<codoacodo/>

Agencia de Aprendizaje a lo largo de la vida

FULL STACK PYTHON
Clase 20

Vue 2

BA

DOM y eventos

Les damos la bienvenida

Vamos a comenzar a grabar la clase

Clase 19

Introducción a Vue

- Introducción a Vue.js. ¿Qué es?
- Renderizado.
- Modificación del DOM.
- Instalación. CDN.
- Directivas condicionales,
- estructurales y de atributo.
- Métodos y eventos.
- Conceptos claves.

Clase 20

DOM y Eventos

- Aplicaciones Reactivas. Reactividad en 2 sentidos.
- Propiedades computadas.
- Componentes.
- Watchers.
- Acceder a los elementos del DOM utilizando $refs
- Concepto MVC y MVVM.

Clase 21

SPA y Asincronía

- Introducción a SPA.
- SPA. ¿Qué es y qué beneficios tiene?
- Ejemplo práctico de un SPA en Vue.
- Enviar y pedir datos a un servidor
- Asincronía.
- Consumo de API REST a través de fetch y Axios.

Aplicaciones reactivas

Una de las características que diferencian a Vue es su sistema de reactividad. Los modelos simplemente son objetos de JavaScript. Cuando se modifican, se actualiza la vista. Esto hace que el gestor de estados sea simple e intuitivo.

Vue.js proporciona opciones para agregar reactividad a las propiedades, que se agregan dinámicamente, mediante el uso de *observadores*.

Reactividad en 2 sentidos

Partiremos del ejemplo del array de frutas, donde agregamos con la directiva **v-for** las frutas del array de objetos y le agregamos un input y un botón para agregar la fruta que deseamos al array de objetos:

```
<div id="app">
    <input type="text" v-model="nuevaFruta">
    <button v-on:click=agregarFruta()>Clic para agregar fruta</button>
    <ul>
        <li v-for="fruta in frutas">
            {{ fruta.nombre }} - {{ fruta.cantidad }}
        </li>
    </ul>
</div>
```

Reactividad en 2 sentidos

Agregamos la propiedad `nuevaFruta: ''` dentro de **data** y también agregaremos el método `agregarFruta()`.

Con el método **push** agregaremos un nuevo elemento al array de objetos (frutas), con los pares clave: valor.. El nombre de la fruta es el de **nuevaFruta** y utilizamos el this para hacer referencia a esa instancia. Además sumamos que la nuevaFruta inicie con cantidad 0.

```
const { createApp } = Vue
createApp({
    data() {
        return {
            frutas: [
                { nombre: "naranja", cantidad: 10 },
                { nombre: "banana", cantidad: 0 },
                { nombre: "durazno", cantidad: 3 }
            ],
            nuevaFruta: ''
        }
    },
    methods: {
        agregarFruta() {
            this.frutas.push({ nombre:
this.nuevaFruta, cantidad: 0 })
        }
    }
}).mount('#app')
```

Reactividad en 2 sentidos

Una mejora que se puede hacer es que se limpie la caja de texto cuando agregamos un elemento. Esto se logra agregando en el método **agregarFruta()** esta instrucción:

```
this.nuevaFruta= ''
```

Podemos hacer una mejora para que no nos permite agregar una fruta hasta que no se haya completado la caja de texto:

```
agregarFrutaConIf(){
if (this.nuevaFruta != "") {
    this.frutas.push({ nombre: this.nuevaFruta, cantidad: 0 })
    this.nuevaFruta= ''
}
```

Manejo de eventos

Como vimos anteriormente, para interactuar con los usuarios, podemos usar la directiva **v-on** para adjuntar detectores de eventos que invocan métodos en nuestras instancias de Vue.

En definitiva, usamos la directiva **v-on** para escuchar eventos DOM y ejecutar código cuando se activan.

Eventos | Modificadores de teclas

Vue es capaz de interceptar y atender eventos disparados al presionar determinadas teclas. Se pueden usar los eventos **keyup** y **keydown**, con el código de la tecla esperada:

```
<!-- solo llame a `vm.submit ()` cuando el `keyCode` es 13 -->
<input v-on:keyup.13="submit">
```

keyup se dispara al soltar una tecla, y **keydown** al presionarla. Las teclas más comunes poseen una descripción propia, para evitar la necesidad de recordar su keyCode: **.enter**, **.tab**, **.delete** (captura ambas teclas “Delete” y “Backspace”), **.esc**, **.space**, **.up**, **.down**, **.left** y **.right**.

Eventos | Modificadores de teclas

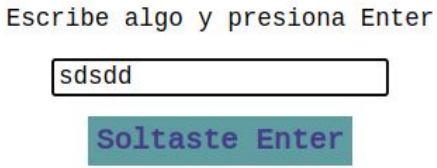
El siguiente ejemplo muestra cómo invocar un método cada vez que se levanta una tecla. Concretamente, al soltar Enter:

```
<div id="app" class="container">
   <p>Escribe algo y presiona Enter</p>
   <!-- En keyup.enter invertimos el valor lógico de "data"-->
   <input v-on:keyup.enter="data1 = !data1"></input><br><br>
   <span v-if="data1" class="tituloTabla">Soltaste Enter</span>
</div>
```

Al presionar Enter se ejecuta el código que invierte el estado lógico de data1. Y el que está debajo muestra o no el texto correspondiente.+info

Y en la instancia Vue tenemos, por supuesto, declarada la propiedad **data1**:

```
const miAplicacion = Vue.createApp({
   data() {
       return {data1: false}
          }
       }).mount("#app")
```



Propiedades computadas

Permiten que VUE agregue una función que realice alguna operación sobre sus propiedades, que se actualizan cuando hay algún cambio en el documento HTML. Por ejemplo, podemos sumar la cantidad de frutas:

```
computed: {
    sumarFrutas() {
       this.totalFrutas = 0
       for (fruta of this.frutas) {
           this.totalFrutas += fruta.cantidad
             }
       return this.totalFrutas
       }
   }
```

Así se agregan los campos computados a Vue. Puede haber varios, y se separan unos de otros usando comas.
En este ejemplo, la propiedad "**totalFrutas**" se computa cada vez que se invoca el método **sumarFrutas**. Recorre todo el arreglo "**frutas**" sumando la cantidad que hay de cada una.

En el HTML tenemos **{{sumarFrutas}}** . Cada vez que va a ser renderizado, se ejecuta el código de **sumarFrutas()** dentro del bloque **computed: {}**

Propiedades computadas

En el código HTML se realiza el renderizado que muestra el total computado:

```
<!-- total frutas -->
<p>Total de frutas : <b>{{sumarFrutas}}</b></p>
```

Lista de frutas:

Agregar nueva fruta

El campo no puede estar vacío.

- Hay 13 unidades de manzanas.
  + -
- Hay 7 unidades de naranjas.
  + -

Total de frutas : **20**

Este valor es el computado por Vue, a partir de los valores que posee el arreglo que contiene los nombres y cantidades de cada fruta.

Propiedades computadas

Como en los métodos, dentro de **computed** se definen todos los campos computados separados por comas. Y dentro de cada una se emplea **this** para acceder a los valores de las propiedades que se han declarado en **data()** o en los métodos de la instancia Vue.

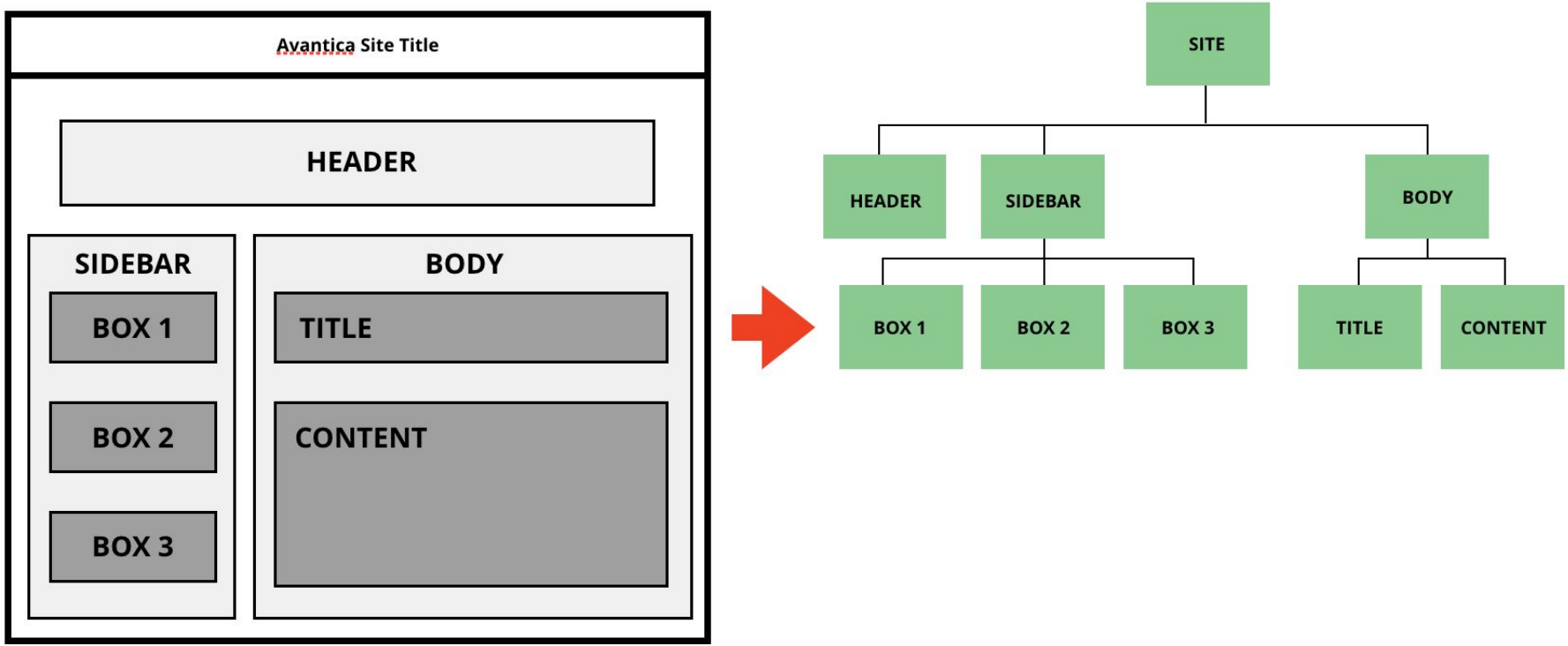
Es importante saber que los campos computados **no pueden recibir valores** mediante parámetros desde fuera de la función. Si es necesario pasar un valor, en lugar de un campo computado se debe crear un **método**.

Otra característica importante de los campos computados es que siempre tienen que devolver un valor. No puede existir un campo computado sin su correspondiente **return**.

Componentes

El sistema de componentes de Vue.js nos permite crear aplicaciones a gran escala compuestas por componentes pequeños, autónomos y reutilizables. Dado que los componentes son instancias de Vue reutilizables, aceptan las mismas opciones que cualquier instancia Vue, como datos, propiedades computadas, métodos, etc. La siguiente figura muestra cómo una interfaz de aplicación típica se abstrae en tres componentes: encabezado, barra lateral y cuerpo.

Sistema de componentes



Creación de componentes

Los **componentes** son una de las características importantes de Vue.JS que ayudan a crear elementos personalizados, que se pueden reutilizar en HTML.

Dentro del componente se agrega un template el cual tiene asignado código HTML. Esta es la manera de registrar globalmente un componente en Vue.JS, que puede ser reutilizado en cualquier instancia Vue. El nombre asignado a las etiquetas será reemplazado por el código del **template** escrito en la creación del componente.

Creación de componentes

En el HTML no existe la etiqueta <saludo>. Pero podemos crearla utilizando componentes Vue:

```
// Creamos un componente:
const componente1 = {
   template: `<h4>Hola {{usuario}}</h4>`,
   data() {
       return { usuario: "Codo a Codo" }
   }
}
```

```
// Creamos nuestra aplicación.
const miAplicacion = Vue.createApp({
   components: {
       'saludo': componente1
   }
}).mount("#app")
```

El componente c**omponente1** creado se agrega a la instancia Vue mediante la palabra clave **components:** . Se pueden crear y agregar todos los componentes que sean necesarios, cada uno con su **template** y, dentro de la instancia de Vue, su propio nombre.

Creación de componentes

Una vez creado el o los componentes, los utilizamos en nuestro código HTML como si fuese una etiqueta más:

```
<!--Este es el contenedor donde funciona Vue.js -->
<div id="app">
   <!--<saludo> es un componente que hemos creado.-->
   <saludo></saludo>
</div>
```

Al renderizar la página, Vue reemplaza este par de etiquetas por el contenido que surja de la ejecución del código del componente asociado. En nuestro caso, veremos el siguiente texto:

Hola Codo a Codo

Creación de componentes

Los componentes pueden tener métodos. Agreguemos un componente más al ejemplo anterior:

```
// Creamos otro componente:
const componente2 = {
    template: `<div
        v-on:mouseover = "cambiarNombre()"
        v-on:mouseout = "reestablecerNombre()">
        <h4><span id="nombre">{{titulo}}</span></h4>
    </div>`,
    data() {
        return { titulo: "Componentes en Vue.js" }
    },
    methods: {
        cambiarNombre() { this.titulo = "¡El mouse está sobre el contenido del DIV!" },
        reestablecerNombre() { this.titulo = "Componentes en Vue.js" }
    }
}
```

componente2 será otro componente de nuestro proyecto.

Lo definimos de manera similar a **comonente1**, pero en este caso agregamos **data()** y **métodos**.

Creación de componentes

Debemos declarar el nuevo componente en la instancia Vue, y luego podemos disponer del componente en el cuerpo del HTML::

```
// Creamos nuestra aplicación.
const miAplicacion = Vue.createApp({
   components: {
       'saludo': componente1,
       'mouse': componente2
   }
}).mount("#app")
```

```
<!--Este es el contenedor donde funciona Vue.js -->
<div id="app">
   <!--<saludo> y <mouse> son componentes que hemos creado.-->
   <saludo></saludo>
   <mouse></mouse>
</div>
```

Al renderizar la página, vemos como el segundo componente renderiza diferente contenido dependiendo de la posición del mouse:

Hola Codo a Codo

¡El mouse está sobre el contenido del DIV!

Watchers

Un **watcher** (observador) en Vue es una característica especial que nos permite observar algunos datos y realizar acciones específicas **cuando cambian**. Es una forma más genérica de observar y reaccionar a los cambios de datos en la instancia de Vue.

Con los observadores, no solo podemos observar una propiedad y realizar acciones personalizadas; también podemos acceder al valor antiguo desde el que está cambiando esta propiedad en particular, así como al nuevo valor al que ha cambiado.

Los Watchers se aplican a los datos que cambian. Por ejemplo, elementos input en formularios. Los Watchers se encargan de manejar cualquier cambio de datos haciendo que el código sea simple y rápido.

Watchers

Veamos cómo implementarlos. En el ejemplo, se crean dos **textboxes**, uno con kilómetros y otro con metros:

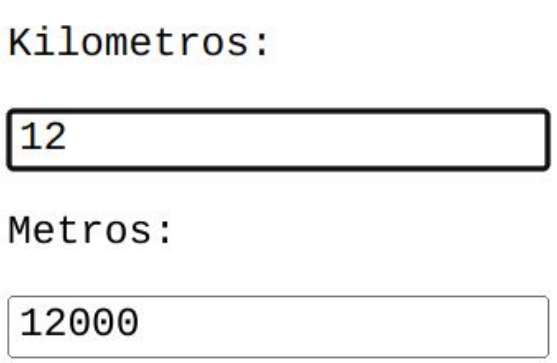
```
<!-- Este es el contenedor donde funciona Vue.js -->
<div id="app">
   <p>Kilometros:</p><input type="text" v-model="kilometros">
   <p>Metros:</p><input type="text" v-model="metros">
</div>
```

En **data()** ambas propiedades son inicializadas en cero. Existe dentro de la instancia Vue un objeto **watch** que se crea con dos funciones, cuyo objetivo es convertir de kilómetros a metros y viceversa. Cada vez que se ingrese algún valor en los **inputs**, watch se encarga de actualizarlos, calculando lo que está declarado en las funciones.

Watchers

```
const miAplicacion = Vue.createApp({
    //DATOS de la aplicación
    data() {
        // Definimos que devuelve nuestra aplicación
        return { kilometros: 0,
                 metros: 0 }
    },
    // Defino watch:
    watch: {
        kilometros(valor) {
            this.kilometros = valor
            this.metros = valor * 1000
            },
        metros(valor) {
            this.kilometros = valor / 1000
            this.metros = valor
            }
        }
}).mount("#app")
```

Estos ***watchers*** sólo deben usarse cuando las propiedades calculadas resultan insuficientes, situaciones en las que necesitamos asincronía o se trata de operaciones muy costosas para cambiar el valor de los datos.



Acceder a los elementos del DOM utilizando $refs

Si necesitamos acceder a elementos del DOM, las instancias de Vue.JS cuentan con diversos mecanismos, uno de ellas es **$refs**. Visto en código sería algo como:

```
app.$refs
vm.$refs
```

$refs es un objeto, dentro de él se van a almacenar todos los elementos del DOM que cuenten con el atributo "**ref**". El atributo "**ref**" vendría a ser algo así cómo proporcionarle un ID al elemento.

```
<input ref="entrada"></input>
```

Acceder a los elementos del DOM utilizando $refs

"**app**" o "**vm**" representan la instancia misma de Vue. También puedes utilizar la palabra reservada "**this**" para referirte a la instancia. **$refs** es una propiedad de la instancia. Se pueden tener varios elementos con el atributo "ref", siempre y cuando el valor del atributo sea diferente para cada elemento:

```
<input ref="entrada"></input>
<input ref="entrada2"></input>
```

Para acceder al objeto que almacena estos elementos lo hacemos con `app.$refs`, y para acceder al elemento dentro del objeto usamos `app.$refs.entrada`

Acceder a los elementos del DOM utilizando $refs

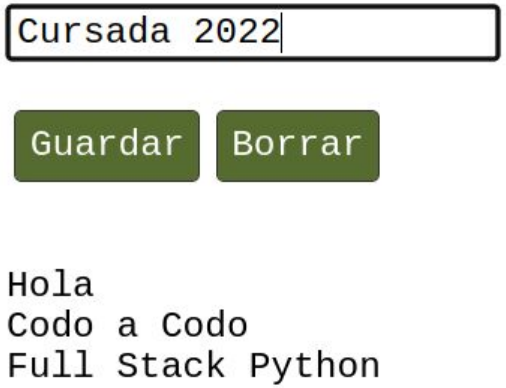
Apliquemos lo anterior con una aplicación sencilla. Lo que hace la aplicación es añadir a un párrafo el texto que escribamos en una entrada:

```
<!--Este es el contenedor donde funciona Vue.js -->
<div id="app">
   <input type="text" ref="texto"><br><br>
   <button type="button" v-on:click="addText">Guardar</button>
   <button type="button" v-on:click="delText">Borrar</button>
   <p ref="textField"></p>
</div>
```

Colocamos los atributos “ref” a los elementos cuyas propiedades queremos acceder, que en este caso son el input y el párrafo. Y por último ponemos los botones a la escucha, para invocar los métodos correspondientes.

Acceder a los elementos del DOM utilizando $refs

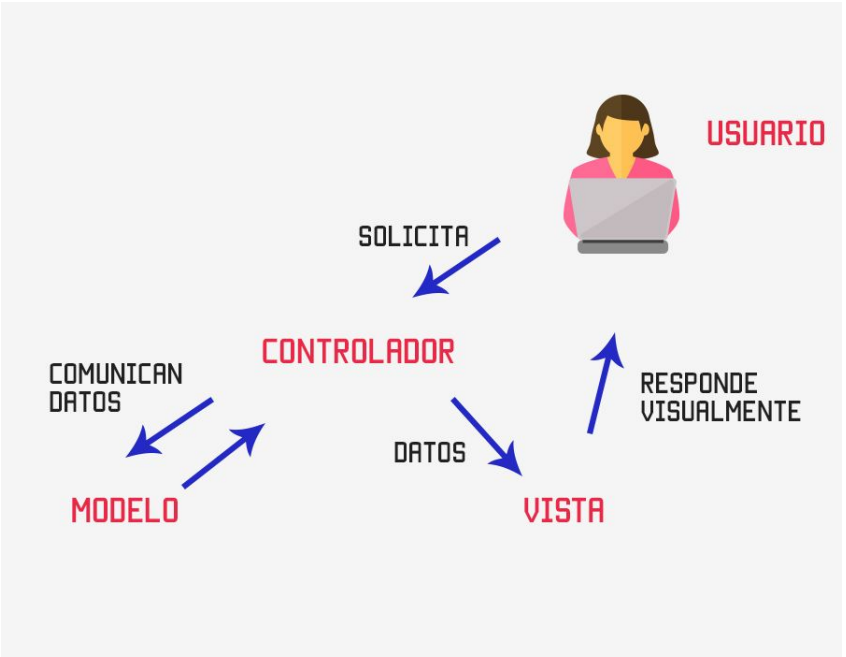
```
const miAplicacion = Vue.createApp({
   methods: {
       addText(){
           const texto = this.$refs.texto.value
           const textField = this.$refs.textField
           textField.innerHTML = textField.innerHTML + "<br>" + texto
       },
       delText() {
           const textField = this.$refs.textField
           textField.innerHTML =""
       }
   }
}).mount("#app")
```



Los métodos **addText()** y **delText()** de la instancia de Vue se encargan de responder al click de los botones. Y con innerHTML cambiamos el contenido del párrafo.

Patrón MVC | Modelo-Vista-Controlador

El patrón **MVC** (Model-View-Controller) es un patrón de arquitectura de software. Busca desacoplar al máximo la interfaz de usuario de la lógica de la aplicación, separando los componentes de una aplicación en tres grupos (o capas) principales: el **modelo**, la **vista**, y el **controlador**, y describe cómo se relacionarán entre ellos para mantener una estructura organizada, limpia y con un acoplamiento mínimo entre las distintas capas.



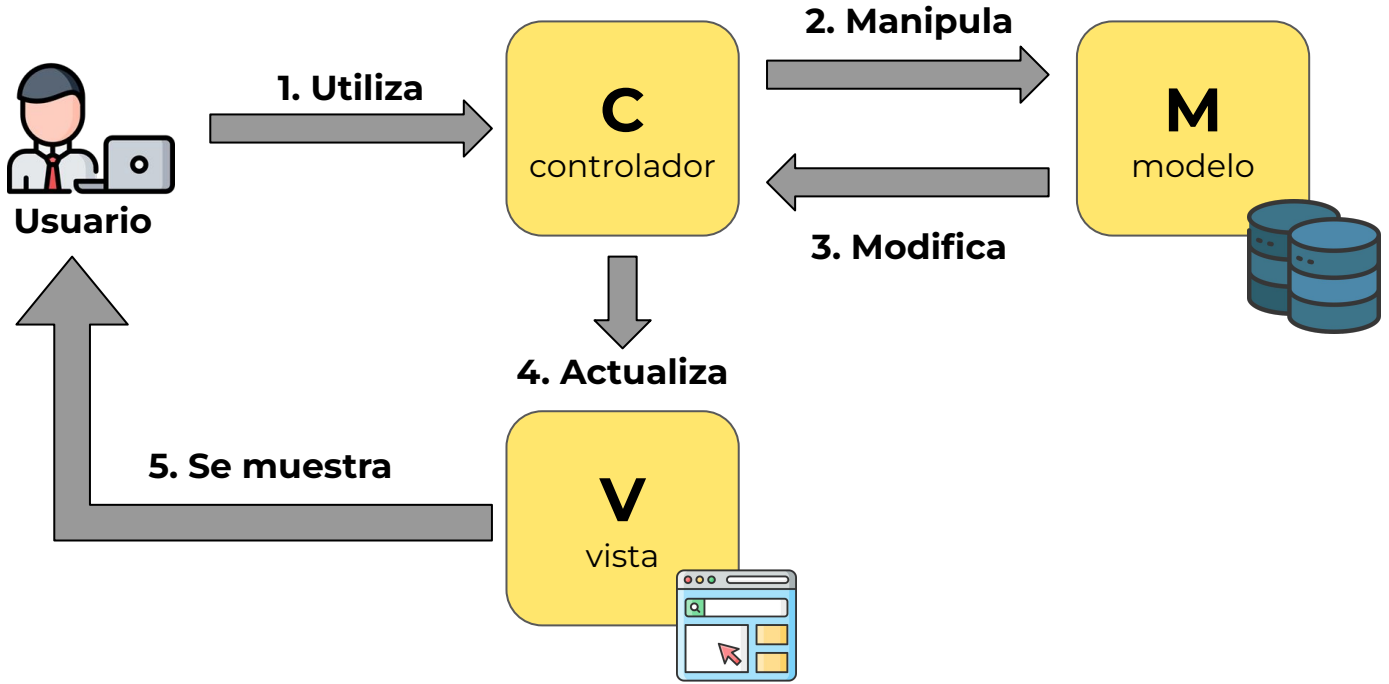
Patrón MVC

Vista: son los elementos visuales que contienen la información. Los componentes de la Vista son los responsables de generar la interfaz de nuestra aplicación. Es una representación del estado del Modelo en un momento concreto.

Controlador: intermediario entre el modelo y la vista, entre el usuario y el sistema. Capturará las acciones del usuario sobre la Vista, las interpretará y actuará en consecuencia (por ejemplo, cambiando la vista o invocando al Modelo para actualizar información). Es como un coordinador general del sistema, que regula la navegación y el flujo de información con el usuario.

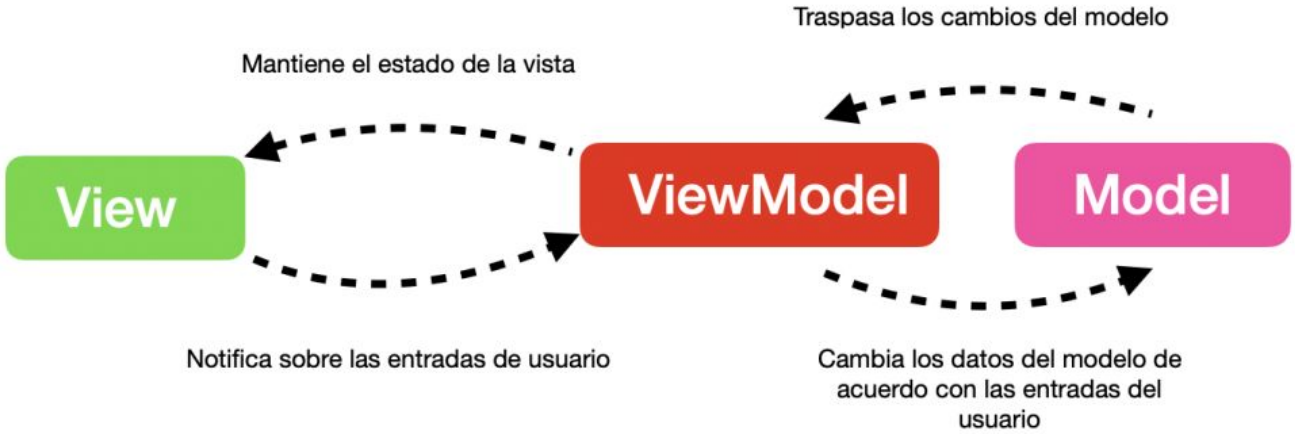
Modelo: aquí encontraremos la representación de los datos del dominio, es decir, aquellas entidades que nos servirán para almacenar información del sistema que estamos desarrollando. Por ejemplo, si estamos desarrollando una aplicación de facturación, en el modelo existirán las clases Factura, Cliente o Proveedor, entre otras. También encontraremos la lógica de negocio, es decir, la implementación de las reglas, acciones y restricciones que nos permiten gestionar las entidades del dominio, y los mecanismos de persistencia de nuestro sistema.

Patrón MVC | Modelo-Vista-Controlador



MVC y MVVM

De manera similar al MVC, el **MVVM** o **modelo–vista–modelo de vista** (model – view – viewmodel) es un patrón de arquitectura de software que se caracteriza por desacoplar lo máximo posible la interfaz de usuario de la lógica de la aplicación.



MVC y MVVM

Cada capa tiene una función específica:

- El **modelo** representa la capa de datos y/o la lógica de la aplicación. Contiene información, pero no interactúa con la vista.
- La **vista** representa la información a través de los elementos visuales que la componen. Las vistas en MVVM son activas, contienen comportamientos, eventos y enlaces a datos.
- El **modelo de vista** es un actor intermediario entre el modelo y la vista, contiene toda la lógica de presentación y se comporta como una abstracción de la interfaz. La comunicación entre la vista y el modelo de vista se realiza por medio de los enlaces de datos.

Diferencia entre MVVM y MVC

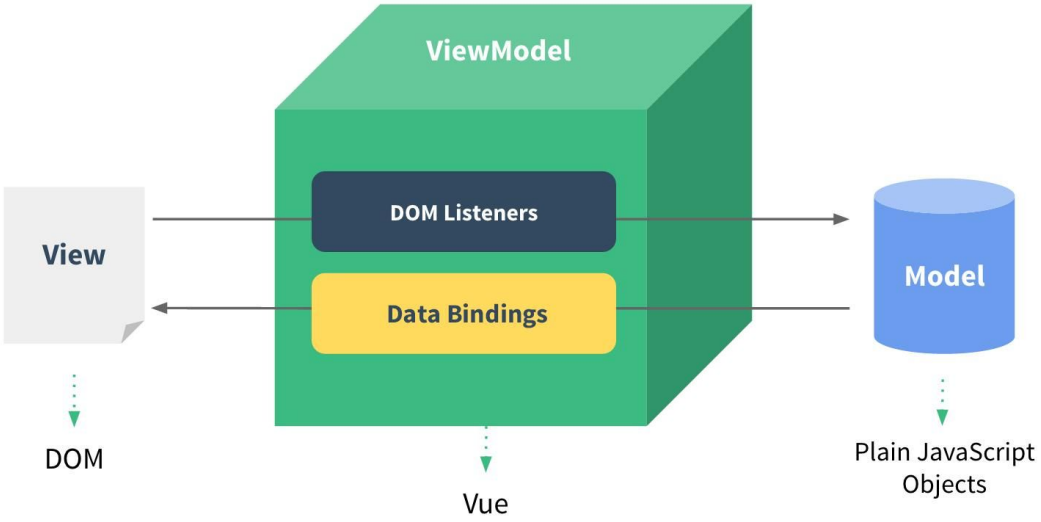
MVVM realiza la sincronización automática de la vista y el modelo, es decir, cuando cambia el atributo del modelo, no necesitamos operar manualmente el elemento dom para cambiar la visualización de la vista, la capa de la vista cambiará automáticamente.

Puntos clave del patrón MVVM:

1) El usuario interactúa con la Vista.
2) View y ViewModel tienen una relación de uno a uno, lo que significa que una Vista solo se puede asignar a un ViewModel.
3) View contiene una referencia a ViewModel, pero View no tiene información sobre el modelo.
4) Existe una relación de enlace de datos bidireccional entre View y ViewModel.

Vue y MVVM

Vue es un framework que usa la arquitectura MVVM.



Material extra

Artículos de interés

Videos:

- Fundamentos de Vue.js 3
- Curso Vue.js 3 desde cero
- Reactividad en 2 sentidos

No te olvides de dar el presente

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios obligatorios.

Todo en el Aula Virtual.

Muchas gracias por tu atención.

Nos vemos pronto