ÚLTIMA ACTUALIZACIÓN DEL DOCUMENTO: 25/1/2025

UNIVERSIDAD DE ALICANTE | SERVICIO DE INFORMÁTICA

INTRODUCCIÓN

VUE 3 Y TYPESCRIPT

Servicio de informática | Universidad de Alicante
03690 | Campus de Sant Vicent del Raspeig | España
https://si.ua.es/

CONTENIDO

CONTROL DEL DOCUMENTO

INFORMACIÓN GENERAL

Título	Introducción / Manual básico de Vue 3
Creado por	Andrés Vallés
Revisado por	
Lista de distribución	
Nombre fichero	

HISTÓRICO DE REVISIONES

Versión	Fecha	Autor	Observaciones
001	22/01/2025	Andrés Vallés	Manual básico para comenzar o para repasar Vue

INTRODUCCIÓN

Vue 3 es el lenguaje de programación escogido en el Servicio de Informática para el desarrollo, de la parte cliente, de aplicaciones web. Algunas de las razones de la selección son:

1. **Curva de aprendizaje**: La curva de aprendizaje de Vue 3 es bastante suave, especialmente para desarrolladores que ya tienen conocimientos de HTML, CSS y JavaScript. Es la principal razón de no usar otros Frameworks como Angular o React.

2. **Integración con plantillas HTML**: Vue 3 permite una integración sencilla con plantillas HTML, lo que facilita el uso de componentes reutilizables y la organización del código.

3. **Estructura de script, template y estilos**: Vue 3 utiliza una estructura de componentes que separa el código en tres secciones: <script>, <template> y <style>. Esto mejora la legibilidad y el mantenimiento del código.

4. **Reactividad**. Es una de sus características más poderosas y atractivas. Permite que los datos y el estado de la aplicación se actualicen automáticamente en la interfaz de usuario cuando cambian.

5. **Rendimiento**: Vue 3 ha mejorado significativamente en términos de rendimiento en comparación con versiones anteriores. Si además hacemos el uso de Composition API, las aplicaciones son más reutilizables y escalables.

6. **Integración y depuración en navegadores**: Vue 3 ofrece herramientas de desarrollo robustas, como Vue Devtools, que facilitan la depuración y el monitoreo de aplicaciones directamente en el navegador. Esto permite a los desarrolladores identificar y solucionar problemas rápidamente.

En resumen, Vue 3 combina facilidad de uso, rendimiento y herramientas de desarrollo avanzadas, lo que lo convierte en una opción ideal para el desarrollo de aplicaciones de la UA.

El objetivo de este documento es perder el miedo a usar esta tecnología, que simplifica enormemente trabajar con la parte cliente de nuestra aplicación.

Se ha simplificado al máximo, para que se entienda cada uno de los conceptos. En muchos apartados encontrarás enlaces a otros documentos, que complementan lo visto.

Mi recomendación es que primero leas todo el documento. Esto te va a permitir crear tus primeras aplicaciones y ver la simplicidad de su integración y uso.

En el momento que quieras hacer cosas más complejas, necesitarás acceder a los enlaces con la información ampliada.

ESTRUCTURA

La estructura básica se reduce a 3 elementos

```
<script setup lang="ts">
</script>

<template>
</template>

<style  type="scss"  scoped>
</style>
```

1. **Script** 1 <script setup lang="ts">):
    - Utiliza TypeScript (lang="ts") para añadir tipado estático.
    - Utilizamos Composition API, que es una de las mejoras implementadas en Vue 3.
2. **Template** (<template>):
    - Define la estructura visual del componente con HTML.
    - En muchas ocasiones haré referencia a “vista” cuando quiera indicar la sección template, ya que es lo que visualiza el usuario.
3. **Style** (<style type="scss" scoped>):
    - Define los estilos CSS que se aplican solo a este componente (scoped).
    - Indicamos que vamos a trabajar con SASS, que permite anidar selectores y usar variables, lo que hace que los estilos sean más organizados y fáciles de mantener

Esta estructura permite aprovechar las ventajas de TypeScript, como el tipado estático y la detección temprana de errores, junto con la API de composición de Vue 3 para una mejor organización y reutilización del código.

EJEMPLO BÁSICO

```
<script setup lang="ts">
  import { ref } from 'vue'

  const nombre = ref<string>('Mi nombre')
</script>

<template>
  <h1>Mi primera página</h1>
```

```
    <p>
      Bienvenido {{ nombre }}
    </p>
  </template>

  <style lang="scss" scoped>
    h1 {
      color: blue;
    }
  </style>
```

Script

```
  <script setup lang="ts">
    import { ref } from 'vue'

    const nombre = ref<string>('Mi nombre')
  </script>
```

- **<script setup lang="ts">**: Indica que estamos usando la API de composición (setup) con TypeScript (lang="ts").
- **import { ref } from 'vue'**: Importa la función ref de Vue, que se utiliza para crear referencias reactivas.
- **const nombre = ref<string>('Mi nombre')**: Crea una referencia reactiva llamada nombre con un valor inicial de 'Mi nombre' y especifica que es de tipo string.

Template

```
  <template>
    <h1>Mi primera página</h1>
    <p>
      Bienvenido {{ nombre }}
    </p>
  </template>
```

- **<template>**: Define la estructura HTML del componente.
- **<h1>Mi primera página</h1>**: Un encabezado simple.
- ** Bienvenido {{ nombre }} **: Un párrafo que muestra el valor de la referencia nombre usando interpolación de Vue ({{ }}).

Style

```
  <style lang="scss" scoped>
    h1 {
      color: blue;
    }
  </style>
```

- **<style lang="scss" scoped>**: Indica que los estilos están escritos en SCSS (lang="scss") y que son scoped, es decir, solo se aplican a este componente.
- **h1 { color: blue; }**: Aplica el color azul al texto del encabezado <h1>.

El resultado es:

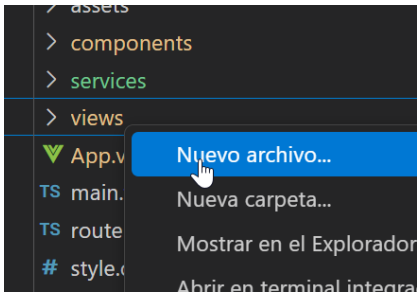
Mi primera página

Bienvenido Mi nombre

CREAR MI PRIMERA VISTA / PÁGINA

Cada pantalla o páginas, se llama vista en Vue. Por unificar las aplicaciones, se dejan en la carpeta views (en src) y el nombre de la vista comienza por mayúscula (PascalCase) y la extensión debe ser .vue.

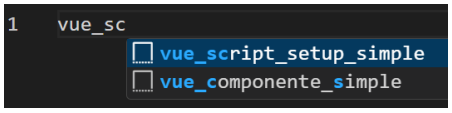
En Visual Studio Code con el botón derecho sobre views, seleccionamos un Nuevo archivo …



Un nombre correcto sería Home.vue

Tenemos un atajo para crear la estructura básica, vue_script_setup_simple.

Simplemente comenzamos a escribir vue_ y veremos las opciones. Seleccionamos vue_script_setup_simple



Y genera

```
1  <script setup lang="ts">
2
3  </script>
4
5  <template>
6
7  </template>
8
9  <style scoped lang="scss">
10
```

```
11  </style>
```

Para aprender como asociar una dirección / ruta a una vista consulta Aplicaciones Vue 3 y TypeScript - Enrutamiento.pdf

VARIABLES

En TypeScript, las variables y constantes se declaran de manera similar a JavaScript, pero con la opción de añadir tipos para mayor seguridad y claridad.

Variables: Se declaran usando let o var. Debes especificar el tipo de la variable después del nombre, poniendo : y el tipo.

```
1  let edad: number = 30;
2  let nombre: string = 'Sergio';
```

Constantes: Se declaran usando const y, al igual que las variables, puedes especificar el tipo. Las constantes no pueden ser reasignadas. El nombre se recomienda que se ponga en mayúsculas para diferenciarlas de las variables

```
1  const PI: number = 3.14;
2  const SALUDO: string = 'Hola';
```

Enum: Es una forma de definir un conjunto de valores con nombre, lo que puede hacer que el código sea más legible y fácil de mantener.

```
1  enum Color {
2    Rojo = 'Rojo',
3    Verde = 'Verde',
4    Azul = 'Azul'
5  }
6
7  let colorFavorito: Color = Color.Rojo;
```

- enum Color: Define un enumerado llamado Color con tres valores: Rojo, Verde y Azul.
- colorFavorito: Color: Declara una variable colorFavorito de tipo Color y le asigna el valor Color.Rojo.

Tipos

El uso de tipos en TypeScript ayuda a detectar errores en tiempo de desarrollo y mejora la documentación del código, haciendo que sea más fácil de entender y mantener

- **Primitivos**: number, string, boolean, null, undefined, symbol, bigint.

  let nombre: string = 'Paco'

- **Arrays**: Puedes definir arrays de un tipo específico.

  let numeros: number[] = [1, 2, 3];

- **Objetos**: Puedes definir la estructura de un objeto.

  let persona: { nombre: string; edad: number } = { nombre: 'Paco', edad: 35 };

  Si quieres profundizar en como normalizar objetos con interfaces, hay un documento específico Aplicaciones Vue 3 y TypeScript - Buenas prácticas.pdf

REF Y REACTIVE

En Vue 3, las variables reactivas se manejan principalmente mediante las funciones ref y reactive.

ref: Se utiliza para crear una referencia reactiva a un valor primitivo (como un número, cadena o booleano). Cuando el valor cambia, Vue actualiza automáticamente la vista.

```
1  import { ref } from 'vue';
2  const contador = ref(0);
3  contador.value++; // Actualiza la vista
```

reactive: Se utiliza para crear un objeto reactivo, ideal para estructuras de datos más complejas como objetos y arrays. Los cambios en las propiedades del objeto también actualizan la vista.

```
1  import { reactive } from 'vue';
2  const usuario = reactive({ nombre: Sergio, edad: 30 });
3  usuario.edad++; // Actualiza la vista
```

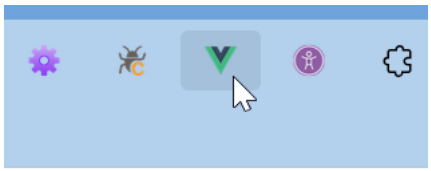
La reactividad en Vue 3 permite que los cambios en los datos se reflejen automáticamente en la pantalla del usuario, simplificando la gestión del estado y mejorando la eficiencia del desarrollo.

Si quieres profundizar en como normalizar objetos con interfaces, hay un documento específico Aplicaciones Vue 3 y TypeScript - Buenas prácticas.pdf

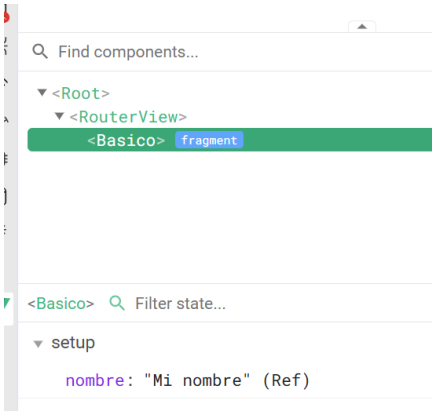
DEPURACIÓN

Disponemos de una magnífica herramienta para depurar nuestras aplicaciones Vue.js, Vue DevTools, https://devtools.vuejs.org/, disponible como extensión para Firefox, Chrome y Edge.

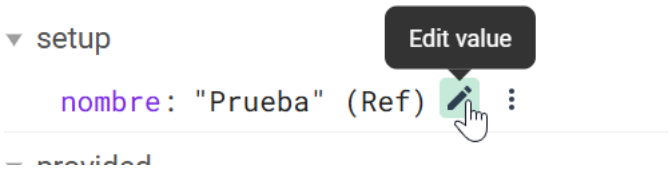
Una vez instalada, accedemos con este icono, en la barra superior o en las herramientas de desarrollo



Seleccionando nuestra vista o componente accedemos a todas las variables



Podemos modificar los valores y, al ser elementos reactivos, se reflejan en la página web.



Mi primera página

Bienvenido Prueba

FORMULARIOS

Trabajar con formularios es muy sencillo, definimos una variable para cada campo del formulario.

```
<script setup lang="ts">
  import { ref } from 'vue'

  const nombre = ref('');
  const apellidos = ref('');
  const email = ref('');
</script>
```

Luego lo integramos en el código. Antes hemos visto que para visualizar una variable en la sección templates usábamos {{ nombrevariable }}.

Para poder asignar variables a campos de un formulario, y que los cambios se reflejen en éstas, usamos v-model.

```
  <form class="needs-validation" novalidate>
    <div class="mb-3">
      <label for="nombre" class="form-label">Nombre</label>
      <input type="text" id="nombre" class="form-control" v-model="nombre" required>
      <div class="invalid-feedback">
        Por favor, ingresa tu nombre.
      </div>
    </div>
    <div class="mb-3">
      <label for="apellidos" class="form-label">Apellidos</label>
      <input type="text" id="apellidos" class="form-control" v-model="apellidos" required>
      <div class="invalid-feedback">
        Por favor, ingresa tus apellidos.
      </div>
    </div>
    <div class="mb-3">
      <label for="email" class="form-label">Correo Electrónico</label>
      <input type="email" id="email" class="form-control" v-model="email" required>
      <div class="invalid-feedback">
        Por favor, ingresa un correo electrónico válido.
      </div>
    </div>
    <button type="submit" class="btn btn-primary">Enviar</button>
  </form>
```

Cuando sean muchos campos, en vez de crear muchas variables creamos un objeto, con muchas propiedades.

```
<script setup lang="ts">
  import { reactive } from 'vue'

  const persona = reactive({
    nombre: '',
    apellidos: '',
    email: ''
  });

</script>
```

Y adaptamos los v-model a esta estructura, persona.propiedad

```
<template>
  <form class="needs-validation" novalidate>
    <div class="mb-3">
      <label for="nombre" class="form-label">Nombre</label>
      <input type="text" id="nombre" class="form-control" v-model="persona.nombre" required>
      <div class="invalid-feedback">
        Por favor, ingresa tu nombre.
      </div>
    </div>
    <div class="mb-3">
      <label for="apellidos" class="form-label">Apellidos</label>
      <input type="text" id="apellidos" class="form-control" v-model="persona.apellidos" required>
      <div class="invalid-feedback">
        Por favor, ingresa tus apellidos.
      </div>
    </div>
    <div class="mb-3">
      <label for="email" class="form-label">Correo Electrónico</label>
      <input type="email" id="email" class="form-control" v-model="persona.email" required>
      <div class="invalid-feedback">
        Por favor, ingresa un correo electrónico válido.
      </div>
    </div>
    <button type="submit" class="btn btn-primary">Enviar</button>
  </form>
</template>
```

Para aprender como trabajar con otros elementos de formulario ,como checkbox, radios o desplegables, consulta el manual oficial Form Input Bindings | Vue.js.

Si quieres profundizar en como validar los campos de un formulario, hay un documento específico Aplicaciones Vue 3 y TypeScript - Gestión formularios.pdf

FUNCIONES

Las funciones son muy parecidas a como las hemos hecho en JavaScript, pero debemos definir los tipos de los parámetros y de la respuesta.

```
function nombreCompleto (nombre: string, apellidos: string): string {
  return `${nombre} ${apellidos}`;
}
```

Cada vez se está normalizando la definición de funciones con esta forma

Si es una sola línea, que devuelve lo que hay en ella (no hace falta el return)

```
1  const nombreCompleto = (nombre: string, apellidos: string): string => `${nombre}
   ${apellidos}`;
```

O con varias líneas (requiero el return)

```
1  const nombreCompleto = (nombre: string, apellidos: string): string => {
2    return `${nombre} ${apellidos}`;
3  };
```

La razón de uso es la mejora de rendimiento en el procesado y ejecución.

Su uso, en todos los casos, se hace igual

```
1  let nombre = "Juan";
2  let apellidos = "Pérez";
3  const nombreCompleto = nombreCompleto (nombre, apellidos);
```

CON ACCIONES

Lo más común es realizar llamadas a funciones antes acciones del usuario, por ejemplo, al pulsar un botón, al validar un formulario, al cambiar un campo, etc.

El pulsar un botón, es tan simple como capturar el clic en éste.

```
1  <button @click="saludar">Saludar</button>
```

Las acciones se integran en el template, dentro del HTML con @evento o v-on:evento

Para profundizar en el uso de acciones, consulta el manual oficial Event Handling | Vue.js.

COMPUTED PROPERTIES

Son una forma de declarar propiedades que dependen de otras propiedades reactivas. Se utilizan para realizar cálculos derivados de los datos de la vista o el componente y se recalculan automáticamente cuando las dependencias cambian.

Partiendo que tenemos dos variables nombre y apellidos, el nombre completo es una concatenación de las dos

```
1  const nombre = ref('Félix');
2  const apellidos = ref('Blanes');
3
4  const nombreCompleto = computed<string>(() => {
5    return `${nombre.value} ${apellidos.value}`;
6  });
```

A diferencia de la función, no le paso los parámetros (por eso ponemos ()), porque se calcula con otras variables. Todo es reactivo y un cambio en una de las variables, cambia el nombreCompleto.

Aparentemente nombreCompleto es una variable más.

Para profundiza en el uso de computed properties, consulta el manual oficial Computed Properties | Vue.js

WATCH

Son funciones reactivas que permiten observar y reaccionar a los cambios en datos específicos.

Son útiles para ejecutar lógica adicional cuando cambian ciertos valores, como realizar llamadas a APIs, validar datos, o actualizar otros estados

```
1  const nombre = ref('');
2
3  // Observamos cambios en 'nombre'
4  watch(nombre, (nuevoNombre) => {
5    console.log(`Nombre cambiado a: ${nuevoNombre}`);
6  });
```

Si son varios campos, los metemos en un array

```
1  const nombre = ref('');
2  const apellidos = ref('');
3
4  // Observamos cambios en 'nombre' y 'apellidos'
5  watch([nombre, apellidos], ([nuevoNombre, nuevosApellidos]) => {
6    console.log(`Nombre cambiado a: ${nuevoNombre}, Apellidos cambiados a:
   ${nuevosApellidos}`);
7  });
```

Si es una variable reactive

```
1    const persona = reactive({
2      nombre: '',
3      apellidos: '',
4      email: ''
5    });
6
7    watch(persona, (nuevapersona) => {
8      console.log(`Nombre cambiado a: ${nuevapersona.nombre}, Apellidos cambiados a:
     ${nuevapersona.apellidos}`);
9    });
```

Para profundiza en el uso de los watches, consulta el manual oficial Watchers | Vue.js

EVENTOS ENTRE COMPONENTES

Una de las cosas más complejas a la hora de trabajar con Vue.js es el paso de información entre padres e hijos, entre hijos a padres y entre cualquier componente, independiente del nivel de parentesco.

Vamos a verlo con ejemplos para simplificarlo.

EN EL PROPIO .VUE

Como hemos comentado antes, lo hacemos @ o v-on: y la acción, por ejemplo, click, keyup, submit, etc.

Es el más básico, pero que lo usaremos con frecuencia.

Hay una serie de modificadores, asociados al evento, que da más posibilidades Event Handling | Vue.js (parar propagación, que se ejecute solo una vez, detectar tecla pulsada, etc.).

ENTRE PADRE E HIJO

Los **props** se utilizan para pasar datos de un componente padre a un componente hijo.

Para ello el hijo indica que propiedades puede recibir

```
1  // Componente Hijo
2  <script setup lang="ts">
3  import { defineProps } from 'vue';
4
5  const props = defineProps<{ mensaje: string }>();
6  </script>
7
```

```
<template>
  <div>
    <p>{{ mensaje }}</p>
  </div>
</template>
```

Y el padre lo llama con :prop=”valor”

```
// Componente Padre
<script setup lang="ts">
import ChildComponent from './ChildComponent.vue';
import { ref } from 'vue';

const mensajePadre = ref('Hola desde el componente padre');
</script>

<template>
  <ChildComponent :mensaje="mensajePadre" />
</template>
```

Para profundiza en el uso de props , consulta el manual oficial [Props | Vue.js]

ENTRE HIJO Y PADRE

Los emits se utilizan para enviar eventos desde un componente hijo a un componente padre.

```
// Componente Hijo
<script setup lang="ts">
import { defineEmits } from 'vue';

const emit = defineEmits(['mensajeEnviado']);

const enviarMensaje = () => {
  emit('mensajeEnviado', 'Hola desde el componente hijo');
};
</script>

<template>
  <button @click="enviarMensaje">Enviar Mensaje</button>
</template>
```

Y el padre lo recibe como si fuera otro evento (con @ on v-on:)

```
// Componente Padre
<script setup lang="ts">
import ChildComponent from './ChildComponent.vue';
```

```
4
5   const manejarMensaje = (mensaje: string) => {
6     console.log(mensaje);
7   };
8   </script>
9   <template>
10    <ChildComponent @mensajeEnviado="manejarMensaje" />
11  </template>
```

Para profundiza en el uso de emits , consulta el manual oficial Component Events | Vue.js

ENTRE UN PADRE Y CUALQUIER COMPONENTE HIJO, NIETO, ETC.

provide y inject permiten compartir datos entre componentes, relacionado jerárquicamente, pero sin tener que ser directa.

provide

El componente padre utiliza provide para ofrecer datos a sus descendientes.

```
1   // Componente Padre
2   <script setup lang="ts">
3   import { provide } from 'vue';
4
5   const mensaje = 'Hola desde el componente padre';
6   provide('mensaje', mensaje);
7   </script>
8   <template>
9     <ChildComponent />
10  </template>
```

inject

El componente hijo utiliza inject para recibir los datos proporcionados por un ancestro.

```
1   // Componente nieto
2   <script setup lang="ts">
3   import { inject } from 'vue';
4   const mensaje = inject<string>('mensaje');
5   </script>
6
7   <template>
8     <p>{{ mensaje }}</p>
9   </template>
```

Para profundiza en el uso de provide y inject , consulta el manual oficial Provide / Inject | Vue.js

ENTRE CUALQUIER COMPONENTE

En ocasiones nos interesa comunicar componentes que no contienen relación jerárquica o la desconocemos.

En este caso debemos hacer uso de una librería externa mitt, que genera un bus de eventos al que cualquier componente se puede conectar para enviar o recibir.

Los pasos son los siguientes:

- Crear un bus específico o general para toda la aplicación
- Emitir eventos al bus
- Quedar a la espera de eventos del bus

Crear bus

Nos creamos un fichero TypeScript en la carpeta servicios. En este caso lo he llamado useEventBus.js y simplemente inicializa un bus mitt para poder hacer la comunicación.

```
1  import mitt from 'mitt';
2
3  const emitter = mitt();
4
5  export default function useEventBus() {
6      return emitter
7  }
```

Emitir evento

Para enviar un evento debemos importar este bus creado anteriormente y hacer uso de la función emit, que tiene dos campos, clave o identificador del mensaje, y el propio mensaje. Puede ser cualquier elemento, una cadena de texto, un objeto, …

```
1  import useEventBus from "../../servicios/useEventBus"
2
3  const { emit } = useEventBus()
4
5  const mensajeTodos = () => {
6      emit('mensaje', 'Bienvenidos al curso de Vue 3')
7  }
```

Recepción evento

Es algo parecido a lo anterior, pero esta vez quedamos a la espera con el evento on, que lo asociamos a una función nuestra

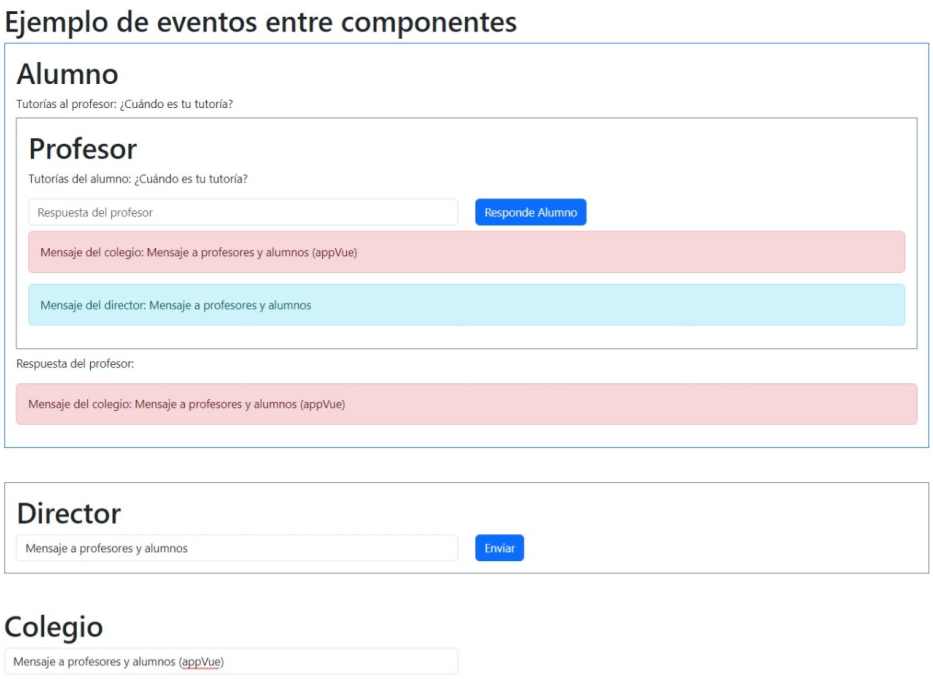
```
1  import useEventBus from "../../servicios/useEventBus"
2
3  const { on } = useEventBus()
4
5  on("mensaje", (mensajedirector: string) => {
6      console.log("mensaje", mensajedirector);
7  })
```

EJEMPLO COMPLETO

Vamos a trabajar con 3 componentes, director, profesor y alumno. Además, usamos la aplicación como si fuera el colegio y es el que manda mensaje a todos los componentes.

- El profesor y el alumno, al ser padre e hijo, se comunican con prop y emit
- El colegio, al ser el padre de todos los componentes, hace uso de provide. Los componentes hacen uso de inject para recibirlo
- Por último, el director y el profesor no tienen dependencia jerárquica y hacen uso de la librería Mitt

El resultado final es el siguiente:



Eventos.Vue (vista)

Al asignar en provide una clase reactiva, cualquier modificación de ella, se verá reflejada en el inject que lo recibe.

```
1  <script setup lang="ts">
2  import { provide, ref } from "vue";
3
4  import Alumno from '@/components/eventos/Alumno.vue';
5  import Director from '@/components/eventos/Director.vue';
6
7  const mensajeColegio = ref('Mensaje a profesores y alumnos (appVue)');
8  provide('key', mensajeColegio)
9  </script>
10
11 <template>
12    <h1>Ejemplo de eventos entre componentes</h1>
13
14    <Alumno />
15
16    <Director class="mt-5" />
17
18    <h2 class="mt-5">Colegio</h2>
19
20    <div class="row g-3 align-items-center">
21       <div class="col-2">
22          <label for="mensajecolegio" class="col-form-label">Mensaje del colegio</label>
23       </div>
24       <div class="col-7">
25          <input type="text" id="mensajecolegio" class="form-control" aria-
   describedby="descripcionmnensajecolegio" v-model="mensajeColegio">
26       </div>
27       <div class="col-3">
28          <span id="descripcionmnensajecolegio" class="form-text">
29             Texto que se envía a profsores y alumnos
30          </span>
31       </div>
32    </div>
33
34 </template>
35
36 <style scoped lang="scss"></style>
```

Director.vue

El director es un componente sin jerarquía sobre profesores o alumnos. Usamos el eventbus.

```
1  <script setup lang="ts">
2  import { provide, ref } from "vue";
3
4  import useEventBus from "../../servicios/useEventBus";
5  const { emit } = useEventBus();
6
7  const mensajeDirector = ref('Mensaje a profesores y alumnos');
```

```
8
9   const mensajeTodos = () => {
10     emit('mensaje', mensajeDirector.value)
11     console.log("mensajeDirector", mensajeDirector.value)
12   }
13
14  </script>
15
16  <template>
17     <div class="border border-secondary p-3">
18        <h2>Director</h2>
19
20        <div class="row g-3 align-items-center">
21           <div class="col-auto">
22              <label for="tutoriaalumno" class="col-form-label">Mensaje a los profesores</label>
23           </div>
24           <div class="col-auto">
25              <input type="text" v-model="mensajeDirector" id="tutoriaalumno" class="form-
    control" aria-describedby="descripciontutoriaalumno">
26           </div>
27           <div class="col-auto">
28              <span id="descripciontutoriaalumno" class="form-text">
29                 Texto a los profesores
30              </span>
31           </div>
32           <div class="col-auto">
33              <button class="btn btn-primary" @click="mensajeTodos">Enviar</button>
34           </div>
35        </div>
36
37     </div>
38
39  </template>
40  <style scoped lang="scss"></style>
```

Profesor.vue

El profesor puede recibir mensaje del colegio (superior jerárquico), de un alumno (su hijo) o del director (sin relación jerárquica)

Del colegio lo recibimos por inject (lo ha mandado por provide). Al ser reactiva la variable que nos pasa, los cambios son instantáneos

Del director lo recibimos por el eventbus. En este caso lo gestionamos con la función on.

Por último, el alumno nos envía los datos con props y le contestamos con emits, que gestionará el alumno.

```
1   <script setup lang="ts">
2   import { inject, ref, onMounted } from "vue";
3   import useEventBus from "../../servicios/useEventBus";
4
5   const { on } = useEventBus();
6
```

```
7  const props = defineProps<{ tutorias: string }>();
8  const respuestaAlumno = ref('');
9
10 const mensajeColegio = inject<string>('key');
11 const mensajeDirector = ref('');
12
13 const emit = defineEmits(['responderAlumno']);
14
15 on("mensaje", (mensajedirector: string) => {
16     mensajeDirector.value = mensajedirector;
17 });
18
19 const responderAlumno = () => {
20     emit('responderAlumno', respuestaAlumno);
21 };
22
23 </script>
24
25 <template>
26     <div class="border border-secondary p-3">
27         <h2>Profesor</h2>
28
29         <p>
30             Tutorías del alumno: {{ props.tutorias }}
31         </p>
32
33         <div class="row">
34             <div class="row g-3 align-items-center">
35                 <div class="col-auto">
36                     <label for="respuestaprofesor" class="col-form-label">Respuesta
   profesor</label>
37                 </div>
38                 <div class="col-auto">
39                     <input type="text" id="respuestaprofesor" v-model="respuestaAlumno"
   class="form-control"
40                         aria-describedby="descripcionrespuestaprofesor">
41                 </div>
42                 <div class="col-auto">
43                     <span id="descripcionrespuestaprofesor" class="form-text">
44                         Respuesta del profesor a la tutoría del alumno.
45                     </span>
46                 </div>
47                 <div class="col-auto">
48                     <button class="btn btn-primary" @click="responderAlumno">Responde
   Alumno</button>
49                 </div>
50             </div>
51         </div>
52
53         <div class="alert alert-danger mt-2" role="alert" v-if="mensajeColegio">
54             Mensaje del colegio: {{ mensajeColegio }}
55         </div>
56
57         <div class="alert alert-info mt-2" role="alert" v-if="mensajeDirector">
58             Mensaje del director: {{ mensajeDirector }}
59         </div>
60
61     </div>
62
```

```
63  </template>
64
65  <style scoped lang="scss"></style>
```

Alumno.vue

El alumno puede recibir mensaje del colegio (superior jerárquico) o de su profesor (su padre).

Del colegio lo recibimos por inject (lo ha mandado por provide). Al ser reactiva la variable que nos pasa, los cambios son instantáneos

Por último, el profesor nos contesta con emits, lo gestionamos como un evento más y el alumno le envía los datos con props.

```
66  <script setup lang="ts">
67
68  import { ref,inject } from 'vue'
69  import Profesor from './Profesor.vue';
70
71  const tutoriaProfesor = ref('¿Cuándo es tu tutoría?');
72  const respuestaProfesor = ref('');
73  const mensajeColegio = inject<string>('key');
74
75  const mostrarRespuesta = (respuesta: string) => {
76      respuestaProfesor.value = respuesta;
77  };
78  </script>
79
80  <template>
81      <div class="border border-primary p-3">
82          <h1>Alumno</h1>
83
84          <p class="mb-2">
85              Tutorías al profesor: {{ tutoriaProfesor }}
86          </p>
87
88          <Profesor :tutorias="tutoriaProfesor" @responderAlumno="mostrarRespuesta" />
89
90          <p class="mt-2">
91              Respuesta del profesor: {{ respuestaProfesor }}
92          </p>
93
94          <div class="alert alert-danger mt-2" role="alert" v-if="mensajeColegio">
95              Mensaje del colegio: {{ mensajeColegio }}
96          </div>
97      </div>
98  </template>
99
100 <style scoped lang="scss"></style>
```

COMPONENTES

Los componentes se han simplificado mucho en Vue 3 permitiendo agrupar la lógica relacionada en funciones reutilizables.

Tiene la misma estructura que una vista, lo que facilita enormemente su uso.

Si quieres profundizar en cómo crear componentes, hay un documento específico, Aplicaciones Vue 3 y TypeScript - Componentes.pdf

ESTRUCTURA

La misma que una vista.

```
1  <script setup lang="ts">
2  </script>
3
4  <template>
5  </template>
6
7  <style  type="scss"  scoped>
8  </style>
```

Luego para incluirlo en una vista o en otro componente, solo debemos importar el componente. Tenemos que incluir la extensión .vue en el nombre.

```
1  import Alumno from '@/components/componentes/Alumno.vue';
```

y usarlo en la sección template

```
1  <Alumno />
```

SLOTS

Son una característica que permite a los componentes aceptar contenido dinámico de sus padres. Son útiles para crear componentes más flexibles y reutilizables.

```
1  <script setup lang="ts">
2  </script>
3
4  <template>
5      <div class="border border-primary p-3">
6          <h1>Alumno</h1>
```

```
7
8         <div class="alert alert-info" role="alert">
9           <slot name="aviso">
10            <p>El alumno pregunta al profesor por la tutoría</p>
11          </slot>
12        </div>
13
14        <p>
15        <slot name="contenido">
16          Aquí explicamos cómo se pregunta al profesor por la tutoría
17        </slot>
18        </p>
19    </div>
20  </template>
21
22  <style scoped lang="scss"></style>
```

Luego el que lo llama debe indicar en que slot quiere modificar el contenido por defecto y poner uno nuevo.

```
23  <script setup lang="ts">
24    import Alumno from '@/components/componentes/Alumno.vue';
25  </script>
26
27  <template>
28    <h1>Ejemplo de componentes</h1>
29
30    <Alumno>
31      <template #aviso>
32        <p>El alumno pregunta al profesor por la tutoría (modificado por el slot)</p>
33      </template>
34    </Alumno>
35
36  </template>
37
38  <style scoped lang="scss">
39  </style>
```

Para profundiza en el uso de los slots, consulta el manual oficial [Slots | Vue.js]

V-MODEL

Al igual que vimos al trabajar con los formularios, la forma mejor de que nuestro componente gestione una propiedad, que se modifique en ambos sentidos, es con v-model.

Desde la versión 3.4 han simplificado mucho el proceso de pasar y actualizar los v-model

```
const model = defineModel()
```

Para recuperar o actualizar el modelo accedemos directamente con model.value

```
model.value = 'hola'
```

Si queremos asignar el tipo y valor por defecto

```
const model = defineModel({ type: Number, default: 0 })
```

Vue 3 se permite trabajar con varios v-model, lo que nos da muchas más posibilidades. Consulta el manual oficial [Component v-model | Vue.js]

SERVICIOS

Un servicio es un conjunto de funciones y propiedades que encapsulan lógica reutilizable y pueden ser compartidas entre diferentes componentes o vistas. Con la Composition API, los servicios se suelen definir como funciones que comienzan con el prefijo use, siguiendo la convención de nomenclatura useNombreServicio.

Si tenemos que hacer un símil con C#, serían las clases en plural.

ESTRUCTURA

Creamos una función exportable, export function useNombreServicio, que incluye funciones y propiedades.

Las propiedades y funciones que devolvamos en el return son accesibles desde el exterior. El resto son para complementar o de apoyo al servicio.

```
1   import { ref } from 'vue'
2
3   const propiedad = ref("Desconocido")
4
5   export function useServicioGenerico() {
6
7     function funcion(nuevovalor: string) {
8       propiedad.value = nuevovalor
9     }
10
11    return {
12      propiedad,
13      funcion
14    }
```

```
15  }
```

Las propiedades que definamos fuera de export function, como es el caso de “propiedad”, son compartidas para todas las vistas que lo utilicen. Un cambio desde una vista afecta al resto de vistas.

Para usarlo desde una vista debemos importarlo

```
1   import { useServicioGenerico } from ‘@/services/aprender/useServicioGenerico’
```

En el caso de los servicios, como en las intefaces, no es necesario poner la extensión .ts en el import.

Y luego usar la propiedad o el servicio

```
2   onMounted(() => {
3       useServicioGenerico().funcion("Nuevo texto")
4   })
```

```
5   <template>
6       <h1>Servicios</h1>
7
8       {{  useServicioGenerico().propiedad }}
9
10
11  </template>
```

Si queremos descomponer las propiedades y funciones, usaremos:

```
12  const { propiedad, funcion } = useServicioGenerico()
13  onMounted(() => {
14      funcion("Nuevo texto")
15  })
16  …
17  {{  propiedad }}
```

En versiones anteriores de Vue.js se perdía la reactividad al hacer esta asociación. Teníamos que hacer uso de toRef para volver a hacer la propiedad reactiva. Tenerlo en cuenta por si en algún momento detectáis que se ha perdido la reactividad.