, con la siguiente sentencia dentro de la sección **script.** La notación **@** hace referencia al directorio **src.**

Para **renderizar** el componente **TablaClientes**, basta agregar **<tabla-clientes />** en notación Kebab Case, al código HTML. El código final sería tal como se muestra.

App.vue | index.html | TablaClientes.vue

```vue
<template>
  <div id="app" class="container">
    <div class="row">
      <div class="col-md-12">
        <h1>CLIENTES</h1>
      </div>
    </div>
    <div class="row">
      <div class="col-md-12">
        <TablaClientes />
      </div>
    </div>
  </div>
</template>

<script>
import TablaClientes from "@/components/TablaClientes.vue";
  export default {
    name: 'app',
    components:{
      TablaClientes,
    },
  }
</script>

<style>
button{
  background: #008953;
  border: 1px solid #008953;
}
</style>
```

El resultado sería el siguiente:

- **Datos en el aplicación**

De momento solo tenemos un texto estático. Vamos a reemplazar los datos de la tabla de los clientes por datos dinámicos, en este caso, por un **array de objetos** que contengan sus datos. Para ello agregamos el método **data** a la aplicación en el archivo en el archivo **Vue.js**

```vue
<script>
import TablaClientes from "@/components/TablaClientes.vue";
  2 usages
export default {
  name: 'app',
  components:{
    TablaClientes,
  },
  data(){
    return{
      clientes:[
        {
          id: 1,
          nombre: 'Lou',
          apellidos: 'Reed',
          email: 'lreed@email.com',
        },
        {
          id: 2,
          nombre: 'Patti',
          apellidos: 'Smith',
          email: 'psmith@email.com',
        },
        {
          id: 1,
          nombre: 'Janis',
          apellidos: 'Joplin',
          email: 'jjoplin@email.com',
        },
      ],
    }
  },
}
</script>
```

Una vez agregado los datos al componente **App.js** hay que pasárselos al compoonente TablaClientes, y se transfieren como propiedades con la sintaxis ***:nombre = 'datos'*** es decir, como un **atributo HTML,** aunque con **dos puntos :** delante. También se puede usar **v-bind:** que es la forma larga de agregar propiedades.

A continuación debemos aceptar los datos de los clientes en el componente TablaClientes, para ello definimos la **propiedad clientes**, en el objeto props. Este objeto debe contener todas las propiedas que va a recibir el component, conteniendo pares con el nombre de la **propiedad y tipo**. En este caso:

```vue
<script>
2 usages  XoanCarlos *
export default {
  name: "tabla-clientes",
  props: {
    clientes: Array,
    <!-- También vale así: props: ['cliente'], -->
  },
}
</script>

<style scoped>

</style>
```

O así,

```
<script>
2 usages  XoanCarlos *
export default {
  name: 'tabla-clientes',
  props: ['clientes'],
  }

</script>
```

Para acabar, como se puede sospechar, hay que recurrir a un bucle para recorrer el array y así mostrar la tabla en cada interacción. PAra ello se usa el **atributo** `v-for`, que recorrerá la propiedad **clientes.** Nos vamos al fichero TablaClientes.vue y lo modificamos así.

- **Formularios**

Vamos a crear el archivo `FormularioClientes.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 del cliente a introducir.

Finalmente también agregaremos un botón de tipo `submit` que nos permita enviar los datos.

FormularioClientes.vue | README.md | main.js | index.html | App.vue

```
<template>
  <div id="formulario-clientes">
    <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">Guardar</button>
            </div>
          </div>
        </div>
      </div>
    </form>
  </div>
</template>
```

En el código JavaScript, crearemos la propiedad cliente como una propiedad que será **devuelta por el componente**, que incluirá el `nombre`, el `apellido` y el `email` del cliente añadido.

Solo nos queda ir al fichero **App.vue** para **importar** el componente creado **FormularioClientes.vue.**

En el **template:**

En el **script**

Resultado:

Para enlazar los campos del formulario con sus respectivas variables de estado se puede usar el atributo **v-model.** Cambiamos el código en **FormularioClientes.vue** a lo siguiente:

A continuación, tenemos que enviar esos datos que recorre el formulario al componente principal que es la aplicación App agregando los datos del nuevo cliente.

Para ello tenemos que **agregar un evento,** concretamente **Event Listener** al formulario. Será el evento **onSubmit** que ejecutará un método cuando se pulse el botón.

Hay dos atributos que hacen lo mismo ***@submit*** y ***v-on:submit.***

Además añadiremos, por último, el método **event.preventDefault,** que evita el refresco de la página al enviar el formulario.

Para ello **submit** cuenta con el **modificador prevent,** con lo que el código en el atributo del formulario queda:

Nos queda agregar el **método envioForm** al componentes.

Los métodos de los componentes de Vue se incluyen en el interior de la propiedad **methods.**

Ejecutar y comprobar la consola.

Ahora necesitamos enviar los datos del cliente a **App.vue** es decir a la aplicación. Para ello usaremos el método **$emit** del método envioForm.

El método 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 **cliente** y el evento que llamamos **add-Cliente**

Es importante que tengas en cuenta que el **nombre de los eventos** debe seguir siempre la **sintaxis kebab-case**.

El componente **FormularioCliente** envía los datos a través del evento **add-cliente**. Ahora debemos capturar los datos en la aplicación.

Para ello, agregaremos la propiedad **@add-cliente** en la etiqueta **formulario-cliente** mediante la cual incluimos el componente.

En ella, asociaremos un nuevo método al evento, al que llamaremos **addCliente:**

Seguidamente, creamos el método en la propiedad **methods** del archivo **App.vue,** agregando un nuevo objeto al mismo:

Hemos usado el **operador spread de propagación**, es decir, al array le añadirmos un nuevo dato.

Por otro lado, debemos asignar un ID único al cliente creado. Habitualmente, insertaríamos en una base de datos, que nos devolverían el cliente con un nuevo ID.
Sin embargo, por ahora nos limitaremos a generar un ID basándonos en el ID del elemento inmediatamente anterior al actual en **App.vue**:

Los 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.

- **Validaciones en Vue**

Nuestro formulario funciona. 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. Para ello usamos **propiedades comuntadas,** en **FormularioClientes.vue**

Los datos se suelen validar mediante **propiedades computadas** o *computed properties*, 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.

Esta 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 **FormularioCliente** a la que llamaremos **procesando**, que comprobará si el formulario se está enviando actualmente o no.

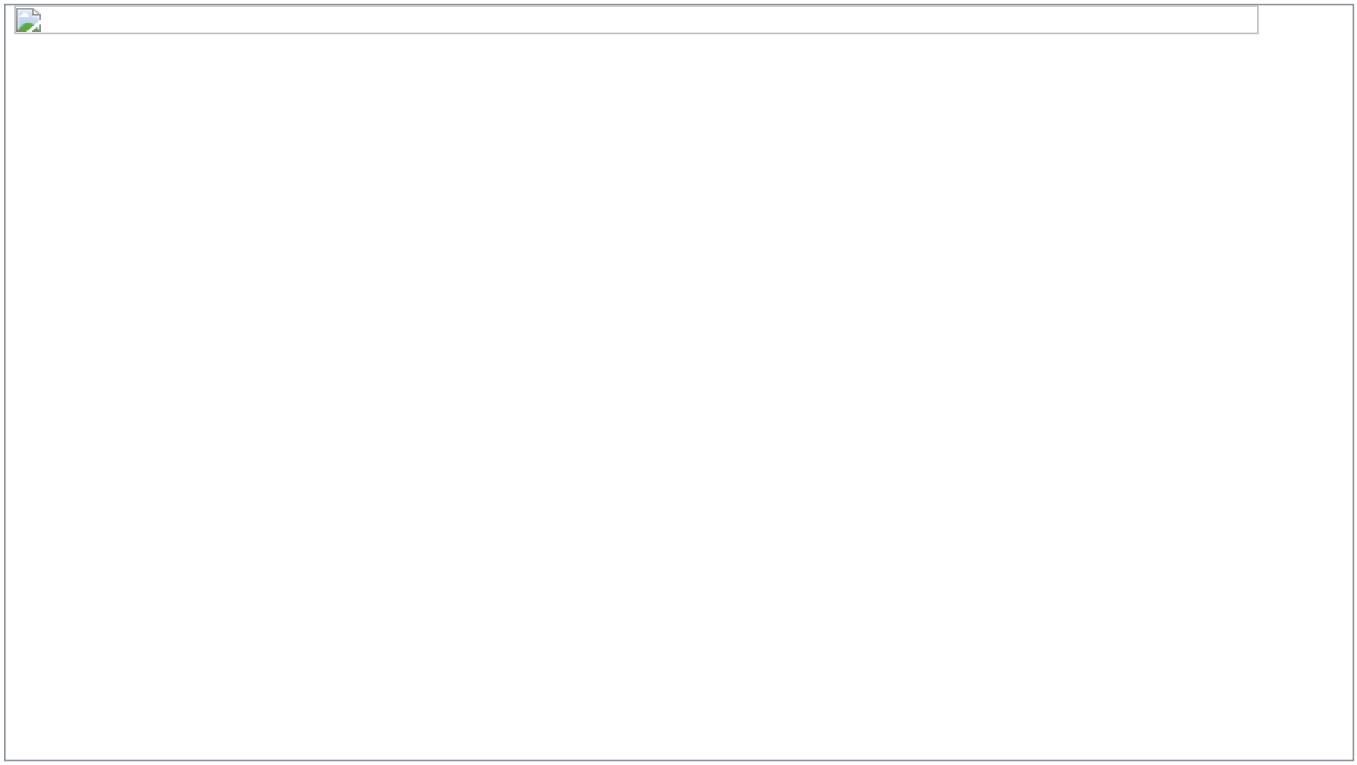
También agregaremos las variables **error y correcto.**

El valor de la **variable error sera true** si el formulario se ha enviado correctamente o **false** si ha habido algún error.

Del mismo modo, el valor de la **variable correcto será true** si el formulario se ha enviado correctamente o **false** en caso contrario:.

También debemos modificar **envioForm** para restablecer **a true** la variable **procesando** al finalizar el envío, y false cuando se obtenga algún resultado. En función de ese resultado la variable error pasa a true si ha habido algúrn error o la variable correcto a true si todo fue correcto.

Finalmente **resetEstado** para resetear algunas variables de estado.



- **Sentencias condicionales**

Todo lo anterior permitirá modificar el formulario para controlar que controlar que todo va correcto, o no, con sentencias condicionales. Si algún campo fallo **usarermos la clase CSS has-error y un mensaje**

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, que hará que el elemento en el que se incluya solamente se muestre si la condición especificada se evalúa como **true**.

Según esto saldrá un mensaje de éxto o fallo.

También es posible usar sentencias **v-else y v-else-if,** si fuesen necesarias.

Para más información acerca del renderizado condicional, consulta la documentación de Vue.

- **Referencias con Vue**

Por temas de **accesibilidad web,** una vez ejecutado el submit del formulario debería situarse el foco (cursor de la pantalla) en el primer elemento del formulario, en este caso en el nombre.

Para ello podemos usar las referencias de Vue, concretamente **el atributo ref en el input nombre** (ver template en formulario)

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 el atributo ref en el input nombre** (ver script en formulario)

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.

- **Elminando usuarios**

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

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

Ahora, al igual que hemos hecho en el formulario, **debemos emitir un evento al que llamaremos deleteCliente,** que enviará el **id del cliente** a eliminar al componente padre, que en este caso es la aplicación **App.vue.**

Agregamos en **App.vue** un método que se ejecute de acuerdo al evento recibido.

Ahora solo queda el método **deleteCliente** justo debajo del método **addCliente** que hemos creado anteriormente:

Hemos usado el método **filter**, que conservará aquellos elementos del **array clientes** cuyo **id** no sea el indicado. Para más información acerca de cómo usar este método, consulta el siguiente tutorial, en donde explico cómo filtrar un array en JavaScript.

Finalmente conviene agregar un **mensaje informativo** para mejorar la usabilidad de la aplicación que puede ser colocado, por ejemplo antes de la etiqueta **table**.

Última modificación: Luns, 16 de Xaneiro de 2023, 09:25

◄ UD5.Video y Sonido

Ir a...

2. API REST con Vue.js ►