Vue.js
Página de Atajos

{dev/talles}

Iniciar un proyecto: (Recomendado)

```
npm init vue@latest
```

Proyecto con Vite: (opcional)

```
npm create vite
yarn create vite
pnpm create vite
```

Usar desde un CDN:

```
<script src="https://unpkg.com/vue@3/
dist/vue.global.js"></script>
```

Ejemplo:

```
<script src="https://unpkg.com/
vue@3/dist/vue.global.js"></script>

<div id="app">{{ message }}</div>

<script>
  const { createApp } = Vue;

  createApp({
    data() {
      return {
        message: 'Hello Vue!'
      }
    }
  }).mount('#app');
</script>
```

Configuraciones comunes en el main.ts

```
import { createApp } from 'vue';
import { createPinia } from 'pinia';

import App from './App.vue';
import router from './router';

import './assets/main.css';

const app = createApp(App);

app.config.errorHandler = (err) => {
    /* handle error */
}

app.use(createPinia());
app.use(router);

app.mount('#app');
```

Registrar App-Scoped assets. (Ej: Componente)

```
app.component(
  'TodoDeleteButton', TodoDeleteButton
);
```

Directivas

Las directivas en **Vue.js**, tiene el prefijo **v-**, estas directivas esperan una expresión de JavaScript con la excepción de **v-for, v-on, v-slot**.

Raw HTML

```
<p>Using text interpolation: {{ rawHtml }}</p>
<p>Using v-html directive:
   <span v-html="rawHtml"></span>
</p>
```

Using text interpolation: This should be red.
Using v-html directive: This should be red.

v-bind

```
<div v-bind:id="dynamicId"></div>

// Format corta
<div :id="dynamicId"></div>

// Válido también
<div :id="`list-${id}`"></div>
```

Enlazar multiples propiedades a la vez

```
const objectOfAttrs = {
  id: 'container',
  class: 'wrapper'
}

<div v-bind="objectOfAttrs"></div>
```

Atributos dinámicos:

```
<a v-on:
[eventName]="doSomething"> ... </a>

<!-- shorthand -->
<a @[eventName]="doSomething">
```

v-on

```
<a v-on:click="doSomething"> ... </a>

<!-- shorthand -->
<a @click="doSomething"> ... </a>
```

Vue.js
Página de Atajos

{dev/talles}

TypeScript con CompositionAPI

Inferir props: (TypeScript realizará el tipado)

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

const props = defineProps({
  foo: { type: String, required: true },
  bar: Number
});

props.foo; // string
props.bar; // number | undefined

</script>
```

defineProps es un genérico y se puede especificar en línea, con interfaces o clases.
(No es necesario importar **defineProps** - es global)

```
const props = defineProps<{
  foo: string
  bar?: number
}>();

// Con interfaces/clases
interface Props {
  foo: string;
  bar?: number;
}

const props = defineProps<Props>();
```

IMPORTANTE: Esto no es válido

```
import { Props } from './other-file';

// No es soportado
defineProps<Props>();
```

defineProps con **withDefaults.** (Docs)
Cuando se usa TS para los props, perdemos la posibilidad de valores por defecto. Para esto podemos usar 2 formas:

#1

```
export interface Props {
    msg?: string
    labels?: string[]
}

const props =
withDefaults(defineProps<Props>(), {
    msg: 'hello',
    labels: () => ['one', 'two']
});
```

#2

```
interface Props {
    name: string
    count?: number
}

// reactive destructure for defineProps()
const {
   name,
   count = 100
} = defineProps<Props>();
```

Fuera del Script Setup, es necesario usar el **defineComponent( )** para habilitar props y su tipado.

```
import { defineComponent } from 'vue';

export default defineComponent({
  props: {
    message: String
  },
  setup(props) {
    props.message //type: string
  }
});
```

defineEmits dentro del Script Setup puede ser inferida o con declaración de tipo.
(No es necesario importar **defineEmits** - es global)

```
// runtime
const emit = defineEmits(['change', 'update']);

// type-based
const emit = defineEmits<{
  (e: 'change', id: number): void
  (e: 'update', value: string): void
}>();
```

Fuera del Script Setup, hay que utilizar **defineComponent( )** para obtener y llamar el emit.

```
import { defineComponent } from 'vue';

export default defineComponent({
  emits: ['change'],
  setup(props, { emit }) {
    // type check / auto-completion
    emit('change');
  }
});
```

Vue.js
Página de Atajos

{dev/talles}

Reactividad de Vue.js

Asignando el tipo a ref( )
La función ref, crea un objeto reactivo, el cual puede inferir el tipo o usarlo como un genérico.

```
import { ref } from 'vue';

// inferred type: Ref<number>
const year = ref(2020);

// => TS Error: Type 'string' is not
assignable to type 'number'.
year.value = '2020';
```

También:

```
import { ref } from 'vue';
import type { Ref } from 'vue';

const year: Ref<string | number> = ref('2020');

year.value = 2020; // ok!
```

Personalmente mi favorita:

```
import { ref } from 'vue';

// Type: Ref<string | number>
const year = ref<string | number>('2020');

year.value = 2020; // ok!
```

Asignando el tipo a reactive( )
La función reactive, convierte un objeto normal a un objeto reactivo. (Primitivos no son soportados)

```
import { reactive } from 'vue';

// inferred type: { title: string }
const book = reactive({
   title: 'Vue Guide'
});
```

Es interesa que se pueden usar interfaces simples con la función reactive. (No usarlo como genérico)

```
import { reactive } from 'vue';

interface Book {
  title: string
  year?: number
}

const book: Book = reactive({
  title: 'Vue Guide'
});
```

Asignando el tipo a computed( )
TS puede inferir el tipo basado en el retorno, pero se aconseja especificarlo.

```
import { ref, computed } from 'vue';

const count = ref(0);

// inferred type: ComputedRef<number>
const double = computed(() => count.value * 2);


// Mediante genéricos
const double = computed<number>(() => {
  // type error
  // if this doesn't return a number
})
```

Asignando el tipo a **eventos**:

```
<script setup lang="ts">
function handleChange(event) {
  // `event` implicitly has `any` type
  console.log(event.target.value)
}
</script>

<template>
  <input
     type="text"
     @change="handleChange" />

</template>
```

Lo anterior puede dar errores por el tipo any, por lo cual asignar Event y luego hacer un cast.

```
function handleChange(event: Event) {
  console.log(
    (event.target as HTMLInputElement).value
  );
}
```

Asignando el tipo a referencias HTML

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

const el = ref<HTMLInputElement | null>(null);

onMounted(() => {
  el.value?.focus();
});
</script>

<template>
  <input ref="el" />
</template>
```

Vue.js
Página de Atajos

{dev/talles}

Asignar el tipo a referencias en el Template:
Útil cuando quieres exponer información entre componentes padre e hijo usando Script Setup.

```vue
<!-- MyModal.vue -->
<script setup lang="ts">
import { ref } from 'vue';

const isContentShown = ref(false);
const open = () =>
    (isContentShown.value = true);

defineExpose({
  open
});
</script>
```

Y para poder obtener la instancia de "MyModal", necesitamos obtener el tipo vía **typeof** y el **InstanceType**:

```vue
<!-- App.vue -->
<script setup lang="ts">
import MyModal from './MyModal.vue';

const modal = ref<InstanceType<typeof MyModal> | null>(null);

const openModal = () => {
  modal.value?.open()
}
</script>
```

State Management:

Reactive Objects:
Vue cuenta con una forma nativa te trabajar con un store global usando la función **reactive( )**.

```vue
// store.js
import { reactive } from 'vue';

export const store = reactive({
  count: 0,
  increment() {
    this.count++
  }
});
```

```vue
<template>
  <button @click="store.increment()">
    From B: {{ store.count }}
  </button>
</template>
```

También puedes construir tu Store con una combinación de funciones reactivas de Vue.

```vue
import { ref } from 'vue';

// Global state,
// Creado en el scope del módulo
const globalCount = ref(1);

export function useCount() {
  // local state, creado por componente
  const localCount = ref(1);

  return {
    globalCount,
    localCount
  }
};
```

Pinia Store

Pinia es el gestor de estado actualmente recomendado por Evan You, autor y creador de Vite y Vue.

Instalar Pinia

```
yarn add pinia
npm install pinia
```

```vue
import { createApp } from 'vue';
import { createPinia } from 'pinia';
import App from './App.vue';

const pinia = createPinia();
const app = createApp(App);

app.use(pinia);
app.mount('#app');
```

Ejemplo de Pinia Store - Option Store

```vue
export const useCounterStore =
 defineStore('counter', {
   state: () => ({
      count: 0,
      name: 'Eduardo'
   }),
   getters: {
    doubleCount: (state) => state.count * 2,
   },
    actions: {
     increment() {
     this.count++
     },
    },
});
```

Vue.js
Página de Atajos

{dev/talles}

Ejemplo de Pinia Setup Store. Usando la composición reactiva de **Vue**.

```
export const useCounterStore = defineStore('counter', () =>
{
    const count = ref(0);
    const name = ref('Eduardo');


   const doubleCount = computed(() => count.value * 2);
    function increment() {
      count.value++
    }

    return { count, name, doubleCount, increment };
});
```

Dentro del **Setup Store**:

- **ref( )'s** se vuelven propiedades de estado
- **computed( )'s,** se vuelven getters
- **functions( )'s**, se vuelven acciones

¿Cuál de las dos formas usar?
Usa la que se haga más familiar para ti o resulte más simple, si no estás seguro, empieza con Option Stores.

Consumir el store:

```
import { useCounterStore } from '@/stores/counter'

export default {
  setup() {
    const store = useCounterStore();

    return {
    //Puedes regresar toda la instancia del store
    // y así usarlo en el template
      store,
    }
  },
}
```

IMPORTANTE: Esto no funciona

```
const store = useCounterStore();
// ❌ esto no funciona, rompe la reactividad
// Es lo mismo que desestructurar Props
const { name, doubleCount } = store;
```

En caso de necesidad, usar **storeToRefs( )**

```
import { storeToRefs } from 'pinia';

export default defineComponent({
  setup() {
    const store = useCounterStore();

    // State a variables reactivas
    const { name, doubleCount } = storeToRefs(store);

    // Acciones pueden ser extraídas directamente
    const { increment } = store;

    return {
      name,
      doubleCount,
      increment,
    }
  },
});
```

Se puede re-establecer el store llamando **$reset( )**

```
const store = useStore();

store.$reset();
```

Se puede mutar diferentes piezas del store con **$patch**

```
store.$patch({
  count: store.count + 1,
  age: 120,
  name: 'DIO',
});
```

Alternativamente se puede llamar con un callback para obtener el state del store.

```
cartStore.$patch((state) => {
  state.items.push({
    name: 'shoes',
    quantity: 1
  });
  state.hasChanged = true;
});
```

Se puede **suscribir para escuchar** cambios del store, pero la **ventaja sobre el $watch de Vue**, es que la suscripción se dispara una vez después del patch.

```
cartStore.$subscribe((mutation, state) => {
  // import { MutationType } from 'pinia'
  mutation.type; // 'direct' | 'patch object' | 'patch
function'
  // mismo que el cartStore.$id
  mutation.storeId; // 'cart'
  // Sólo disponible con mutation.type === 'patch object'
  mutation.payload; // objeto pasado al cartStore.$patch()

  // Mantener todo el state en el localstorage
  localStorage.setItem('cart', JSON.stringify(state));
});
```

Usar otros stores dentro de un store.

```
import { useOtherStore } from './other-store';

export const useStore = defineStore('main', {
  state: () => ({
    // ...
  }),
  getters: {
    otherGetter(state) {
      const otherStore = useOtherStore();
      return state.localData + otherStore.data;
    },
  },
});
```

Sitios web útiles para continuar:
github.com/vuejs/awesome-vue
next.attojs.org/
pinia.vuejs.org/
github.com/vuejs/apollo