Analizando Código Vue 3 – Viniendo de ReactJs

Composition - API

Estudiante

David Medina

Unidad De Formación

Proyecto de Software 3

Docente

Adrián Izquierdo

Instituto Universitario del putumayo

Utp – Mocoa

04 / 09 / 2025

Teniendo en cuenta que vengo trabajando con React desde hace mucho tiempo decidí hacer una pequeña comparación entre algunos elementos de ReactJs y Vue 3 que tienen en común, elementos que son la base fundamental de cada uno de ellos.

Iniciando por:

Estructura de Componentes

- **Componente Básico**
  React:

```jsx
import React, { useState, useEffect } from 'react';

function ListContacts({ title = 'lista' }) {
  const [contactList, setContactList] = useState([]);

  useEffect(() => {
    console.log(contactList);
  }, [contactList]);

  return (
    <div className="m-6">
      <h3>{title}</h3>
      {/* JSX content */}
    </div>
  );
}
```

Vue 3:

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

// Props (equivale a function props en React)
const props = defineProps({
  title: { type: String, default: 'lista' }
});

// State (equivale a useState)
const contactList = ref([]);

// Effect (equivale a useEffect)
watchEffect(() => {
  console.log(contactList.value);
});
</script>

<template>
  <div class="m-6">
    <h3>{{ title }}</h3>
    <!-- Template content -->
  </div>
</template>
```

Como podemos observar la forma en que usamos props en nuestros componentes es totalmente diferente mientras que en React las obtenemos directamente en los paréntesis de la función, en Vue 3 usamos una palabra reservada para dar a definir los props de nuestro componente. Es decir, lo que recibimos.

La forma en que definimos estados de variables es distinta. Mientras que en React usamos el useState para definir cualquier tipo de variable en Vue es distinto, ya que aca para variables primitivas es decir por

lo general variables que contienen un solo dato, cosa que si se reasigna se reasigna entera. A eso se le llaman variables primitivas y se definen de la siguiente manera:

- **Variables Primitivas**

```
const name = ref('Juan')
```

Como podemos observar solo contiene un solo dato por lo que si reasignamos en cualquier lugar del código se reasigna entera la variable de la siguiente manera:

```
const contador = ref(0)

// Necesitas .value para leer/escribir
console.log(contador.value) // 0
contador.value = 5

// En el template NO necesitas .value
// <template>{{ contador }}</template>
```

Como vemos se reasigna con un .value una sintaxis bastante parecida a cuando manipulamos el DOM de HTML desde JS.

Cabe destacar que para poder leer o Reescribir esa variable se tiene que si o si poner .Value dentro de lo que cabe de la lógica del archivo de Vue.

Ya en el Témplate para poder usarlo lo podemos usar sin él. Value

- **Para Objetos o Arrays**

De igual manera para estados de objetos o arrays se suele usar Reactive y no REF

Aunque depende de como se vaya a reasignar la variable mas adelante. Si se reasigna completa lo mejor es usar REF ya que mantendríamos la reactividad que es que REF se usa cuando se reasigna toda la variable por completo.

```
// ✅ Mejor usa ref para reasignaciones completas
const usuario = ref({ nombre: 'Juan' })
usuario.value = { nombre: 'Pedro' } // ✅ Mantiene reactividad
```

Si se piensa reasignar la variable, pero no entera, lo adecuado es usar Reactive para mantener la reactividad.

```javascript
// ❌ PIERDE REACTIVIDAD
let usuario = reactive({ nombre: 'Juan', edad: 25 })

// Esto rompe la reactividad porque reemplazas toda la referencia
usuario = { nombre: 'Pedro', edad: 30 } // 💥 Ya no es reactivo!
```

```javascript
const usuario = reactive({ nombre: 'Juan', edad: 25 })

// ✅ Mantiene reactividad - modificas propiedades
usuario.nombre = 'Pedro'
usuario.edad = 30

// ✅ O usa Object.assign para múltiples cambios
Object.assign(usuario, { nombre: 'Pedro', edad: 30 })
```

También podemos observar que el cuerpo del componente de React a diferencia del de Vue esta dentro de una función, cosa que en Vue no pasa así, aca usamos etiqueta <script> para poder definir el cuerpo de la lógica del componente y ya luego si Témplate para hacer la función de JSX de React.JS

```
<script setup>
```

Custom Hooks vs Composables

En React usamos Custom Hooks de la siguiente manera para aplicar la lógica del negocio.

```javascript
// useContacts.js
import { useState, useEffect } from 'react';
import { getAllContacts } from '../api';

export function useContacts() {
  const [contacts, setContacts] = useState([]);
  const [message, setMessage] = useState('');

  const showEmpty = (msg) => {
    if (contacts.length < 1) {
      setMessage(msg);
    } else {
      setMessage('');
    }
  };

  useEffect(() => {
    setContacts(getAllContacts());
  }, []);

  return { contacts, message, showEmpty, setContacts };
}
```

En Vue tenemos Composables que son básicamente componentes apartes que almacenan cierta lógica del negocio que se podrá reutilizar en otros componentes, tal como hacemos en React con los Custom Hooks.

```
// useListaContacs.js (tu código)
import { ref } from 'vue';
import { getAllContacts } from '../services/apiContacs';

const contactList = ref(getAllContacts()); // ⚠️ Estado global compartido

export function useListaContacs() {
  const msjEmpy = ref('');

  const showEmpy = (msj) => {
    if (contactList.value.length < 1) {
      msjEmpy.value = msj;
    } else {
      msjEmpy.value = '';
    }
  };

  return { contactList, msjEmpy, showEmpy };
}
```

Como podemos observar así lucen los Composables de Vue son similares a los Custom Hooks de React, sin embargo, hay una gran diferencia. Y es que el estado de la lista se encuentra global en el caso de Vue, ya que se encuentra afuera de la función como tal.

Efectos – UseEffect vs WatchEffect

Mientras que en ReactJs usamos UseEffect para actualizar dependencias de manera dinámica en vue3 usamos WatchEffect que este hace lo mismo, pero detecta las dependencias de manera automática, siendo mucho más eficiente.

React

```
// Efecto con dependencias específicas
useEffect(() => {
  console.log(contactList);
  showEmpty('No se encontraron resultados');
}, [contactList]);

// Efecto que se ejecuta solo una vez
useEffect(() => {
  // setup inicial
}, []);
```

Vue

```
// watchEffect - equivale a useEffect sin dependencias explícitas
watchEffect(() => {
  console.log(contactList.value);
  showEmpy('No se encontro resultados');
});

// watch - equivale a useEffect con dependencias específicas
watch(contactList, (newValue, oldValue) => {
  console.log(newValue, oldValue);
});

// onMounted - equivale a useEffect con []
import { onMounted } from 'vue';
onMounted(() => {
  // setup inicial
});
```

Como podemos observar WatchEffect detecta las dependencias automáticamente, sin embargo, también existen otras palabras reservadas como Watch que equivale a un UseEffect con dependencias especificas de React. O OnMounted que básicamente es como hacer un Setup inicial con React.

Comunicación de Componentes

La comunicación entre componentes también es algo que cambia en cierto modo puesto que en React para comunicarnos de **Padre – Hijo** usábamos directamente Props en la llamada del componente.

De la siguiente manera:

```
// Padre
<ListContacts title="Mi Lista" onFilter={handleFilter} />

// Hijo
function ListContacts({ title, onFilter }) {
  return <h3>{title}</h3>;
}
```

Como podemos ver el Hijo recibe esos datos por medio de las props que contiene la función propia del componente.

En Vue 3 funciona distinto pues mientras que el padre envía la información al hijo de la misma forma por medio de props en el llamado a el componente hijo.

```
<!-- Padre -->
<ListContacts
  title="Mi Lista"
  @filter-contact="handleFilter" />
```

El hijo no recibe esa información de la misma forma que en React, ya que como no maneja como tal una función.

Estos datos se reciben bajo la definición de esos props en el componente hijo, con la palabra reservada asignada para definir props. De la siguiente manera:

```
<!-- Hijo -->
<script setup>
const props = defineProps({
  title: { type: String, default: 'lista' }
});
</script>
```

Comunicación Hijo – Padre

Mientas que React es mas directo a la hora de la comunicación de Hijo – Padre Vue es más basado en eventos.

Es decir, en React tenemos la siguiente forma de comunicarnos de hijo a padre:

```
// 🤔 Padre: "Hijo, aquí tienes esta función para que la uses"
function Padre() {
  const miFuncion = (datos) => {
    console.log('El hijo me envió:', datos);
  };

  return (
    // 👇 Le paso la función como prop
    <Hijo onAlgoQueHaga={miFuncion} />
  );
}

// 🧒 Hijo: "Ok papá, recibo tu función y la uso cuando necesite"
function Hijo({ onAlgoQueHaga }) {
  const handleClick = () => {
    // 👇 Uso la función del padre
    onAlgoQueHaga('¡Hola desde el hijo!');
  };

  return <button onClick={handleClick}>Clickéame</button>;
}
```

Como podemos observar el padre le está dando a el hijo una función propia de él, por medio de props. Ahora el hijo recibe esta función por props y el decide cuando la usa.

Siguiendo esta analogía en ese caso de comunicación:

```
Padre: "Hijo, aquí tienes mi teléfono 📱, llámame cuando necesites algo"
Hijo: *marca el teléfono directamente* ☎️
```

En el caso de Vue 3 como bien dije se enfoca mas a eventos por lo tanto el padre no le da a disposición como tal una función directamente, si no que esta preparado para escuchar eventos del hijo para él así poder reaccionar.

De la siguiente manera.

```
<!-- 👴 Padre: "Hijo, voy a estar escuchando tu evento" -->
<script setup>
const miFuncion = (datos) => {
  console.log('El hijo me envió:', datos);
};
</script>

<template>
  <!-- 👇 Escucho el evento del hijo -->
  <Hijo @algo-que-haga="miFuncion" />
</template>
```

En este caso podemos ver como el padre crea una función en la cual espera recibir datos del hijo (por decirlo así el canal por el cual escuchara los eventos). Luego en los props del componente del hijo se prepara con esa función par poder escuchar los posibles eventos que emita el hijo.

Por el lado del hijo tendríamos lo siguiente:

```
<!-- 👶 Hijo: "Ok papá, cuando algo pase te voy a enviar un evento" -->
<script setup>
const emit = defineEmits(['algoQueHaga']);

const handleClick = () => {
  // 👇 Emito evento al padre
  emit('algoQueHaga', '¡Hola desde el hijo!');
};
</script>

<template>
  <button @click="handleClick">Clickéame</button>
</template>
```

Como observamos el hijo lo primero que hace es definir el canal o los canales por los cuales más adelante podrá enviar información. En este caso el padre esta escuchando por el canal [algoQueHaga], por ende, el hijo tiene que definir ese canal para mas adelante poder enviar información.

Una vez definido el hijo ya puede emitir eventos al padre con información a través de ese canal dedicado entre los dos.

Por ello es que luego llama al canal que había definido arriba y seguido a el pone el mensaje que quiere que escuche el padre.

Un ejemplo más entendible seria este:

- Padre define el canal por el cual escuchara

```
<!-- 👴 Padre: "Voy a estar escuchando en el canal 'filterContact'" -->
<template>
  <SearchContacts @filterContact="miFuncionQueEscucha" />
  <!--                👆 Canal especifico -->
</template>
```

- Hijo define los canales por los cuales él puede transmitir

```
<script setup>
// 👶 Hijo: "Papá, te aviso que puedo transmitir por estos canales:"
const emit = defineEmits(['filterContact', 'clearData', 'userClick']);
//                        👆 Canal 1    👆 Canal 2    👆 Canal 3
</script>
```

- Hijo transmitiendo por el canal que escucha el papa

```
<script setup>
const enviarDatos = () => {
  // 👶 Hijo: "Transmitiendo por canal 'filterContact'..."
  emit('filterContact', 'datos importantes');
  //    👆 Canal especifico que el padre está escuchando
};
</script>
```

Características Propias de Vue

- **Directivas Vue**
  Las directivas son atributos que empiezan con V- que le dan instrucciones al HTML sobre cómo debe comportarse dinámicamente.
  Entre las directivas principales tenemos las siguientes:
  - **V-if / V-else / V-else-if**

```
<template>
  <!-- Solo se renderiza SI la condición es true -->
  <div v-if="isLoggedIn">
    Bienvenido usuario
  </div>

  <div v-else-if="isLoading">
    Cargando...
  </div>

  <div v-else>
    Por favor inicia sesión
  </div>
</template>
```

Básicamente funcionan como condicionales en los que v-if renderiza lo que contiene el div si la variable a la que es igual existe, de lo contrario entra en juego el v-else-if que iguala a otra variable y renderiza algo diferente y por último si ninguna de las dos pasas, pues directamente se iría por el V-else que es la última opción.

Así que si, funciona tal cual como los condicionales de JS

- **Atributos dinámicos**

```
<template>
  <!-- Forma completa -->
  <img v-bind:src="imageUrl" v-bind:alt="imageAlt">

  <!-- Forma abreviada (más común) -->
  <img :src="imageUrl" :alt="imageAlt">

  <!-- Clases dinámicas -->
  <div :class="{ active: isActive, disabled: isDisabled }">
    Contenido
  </div>

  <!-- Múltiples clases -->
  <div :class="[baseClass, { active: isActive }]">
    Contenido
  </div>

  <!-- Estilos dinámicos -->
  <div :style="{ color: textColor, fontSize: size + 'px' }">
    Texto con estilo dinámico
  </div>

  <!-- Atributos booleanos -->
  <button :disabled="isLoading">Guardar</button>
</template>
```

Como podemos observar también podemos poner atributos dinámicos a través de dos formas la forma completa:

```
<!-- Forma completa -->
<img v-bind:src="imageUrl" v-bind:alt="imageAlt">
```

O de la forma abreviada que es la mas general ya que pues se evita escribir demás

```
<!-- Forma abreviada (más común) -->
<img :src="imageUrl" :alt="imageAlt">
```

- **Event Listeners**

Básicamente también tenemos en Vue palabras reservadas para poder escuchar eventos del DOM como el OnClick de toda la vida de JS y demás eventos que se puedan dar, en este caso la forma en la que lo hace Vue es la siguiente:

```
<template>
  <!-- Forma completa -->
  <button v-on:click="handleClick">Click me</button>

  <!-- Forma abreviada (más común) -->
  <button @click="handleClick">Click me</button>

  <!-- Con parámetros -->
  <button @click="handleClick('param1', 'param2')">Click</button>

  <!-- Con modificadores -->
  <button @click.prevent="handleClick">Sin submit</button>
  <form @submit.prevent="onSubmit">Formulario</form>
  <input @keyup.enter="search">Presiona Enter</input>
  <div @click.stop="handleClick">No hace bubble</div>

  <!-- Múltiples eventos -->
  <input
    @focus="onFocus"
    @blur="onBlur"
    @input="onInput"
  >
</template>
```

Basicamente se usa un @con una abreviatura del evento en el caso de OnClick seria @click. Aquí tenemos la tabla mas desglosada de los eventos y la sintaxis que hay tanto JS tradicional y Vue 3

JavaScript tradicional	Vue 3
`onclick="func()"`	`@click="func"`
`onchange="func()"`	`@change="func"`
`oninput="func()"`	`@input="func"`
`onfocus="func()"`	`@focus="func"`
`onblur="func()"`	`@blur="func"`
`onsubmit="func()"`	`@submit="func"`
`onkeyup="func()"`	`@keyup="func"`
`onmouseover="func()"`	`@mouseover="func"`

Enlace Git Hub

https://github.com/neunapp/CursoVue3.git