# Vue Básico

## Contenido

[VIDEO 1 1](#_Toc58345370)

[VIDEO 2 – Vue DevTools 1](#_Toc58345371)

[VIDEO 3 – Enlace Dinamico (v-bind) 1](#_Toc58345372)

[VIDEO 4 – Renderizado de listas (v-for) 1](#_Toc58345373)

[VIDEO 5 – Propiedades Computadas 1](#_Toc58345374)

[VIDEO 6 – Eventos en Vue.JS (v-on) 1](#_Toc58345375)

[VIDEO 7 – La instancia Vue 1](#_Toc58345376)

[VIDEO 8 – Vue Watchers 1](#_Toc58345377)

[VIDEO 9 – Renderizado condicional (v-if) 1](#_Toc58345378)

[VIDEO 10 – Filtros 1](#_Toc58345379)

[VIDEO 11 – Ajax con Axios 1](#_Toc58345380)

[VIDEO 12 – Data binding con estilos y clases (v-bind) 1](#_Toc58345381)

[VIDEO 13 – Data binding en formularios 1](#_Toc58345382)

[VIDEO 14 – Introducción a componentes 1](#_Toc58345383)

[Ejemplos 1](#_Toc58345384)

## VIDEO 1

Primero creamos una instancia del objeto Vue, a la cual hay que pasarle por lo menos un parametro que es 'el' (element). Esto significa que sobre este elemento se va a "montar" Vue.

Ademas tenemos la propiedad 'data' que es el modelo, es decir, que informacion o datos va a manejar nuestra aplicacion (en nuestro ejemplo seran las 'tareas[]'). Eso puede tener todos los niveles y subniveles que quisieramos.

Queda crear un metodo, que es la logica en si o lo que enlaza los elementos, por ejemplo, del input (su contenido) con 'tareas[]'. Todo esto vive dentro de 'methods'. Este OBJETO recibe N numeros de funciones, para nuestro ejemplo solo interesa una, que es la que sirve para guardar una "tarea" (guardarTarea()).

guardarTarea() debe agarrar lo que este dentro del input y guardarlo dentro del modelo. Para eso debemos asociar ese input (donde ingresamos la nueva tarea) con una propiedad del modelo (nuevaTarea: ''). Esta propiedad siempre comienza vacia ya que todavia no tenemos nada.

Ahora para asociar esta propiedad del modelo con un elemento del DOM utilizaremos las directivas (que comienza siempre con 'v-'). 'v-model' realiza esta conexion entre el elemento del DOM y 'nuevaTarea' en nuestro ejemplo. Ahora todo lo que se escriba en el input se verá reflejado en la propiedad 'nuevaTarea' del objeto.

Para mostrar un dato debemos encerrarlo con doble llave '{{ dato }}'. Eso muestra informacion en tiempo real, por lo que si modificamos cualquier dato este lo mostrara.

Ahora tenemos que crear el método que guarde esa nueva tarea dentro de “tareas[]”:

* This esta asociado via Vue directamente con ‘data’.
* “pusheamos” la ‘nuevaTarea’ al arreglo de de tareas ‘tareas[]’.

THIS siempre hace referencia al modelo!!

Ahora debemos asociar el formulario con este nuevo método para empezar a guardar la nuevaTarea. Lo asociamos con una directiva que se encarga de manejar los eventos: ‘v-on:(directiva)=”(método a llamar)”’.

En nuestro ejemplo la directiva es ‘submit.prevent’ y el método será ‘guardarTarea’. Recordar reiniciar la ‘nuevaTarea’ como un string vacio cada vez que la utilicemos para borrar el campo de entrada.

Ahora si queremos ir agregando un <li> por cada iteración debemos utilizar un ‘v-for’ dentro del HTML. Este bucle funciona de manera parecida a un forEach:

* ‘v-for:”(por cada elemento) in (sobre que hay que iterar)”’
* Para nuestro ejemplo será ‘v-for=”tarea in tareas”’ (‘tarea’ es un nombre que ponemos nosotros, pero ‘tareas’ es como llamamos al arreglo dentro del objeto Vue (propiedad o ‘data’).
* Luego dentro del elemento HTML utilizamos {{tarea}}. Nuevamente lo importante es que coincida el nombre de la variable dentro de las llaves con la que estamos utilizando en el bucle (tarea)

## VIDEO 2 – Vue DevTools

Vue DevTools

## VIDEO 3 – Enlace Dinamico (v-bind)

Ahora vamos a crear un enlace dinamico. Para lograr esta funcionalidad debemos utilizar ‘v-bind’.

Para obtener un <a> como enlace dinamico debemos hacerle entender que el string dentro del ‘href=”(cadena)”’ hay que evaluarlo y hacerle una “correspondencia” y una interpolación con una propiedad del modelo. La directiva que se encarga de esto en Vue es ‘v-bind:’:

* ‘v-bind:href=”enlace”’.
* Ahora ‘enlace’ ya no es una cadena vacia si no que es la propiedad ‘enlace’ dentro del objeto Vue.

Desde el las Vue tools podemos modificar el valor de ‘enlace’ y vemos como cambio en tiempo real el direccionamiento del enlace de manera dinámica.

Podemos también utilizar un input y con la directiva ‘v-model’ enlazarlo con la propiedad ‘enlace’, por lo que podríamos ingresar el enlace a mostrar. Con esto logramos reactividad (o data binding) en las dos direcciones:

* Primero desde el modelo hacia el DOM porque el modelo tiene la propiedad ‘enlace’, lo que esta asociado con el ‘v-model’.
* Pero si cambio con el input, ahora la reactividad es desde el DOM hacia el modelo.

Ahora hagamos un ejemplo con una imagen con un alto y ancho dinamico. Debemos agregar 3 propiedades nuevas al modelo:

* baseURL:
* ancho:
* alto:

Para ingresar el alto y el ancho utilizaremos un input tipo range, para ingresar estos dos valores.

Ahora vamos a mostrar una **propiedad computada**. Esto sirve para hacer de “teaser”, para trabajar con valores del modelo y crear un retorno (un valor compuesto) que es lo que se interpola, algo transformado y listo para mostrar en el DOM. La propiedad conmutada se agrega de manera similar a ‘data’ y ‘methods’, siempre como un objeto. Para nuestro ejemplo:

methods: {

        guardarTarea() {

            this.tareas.push(this.nuevaTarea);

            this.nuevaTarea = '';

        }

Y en el HTML agregamos los siguientes elementos (bindeados):

        <input v-model="ancho" type="range">Ancho

        <br>

        <input v-model="alto" type="range">Alto

        <br>

        {{urlCompleta}}

        <br>

        <img :src="urlCompleta" alt="">

## VIDEO 4 – Renderizado de listas (v-for)

Renderizado de listas: para esto utilizamos lo que vimos rápidamente que es la directiva v-for. Esta directiva es capaz de iterar arreglos y objetos (es mas común iterando arreglos). La sintaxis es muy similar a la del forEach.

Tambien podemos utilizar el índice dentro del v-for de manera sencilla agregando ‘index’ al lado del elemento iterador (en nuestro ejemplo es tarea) y luego dentro del elem HTML hacer referencia dentro de otra doble llave:

<div id="app">

        <li v-for="(tarea, index) in tareas">{{tarea.nombre}} - {{index}}</li>

    </div>

En JS

        tareas: [

            {

                nombre: 'Aprender JS moderno',

                tiempo: 3,

            },

            {

                nombre: 'Aprender Vue.js',

                tiempo: 7,

            },

            {

                nombre: 'Reparar auto',

                tiempo: 22,

            },

        ]

Para iterar objetos se procede de una manera muy similar, pero ahora el elemento iterador (antes tarea) ahora debe hacer referencia a la propiedad que querramos acceder. Logicamente no haremos mas referencia sobre un arreglo si no que directamente sobre el objeto que queremos iterar: v-for=”(valor, propiedad) in (objeto)”

<div id="app">

        <p v-for="(valor, propiedad) in usuario">{{valor}}: {{propiedad}}</p>

    </div>

En JS:

usuario: {

            usearname: 'adrianocatena',

            roles: 'admin',

        },

## VIDEO 5 – Propiedades Computadas

Las operaciones que se especificaban o ponían entre las comillas de las directivas, ahora las podemos realizar en la instancia Vue e interpolar su resultado haciendo referencia al identificador de la propiedad conmutada.

Vamos a agregar un filtro por tiempo a nuestras tareas del ejemplo. Primero vamos a mostrar el tiempo en lugar del índice:

<li v-for="tarea in tareas">{{tarea.nombre}} - {{tarea.tiempo}}</li>

Ahora debemos filtrar el tiempo a nivel de DOM con un input tipo range y lo asociamos a una propiedad del modelo con un v-model.

<input v-model="tiempo" type="range">{{tiempo}}

Luego debemos crear una propiedad computada que utilizaremos como filtro de tareas, por lo que la llamaremos ‘filtroTareas()’. Hay que recordar que esto técnicamente es como una función, por lo que necesitamos agregar los paréntesis (con argumentos si fueran necesarios) y debe retornar algo. Para nuestro ejemplo también vamos a agregar una segunda instancia de filtro que va a ser el nombre aprovechando las virtudes de arreglos y strings de JS:

HTML

    <div id="app">

        <input v-model="tiempo" type="range">{{tiempo}} <br>

        <input v-model="busqueda" type="search" placeholder="Buscar tarea">

        <hr>

        <p v-for="(valor, propiedad) in usuario">{{valor}}: {{propiedad}}</p>

        <hr>

        <li v-for="tarea in filtroTareas">{{tarea.nombre}} - {{tarea.tiempo}}</li>

    </div>

JS

    data: {

        tiempo: 0,

        busqueda: '',

        usuario: {

            usearname: 'adrianocatena',

            roles: 'admin',

        },

        tareas: [

            {...

            },

        ]

    },

    computed: {

        filtroTareas() {

            return this.tareas.filter(tarea => {

                return tarea.tiempo <= this.tiempo && tarea.nombre.includes(this.busqueda);

            });

        }

    },

## VIDEO 6 – Eventos en Vue.JS (v-on)

Para este ejemplo continuamos con el código anterior y lo que vamos a hacer es agregar la funcionalidad para poder agregar tareas.

Para escuchar eventos JS Vue utiliza la directiva ‘v-on:’ (que se abrevia con ‘@’), y debemos asociar un método **con un evento** **en un elemento del DOM**, es decir, un evento sobre un elemento y una consecuencia (que hago si sucede ese evento sobre ese elemento).

El método que llamamos desde v-on debe existir en la instancia principal. Cuando el submit se desencadene en el elemento (el form para nuestro ejemplo) lanzara el método.

Es siempre conveniente realizar un bindeo bidireccional de los datos con v-model sobre el elem HTML que querramos sacar la información (v-model=”nuevaTarea”)

Tambien es importante saber elegir las diferentes variantes que se nos puede presentar en el manejo de eventos, como prevención o valores por defectos. Todas estas características están detalladas en la documentación oficial de Vue y son importantísimas para poder lograr el comportamiento que queremos y que la reactividad funcione correctamente. En nuestro ejemplo ‘.prevent’ previene que no se recargue la página:

        <form @submit.prevent="guardarTarea">

            <input type="text" placeholder="Tarea" v-model="nuevaTarea">

            <input type="submit" value="Guardar tarea">

        </form>

    data: {

        nuevaTarea: '',

        tiempo: 0,

        busqueda: '',

        usuario: {

            usearname: 'adrianocatena',

            roles: 'admin',

        },

        tareas: [

            {

                nombre: 'Aprender JS moderno',

                tiempo: 3,

            },

methods: {

        guardarTarea() {

            this.tareas.unshift({

                nombre: this.nuevaTarea,

                tiempo: Math.round(Math.random() \* 100),

            });

            this.nuevaTarea = null;

        }

    }

## VIDEO 7 – La instancia Vue

## VIDEO 8 – Vue Watchers

Nos permite con facilidad “observar” diferentes sucesos que se pueden dar en cualquier atributo dentro del objeto. Por ejemplo podemos ir viendo el ingreso de cierta información e ir reaccionando a esta en tiempo real. En el ejemplo vamos a ver como reaccionar al ingreso de una contraseña y en tiempo real ir reaccionando a esta “observando” el texto que ingresa en un input:

    <div id="app">

        <h3>Cambia tu contrasena</h3>

        <input type="text" v-model="password">

        <br>

        <p>{{estado}}</p>

    </div>

    data: {

        password: null,

        estado: "Introduce una contrasena...",

        noValidos: [

            "admin",

            "12345",

            "password"

        ],

    },

    watch: {

        password(nuevo, antiguo) {

            if (this.noValidos.indexOf(nuevo) > -1) {

                this.estado = "Contrasena NO valida!";

                return false;

            }

            if (nuevo.length < 5) {

                this.estado = "La contrasena debe tener al menos 5 caracteres";

                return false;

            }

            this.estado = "Contrasena correcta";

        }

    }

Es muy importante que la función a ejecutar dentro del watch tenga el mismo nombre que el atributo que estamos observando (en nuestro ejemplo el atributo observado es ‘password’ y por lo tanto la función dentro del watch se llama ‘password()’).

Los atributos que tiene esta función siempre se relacionan al estado nuevo de ese parámetro y al estado anterior. Es recomendable utilizar nombres fáciles que nos recuerde, el funcionamiento como ‘nuevo’ y ‘antiguo’. SIEMPRE el primer argumento es el nuevo valor y el segundo argumento es el valor anterior o antiguo.

## VIDEO 9 – Renderizado condicional (v-if)

Es la directiva ‘v-if=”(condición)”. Nos sirve para renderizar diferentes elementos en el DOM de manera dinámica en función de la condición que establezcamos. Tambien contamos con un ‘v-else’ y un ‘v-else-if’, los cuales funcionan exactamente igual a los condicional de cualquier lenguaje.

Es importante entender que de esta manera lo que estamos haciendo es eliminandor o agregando dicho elemento del DOM, por lo que obviamente esto puede consumir mas memori y por ende tener un rendimiento un poco peor. Es por esto que no se recomienda utilizar v-if dentro de un bucle muy largo o utilizado en grandes elementos.

Para esto Vue nos da la posibilidad de utilizar ‘v-show=”(condición)”’ de la misma manera que un v-if (sin else por supuesto), con la diferencia que no esta eliminado dicho elemento del DOM sino que modifica una propiedad de CSS para ocultarlo (display none), y por lo tanto es mucho mas eficiente en cuanto a memoria ya que no se esta inspeccionando el DOM.

    <div id="app">

        <form>

            <label for="nombre">Nombre</label>

            <input type="text" id="nombre">

            <label for="apellidos">Apellidos</label>

            <input type="text" id="apellidos">

            <label for="direccion">Dirección de envío</label>

            <select id="direccion" v-model="direccion">

                <option value="misma">Misma dirección que facturación</option>

                <option value="nueva">Introducir nueva dirección</option>

            </select>

            <template v-if="direccion === 'nueva'">

                <label for="nueva-direccion">Nueva dirección</label>

                <textarea id="nueva-direccion"></textarea>

            </template>

            <div>

                <input v-model="check" type="checkbox"><span>He leído y acepto la política de privacidad.</span>

            </div>

            <input v-show="check === true" type="submit" value="Enviar">

            <!-- <p v-else>Debes marcar el checkbox para enviar el formulario</p> -->

        </form>

    </div>

    data: {

        direccion: "misma",

        check: false,

    },

## VIDEO 10 – Filtros

Los filtros se utilizan para alterar o transformar texto. A grandes rasgos los filtros reciben un input, lo filtran y retornan un dato. Se pueden concatenar con el carácter ‘|’, conocido como “pipe”.

En nuestro ejemplo volvemos a hacer una petición de datos con fetch en la etapa de creación del objeto vue (created()) en la url que guardamos como atributo.

Luego con una propiedad computada concatenamos el titulo de la persona, el nombre y el apellido, lo cual lo mostramos a través de un ‘v-for’ dentro del HTML. Se utilizo una propiedad computada para transformar elementos del modelo y no tener que concatenar nada dentro del HTML ( {{usuario}} debería ser toda la concatenación y no es una buena practica).

Vamos a usar un filtro llamado ‘mayusculas’. ‘filters: {}’ es el objeto donde viven los filtros en la instancia Vue, y ‘mayusculas()’ es el nombre de la funcion que va a retonar ese input transformado, es por esto que necesitamos un argumento (el cual puede tener el identificador que querramos, para este ejemplo es ‘datos’).

En este caso el retorno es solamente pasar a mayúsculas la cadena recibida. En el html no se tiene que indicar este argumento que toma ya que se da cuenta solo que ese dato es ‘usuario’. Si en el HTML solo dejamos ‘{{usuario}}’ lo que va a mostrar es la concatenación en minúsculas, pero si utilzamos ‘{{usuario | mayúsculas}}’ va a tomar ese usuario y lo va a “pasar” por ese filtro, lo cual en este caso lo que hace es convertir a mayúsculas lo que le enviemos.

Tambien podemos seguir agregando filtros. Por ejemplo podemos recortar la cadena utilizando un nuevo filtro llamado “acortado” que lo único que hace es recortar la cadena de caracteres recibida (que en este caso es la que sale del 1er filtro ‘mayusculas()’) a un cierto numero de caracteres. Para esto debemos agregar la expresión del filtro al elemento HTML como ‘{{usuario | mayúsculas | acortado}}’.

Los filtros pueden aceptar argumentos específicos (como una función cualquiera), lo que los hace muy potentes y útiles. Por ejemplo podemos darle por argumento la cantidad de elementos que queremos que corte nuestro filtro de la cadena original, en el HTML debemos agregarle al filtro ese argumento como cualquier función: ‘{{usuario | mayúsculas | acortado(20)}}’ y en el objeto vue debemos asignar ese argumento como una variable y utilizarla dentro de la función misma (en nuestro ejemplo seria ‘cantidad’)

<div id="app">

<ul>

<li v-for="usuario in nombreCompleto">{{usuario | mayusculas | acortado(10)}}</li>

</ul>

</div>

created() {

fetch(this.url)

.then(respuesta => respuesta.json())

.then(respuesta => this.usuarios = respuesta.results.map(usuario => usuario.name));

},

data: {

url: 'https://randomuser.me/api/?results=50',

usuarios: [],

},

computed: {

nombreCompleto() {

return this.usuarios.map(usuario => `Nombre: ${usuario.title}: ${usuario.first} ${usuario.last}`)

}

},

filters: {

mayusculas(datos) {

return datos.toUpperCase();

},

acortado(datos, cantidad) {

return datos.substring(0, cantidad) + '...';

}

}

Lo mas importante de los Filtros es saber diferenciarlos de las propiedades computadas. La principal diferencia de los filtros es la REUSABILIDAD, ya que una propiedad computada se asocia a una o varias propiedades del modelo, pero solamente a esas por lo que quedan asignadas a estas y no se pueden cambiar. Si luego tengo mas propiedades en el modelo, o modifico la propiedad computada o tengo que crear mas propiedad computadas para esas propiedades del modelo. En cambio, los filtros, por su propia sintaxis a la hora de utilizarlos, se pueden aplicar sobre cualquier lugar pero sobre todo cuando tenemos las dobles llaves {{}} y hacemos interpolaciones, o como valor de ‘v-bind’.

## VIDEO 11 – Ajax con Axios

Vue no incorpora soporte de manera nativa a la librería Ajax, asi reduce el tamaño del propio nucleo de Vue. La alternativa mas popular en el entorno Vue es Axios.

Axios es un cliente HTTP basando en promesas que se puede usar en cualquier proyecto, no solo con Vue. Lo bueno de esta librería es que es soportado por todos los navegadores (incluido IE11), cosa que no ocurre con ‘fetch’. Podemos utilizar ‘fetch’, que también esta muy bien soportado, pero es un poco mas complejo ya que a la hora de obtener JSON hay que utilizar dos métodos, dos clausulas primero para obtener la respuesta y luego para llamar al método json, lo que lo hace mas extenso y tedioso (ademas ‘fetch’ no funciona en IE11).

Ahora vamos a utilizar el CDN (al igual que como hicimos con Vue) para agregarlo a un proyecto. Luego podemos ver como usa NPM:

<script src="https://unpkg.com/vue"></script>

<script src="https://unpkg.com/axios/dist/axios.min.js"></script>

Ahora vamos a lanzar los “request” a 'https://jsonplaceholder.typicode.com/todos' (básicamente al endpoint ‘todos’). En el modelo lo cargamos con la propiedad ‘url’, asi que cuando la instancia este “montada” (‘mounted’) vamos a hacer ese request.

Axios es bastante autoexplicativo, podemos ver la documentación oficial para ver ejemplos rapidos y sencillos de como implementarlo:

axios.get('(a donde)')

.then(function (response) { (que hacer cuando esa respuesta llegue)

// handle success

console.log(response);

})

.catch(function (error) { (si se presenta algún error)

// handle error

console.log(error);

})

.then(function () { (se ejecuta de todas formas)

// always executed

});

El endpoint es la url, cuando tengo la respuesta la añado al modelo. Si hay un error lo mostramos por consola.

mounted(){

axios.get(this.url)

.then(respuesta => this.tareas = respuesta.data)

.catch(error => console.error(error));

},

HTML

<pre>{{$data}}</pre>

Si ejecutamos vamos a ver que cargo todas las “tareas” dentro del modelo.

Gran parte de los ‘request’ a través de AJAX son GET y POST, por lo que de manera análoga, para hacer un POST en axios debemos utilizar justamente el metodo ‘post’. Nuevamente es muy útil ver los ejemplos de la documentación de la libreria que son muy simples y explicativos.

axios.post('/user', {

firstName: 'Fred',

lastName: 'Flintstone'

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

Para nuestro ejemplo, primero vamos a mostrar todas las tareas en una lista en una <ul> con <li>. Luego vamos a agregar un formulario, que no va a tener ‘action’ ya que lo capturamos via Ajax, con un input de texto y un botón. Ahora asociamos el input de texto al modelo ‘nuevaTarea’ con v-model. Recordamos añadir al formulario el método ‘guardarTarea’ con v-on.

<form @submit.prevent="guardarTarea">

<input type="text" placeholder="Tarea" v-model="nuevaTarea" required>

<input type="submit" value="Enviar">

</form>

<ul>

<li v-for="tarea in tareas">{{tarea.title}}</li>

</ul>

<pre>{{$data}}</pre>

Ahora debemos crear el método que estamos llamando (‘guardarTarea’). Primero hay que hacer un “post” a la url guardada como atributo, que lo hacemos con axios.post(). Debemos pasarle dos parámetros: el primero es justamente la url y el segundo es un objeto con los datos que le queremos enviar como “datos post”. Para nuestro ejemplo le enviaremos los datos de la ‘nuevaTarea’ y un numero aleatorio como ID.

Si todo va bien, debemos establecer una estructura que sea “respuesta” (la cual vamos a mostrar por consola) y la vamos a añadir al modelo en el arreglo ‘tareas[]’. Luego es importante “limpiar” la propiedad de ‘nuevaTarea’.

Si algo va mal, igual que antes, mostramos ese “error” por consola.

Si apretamos el botón Enviar en el formulario lanzaremos el método guardarTarea (previniendo el comportamiento por defecto de actualizar la pagina). Haremos un “post” a ‘url’ (endpoint). Luego enviamos el ‘title’ y el ‘userId’, y cuando todo funcione mostramos lo que llega de respuesta por consola y la añadimos a la matriz local (en este caso al modelo para que aparezca en la lista). Luego limpiamos el input y si hay algún error lo mostramos por consola.

methods: {

guardarTarea(){

axios.post(this.url, {

title: this.nuevaTarea,

userId: Math.floor(Math.random()\*10)+1,

})

.then(respuesta=>{

console.log(respuesta);

this.tareas.unshift({

title: this.nuevaTarea,

});

this.nuevaTarea = '';

})

.catch(error => console.error(error));

}

}

Axios se adapta perfectamente al estilo de Vue, por lo que es muy importante tenerlo a mano y considerar su documentación:

* [axios/axios: Promise based HTTP client for the browser and node.js (github.com)](https://github.com/axios/axios).

## VIDEO 12 – Data binding con estilos y clases (v-bind)

Ya sabemos que con la directiva ‘v-bind’ (abreviada solo con ‘:’) podemos convertir atributos html estatico en dinámicos. Uno de los atributos mas útiles a modificar es el de “class”, ya que nos permite aplicar clases de forma dinámica, es decir, aplica esta clase si se da una condición o aplica esta otra se si de esta condición, y por supuesto con data binding y reactividad de Vue.

Hay diferentes formas de tener clases y estilos dinámicos (o databinding con clases y estilos). La primera de esas formas es a traves de la sintáxis de objeto que es la mas sencilla de utilizar.

En nuestro ejemplo lo que hacemos es que se muestre un mensaje si el usuario esta loggeado o no, utilizando un checkbox y un pequeño cuadro que va modificando en función de este check. La propiedad ‘logged’ es la que indica si el usuario esta loggeado o no, donde también asociamos el checkbox para conmutar su valor.

“mensaje” es una propiedad computada que lo que hace es generar el texto a mostrar en función de ‘logged’.

ES IMPORTANTE CARGAR LA MENOR CANTIDAD DE LOGICA DENTRO DE LA PRESENTACION O RENDERIZADO, ES MUY RECOMENDABLE DEJAR TODO ESTO DEL LADO DE LA LOGICA.

Con la sintaxis de objeto es muy simple, solo expresamos un objeto con sus llaves de apertura y de cierre, la propiedad de ese objeto es la clase que queremos modificar y el valor de esa propiedad es la expresión a evaluar. En nuestro caso creamos dos clases de estilo: “logged-in” y “logged-out”. Y tambien es importante aclarar que pueden convivir tranquilamente para un mismo elemento clases estaticas y clases dinámicas. Es recomendable crear también una propiedad computada para las posibilidades de clases, es decir, crear una propiedad computada para “componer” el nombre de la clase que vamos a aplicar. Esta propiedad la llamamos “estilos”

<span>Login/out</span>

<input type="checkbox" v-model="logged">

<article class="user-info" :class="estilos">

{{mensaje}}

</article>

<pre>{{$data}}</pre>

Podemos retornar de muchísimas formas el nombre de la clase desde la propiedad ‘estiilos()’, todas son igual de validas y responden de la misma manera, es una cuestión de gustos como lo puede implementar cada uno. En nuestro ejemplo utilizamos string template pero también es muy útil usar el operador ternario.

el: "#app",

data: {

logged: true,

mensajes:{

in: "Logeado como @adrianoCatena.",

out: "Te has desconectado.",

},

},

computed: {

mensaje(){

return this.logged ? this.mensajes.in : this.mensajes.out;

},

estilos(){

// return {

// 'logged-in': this.logged,

// 'logged-out': !this.logged,

// }

// return {

// this.logged ? 'logged-in' : 'logged-out';

// }

return `logged-${this.logged ? 'in': 'out'}`;

},

}

También se puede aplicar la sintaxis de arreglo, pero sería muy útil si tuviéramos una lista de clases a aplicar y por ende podríamos crear un arreglo para expresarlas a todas.

## VIDEO 13 – Data binding en formularios

Crear formularios con cierta complejidad es super tedioso, por suerte Vue y su directiva ‘v-model’ funcionan a la perfeccion.

En la documentación vemos un montón de ejemplos de los elementos mas utilizados lo que podemos tomar como base. Nuestro ejemplo vamos a maquetar varios elementos HTML utilizados en general en distintos tipos de formularios.

Como vemos es muy fácil enlazar los elementos con el modelo con la directiva ‘v-model’ que ya vimos anteriormente.

Particularmente para los input radio debemos enlazar cada uno de estos con la propiedad que modifican (form.contacto).

Por otro lado los checkboxes no van a devolver el elemento que seleccionamos, y si seleccionamos varios nos va a generar un arreglo con todos los seleccionados.

Para el select solo debemos enlazar el elemento “padre” a la propiedad que querramos con v-model (form.productos). Por defecto el select esta vacio, por lo tanto la propidad asi lo esta, por eso es recomendable darle un valor por defecto en el modelo de Vue a la propiedad que sea igual a una de las opciones del select. Tambien tiene la particularidad de que si le damos el atributo ‘multiple’ al elemento select, este le dará a la propiedad que tenga enlazada los valores de todos las opciones seleccionadas como si fuera un arreglo (de manera similar a los checkboxes).

Yendo mas a profundidad podemos anexar lo que vimos anteriormente de clases dinámicas para mejorar el aspecto del formulario, en este caso agregándole estilos al formulario en función de si esta vacio cierto campo con ‘v:bind’ (‘:’). En nuestro ejemplo la clase que le da estilo se llama ‘invalido’, pero podríamos hacer cualquier cosa que necesitemos.

<main id="app">

<form>

<label for="nombre">Nombre</label>

<input type="text" required id="nombre" v-model="form.nombre" :class="{invalido: !form.nombre}">

<label for="apellidos">Apellidos</label>

<input type="text" required id="apellidos" v-model="form.apellidos" :class="{invalido: !form.nombre}">

<label for="mensaje">Mensaje</label>

<textarea id="mensaje" required v-model="form.mensaje" :class="{invalido: !form.nombre}"></textarea>

<label for="">Hora de contacto</label>

<span>Mañanas</span>

<input type="radio" value="mañanas" v-model="form.contacto">

<span>Tardes</span>

<input type="radio" value="tardes" v-model="form.contacto">

<span>Noches</span>

<input type="radio" value="noches" v-model="form.contacto">

<label for="productos">Producto/s</label>

<select id="productos" v-model="form.productos">

<option value="producto1">Producto 1</option>

<option value="producto2">Producto 2</option>

<option value="producto3">Producto 3</option>

</select>\

<label for="acepta">Aceptacion de condiciones</label>

<span>Acepto el blablabla</span>

<input type="checkbox" id="acepta" v-model="form.acepta">

<input type="submit" value="Enviar" v-if="form.acepta">

</form>

</main>

Aprovechando que tenemos todo el “estado” (lo de estado es relativo, ya que para eso se utiliza Vuex para manejar el estado dentro de una aplicación Vue) de este formulario de forma reactiva dentro de la propiedad form dentro del modelo (que no es mas que un objeto).

Podríamos, por ejemplo, observar cambios dentro de esta propiedad y guardarlos en el almacenamiento local del navegador. Esto lo realizamos desde afuera del objeto (o variable) Vue con ‘watch’.

Hemos visto como usar ‘watch’ desde adentro de la instancia Vue, pero al guardar la instancia dentro de una ‘const’ tenemos una forma mas “imperativa” de observar cierta propiedad. En nuestro ejemplo vamos a observar la propiedad ‘form’, por lo que tenemos que ejecutar la línea app.$watch(‘form’,…).

Como segundo parámetro se ejecuta una función, que como recordamos nos permite tener dos argumentos que son: el nuevo valor del parámetro a observar, y el valor anterior o antiguo. De primera mano lo que podemos hacer es solo mostrar lo que se va modificando en la consola, pero la clave aquí es poner ‘deep’ como un valor de las opciones que le pasamos a ‘watch’. Esto lo hacemos porque queremos que observe los cambios dentro del las propiedas internas de este objeto (para ver mas detalles de esta características podemos ver la documentación oficial del watch). Esto quiere decir que si cambiamos cualquier cosa dentro de cualquiera de una de las propiedades que están dentro de ‘form’ en el modelo, deberían mostrarse en consola. Tambien podemos mostrar algún parámetro en particular del objeto form haciendo solo referencia a este (por ejemplo ‘form.nombre’). De esta manera estoy actualizando o mostrando cada vez que se da algún cambio.

Ahora seria tan sencillo como guardar cada vez que ocurra un cambio en el almacenamiento local del navegador. Para eso utilizamos el método ‘Window.localStorage’, que es un método con una sintaxis muy sencilla que tiene un get, un set, remove y clear. Podemos profundizar este método viendo la documentación oficial. En nuestro caso veremos setItem y getItem, pero el tema esta que cuando guardamos estamos dándole un valor a esa “clave” del almacenamiento local (‘form’ en nuestro ejemplo) pero luego tiene que tener un string pero nosotros queremos guardar un objeto, asi que con stringify debemos convertir este objeto en un string (también se puede ver mucho mas en detalle el método stringify que forma parte de JSON).

Vale aclarar que cuando querramos hace una aplicación el nombre que utilicemos para guardar este objeto convertido a string debe ser algo un poco mas único (no ‘form’ como en este ejemplo).

Si quisiéramos cargalo cuando se inicia la instancia Vue con ‘created()’ o con ‘mounted()’ (dependiendo el ciclo de vida de la instancia Vue que nos sirva). Para nuestro ejemplo utilizaremos mounted() en conjunto con el otro método de ‘localStorage’ para obtener datos: getItem. De la misma manera que tuvimos que convertir un objeto a cadena, ahora debemos convertir la cadena leida a un objeto y guardarlo en el modelo, para lo cual debemos utilizar el metodo .parse() del objeto JSON. Es importante que la ‘key’ que grabo sea igual a la que leo mas tarde, ya que de ahí entiende que esta leyendo.

const app = new Vue({

mounted(){

if(window.localStorage.getItem('form'))

this.form = JSON.parse(window.localStorage.getItem('form'));

},

el: "#app",

data: {

form:{

nombre: '',

apellidos: '',

mensaje: '',

contacto: '',

productos: 'producto1',

acepta: false,

}

},

});

app.$watch('form', (nuevo, antiguo) => {

window.localStorage.setItem('form', JSON.stringify(nuevo));

//console.log(nuevo.nombre);

}, {deep: true});

En las devTools podemos ver el Storage dentro de la pestaña ‘Application’.

## VIDEO 14 – Introducción a componentes

El siguiente paso es aprender a utilizar los Componentes. Estos nos permiten crear un elemento personalizado, como por ejemplo un slider, una diapositiva o un usuario (entre tantas cosas), y reutilizarlos en diferentes escenarios.

Vamos a crear un componente muy sencillo que representa una entrada de un blog. Para esto en nuestro código JS creamos en el modelo, un arreglo de 3 objetos, cada uno con el titulo y el contenido a mostrar dentro de la propiedad ‘posts’.

data: {

posts: [

{

titulo: 'Esto es un titulo de post',

contenido: 'Lorem ipsum dolor sit amet, ',

},

{

titulo: 'Esto es un nuevo titulo de post',

contenido: 'Lorem ipsum dolor sit amet, ',

},

{

titulo: 'Esto es otro titulo de post',

contenido: 'Lorem ipsum dolor sit amet, ',

},

],

},

Ahora lo mostramos directamente dentro de un elemento lo hacemos de manera muy sencilla, primeramente, con un <article> en el cual se lo itera con un ‘v-for’ sobre los ‘posts’. Esto ya lo vimos reitreradas veces. Si grabamos vemos como renderiza de manera correcta lo que tenemos en código.

<article class="post" v-for="post in posts">

<h2 class="post\_\_titulo">{{post.titulo}}</h2>

<p class="post\_\_contenido">{{post.contenido}}</p>

</article>

Pero también sabemos que un post tiene bastante mas que un titulo y un contenido simple, por lo que imaginemos tener que mostrar toda la información en un simple archivo HTML y que posiblemente tengamos que usar lo mismo en otro lugar. Por ejemplo podemos mostrar los posts y en otra pagina mostrar solo los últimos 3 posts, tendríamos que reescribir todo este código de nuevo por cada vez que querramos mostrar alguno, tendríamos que utilizar la misma estructura y estilos cada que mostremos un post. Para esto lo que podemos hacer es “abstraer” toda esta estructura y toda la lógica que conlleva a un **componente**.

En la documentación oficial podemos encontrar los aspectos fundamentales para la creación de componentes:

// Define a new component called button-counter

Vue.component('button-counter', {

data: function () {

return {

count: 0

}

},

template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'

})

Estos tienen algunas características especiales que los diferencian de las instancias. Principalmente que no tienen un elemento al que “montarse’, si no que tiene un ‘template’ ya que por si propia naturaleza son reutilizables y no tiene sentido atarlo a un único elemento. El modelo debe ser un función porque si no todos los componentes heredarían o usarían el mismo modelo, compartirían el “estado”, algo que solo es deseable en pocos casos.

Vamos a crear un componente llamado ‘post’, y luego utilizaremos la misma estructura que habíamos creado para el elemento HTML dentro de ‘template’. ES MUY IMPORTANTE DEFINIR EL COMPONENTE ANTES DE LA INSTANCIA VUE!

Al componente se le tiene que pasar como primer argumento el nombre (‘post’ en nuestro ejemplo) y en segundo argumento un objeto que tiene como mínimo una propiedad ‘template’ que es lo que se va a renderizar en la pagina.

En este punto es donde vamos a colocar la estructura antes creada. Es importante tener en cuenta que en este caso hay que modificar algunas cosas.

Primero que no tiene sentido hacer la iteración v-for dentro del template, es una mala practica y siempre se utiliza directamente sobre algún elemento.

Segundo debemos agregar en el HTML solo la etiqueta con el nombre del template: <post></post> en nuestro ejemplo (recordar que en este caso aplicamos dentro de esta etiqueta el v-for).

Tercero debemos pasar “desde arriba hacia abajo” una propiedad con un valor a evaluar, por lo que la tenemos que definir primero como una ‘props’, a la cual luego voy a enlazar para decirle cuales van a ser sus atributos. Ahora en el elemento HTML podemos bindear post (el que utilizamos para iterar post in posts) con el post que definimos recién en ‘props’:

* v-for=”post in posts” :post=”post”
* props: [‘post’], 🡪 este es el que se va a renderizar y que hacemos referencia en ‘template’

Vue.component('post', {

props: ['post'],

template:` <article class="post">

<h2 class="post\_\_titulo">{{post.titulo}}</h2>

<p class="post\_\_contenido">{{post.contenido}}</p>

</article>`,

});

<main id="app">

<post v-for="post in posts" :post="post"></post>

<pre>{{$data}}</pre>

</main>

Queda un código mucho mas limpio y si quisiera actualizar tengo que ir solamente al template, y no cambiar cada uno de las veces que lo utilice.

## Ejemplos

<body>

<!-- este id es el que utilizamos en el objeto vue 'el' -->

<div class="container-fluid" id="shopping-list">

<div>

<!-- Se pueden utilizar ciertos metodos o funciones de JS dentro de {{}} -->

<!-- dentro de {{}} llamamos al metodo o propiedades del objeto vue -->

<h1>{{ header.toLocaleUpperCase() }}</h1>

<button v-if="estado==='default'" @click="cambiarEstado('edit')">Agregar item</button>

<button v-else @click="cambiarEstado('default')">Cancelar item</button>

</div>

<div v-if="estado==='edit'">

<!-- 'v-on:' se reemplaza por '@' -->

<!-- 'guardarItem' es un metodo del objeto vue, que a su vez es una funcion -->

<!-- v-model enlaza el html bidireccionalmente con el bojeto vue, particular con la propiedad 'nuevoItem' -->

<input v-model="nuevoItem" type="text" placeholder="Agregar un nuevo item" @keyup.enter="guardarItem">

<p>{{ contadorCaracteres }}/200</p>

<!-- permite enlazar (bindear) una variable de Vue con un atributo específico de una etiqueta HTML -->

<!-- 'v-bind' se reemplaza directamente con ':' -->

<button :disabled="nuevoItem.length===0" @click="guardarItem">Guardar item</button>

</div>

<p v-if="items.length===0">Buen trabajo, has hecho todas las compras!</p>

<ul class="list-group" v-else>

<!-- se utiliza el v-for al igual que un foreach -->

<!-- Como agregar las clases dinamicas es cuestion personal -->

<!-- Sintaxis de arreglo (ternario): si se necesita un poco mas de control en la plantilla -->

<!-- <li v-for="item in items" -->

<li v-for="item in invertirItems"

:class="[item.comprado ? 'list-group-item list-group-item-light' : 'list-group-item']"

@click="alternarCompra(item)">

{{ item.etiqueta }}

</li>

<!-- Sintaxis de objeto: usando condicionales simples para mantener limpia la plantilla -->

<!-- <li v-for="item in items" :class="{list-group-item disabled: item.comprado}"> {{ item.etiqueta }} </li> -->

<!-- las clases tambien se pueden asignar de manera estatica

class="[item.comprado ? 'list-group-item disabled' : 'list-group-item', 'prioridad']">

class="prioridad" :class="[item.comprado ? 'list-group-item disabled' : 'list-group-item']"> -->

</ul>

</div>

<script src="https://unpkg.com/vue"></script>

<script>

/\* objeto vue, el nombre puede ser cualquiera y se recomienda que sea explicito \*/

var lista = new Vue({

/\* 'el' es igual al id del elemento html que linkeamos \*/

el: '#shopping-list',

/\* data vendrian a ser como las propiedades de la clase vue \*/

data: {

estado: 'default',

header: 'lista de compras',

nuevoItem: '',

items: [

{

etiqueta: '10 sombreros de fiesta',

comprado: false,

prioridadAlta: false,

},

{

etiqueta: '2 tableros de juegos',

comprado: true,

prioridadAlta: false,

},

{

etiqueta: '20 vasos',

comprado: false,

prioridadAlta: true,

},

]

},

/\* se utilizan para las propiedades computadas, elementos que calculan reactivamente algun dato. las propiedads computadas son herramientas poderosas para encapsular transformaciones de datos y manipulaciones que se mantienen sincronizadas con los datos a los que hace referencia \*/

/\* Es importante retornar siempre un valor desde una propiedad computada \*/

computed: {

invertirItems() {

return this.items.slice(0).reverse();

},

contadorCaracteres() {

return this.nuevoItem.length;

}

},

/\* logicamente, methods vendrian a ser todos los metodos de la clase vue \*/

methods: {

guardarItem: function () {

this.items.push({

etiqueta: this.nuevoItem,

comprado: false,

prioridadAlta: false,

});

this.nuevoItem = '';

},

cambiarEstado: function (nuevoEstado) {

this.estado = nuevoEstado;

this.nuevoItem = '';

},

alternarCompra: function (item) {

item.comprado = !item.comprado;

}

}

/\* Cuando necesitamos cambiar datos, tenemos que usar metodos \*/

/\* Cuando necesitamos cambiar la presentacion de datos existentes, usamos propiedades computadas \*/

});

</script>

</body>