Composable Vue

Pattens and tips for writing good composable logic in Vue

ANTHONY FU

VueDay
Apr. 29th, 2021

Anthony Fu

Vue core team member and Vite team member.

Creator of VueUse, i18n Ally and Type Challenges.

A fanatical full-time open sourceror.

antfu

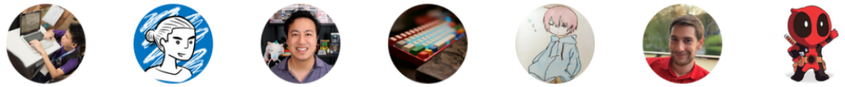
antfu7

antfu.me

Gold Sponsors

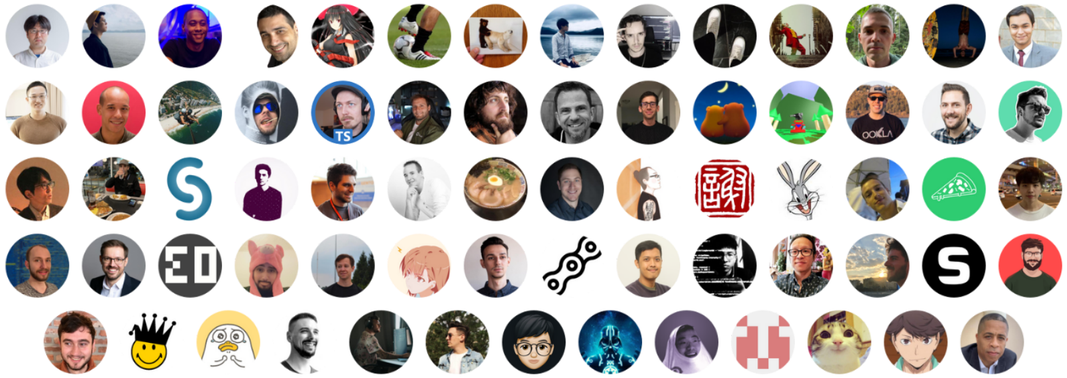
Leniolabs_ Nuxt Vue Mastery Evan You

Sponsors



IU hiroki osame Ben Hong PENG Rui 741A81F48FDC Jan-Henrik FallDownTh...

Backers



Sponsor me at GitHub

Composable Vue

Collection of essential Vue Composition Utilities

v4.9.0 | 71k/month | docs & demos | 114 functions

Stars 3.8k

- Works for both Vue 2 and 3
- Tree-shakeable ESM
- CDN compatible
- TypeScript
- Rich ecosystems

Composition API

a brief go-through

Ref

```
import { ref } from 'vue'

let foo = 0
let bar = ref(0)

foo = 1
bar = 1 // ts-error
```

PROS

- More explicit, with type checking
- Less caveats

CONS

- `.value`

Reactive

```
import { reactive } from 'vue'

const foo = { prop: 0 }
const bar = reactive({ prop: 0 })

foo.prop = 1
bar.prop = 1
```

PROS

- Auto unwrapping (a.k.a `.value` free)

CONS

- Same as plain objects on types
- Destructure loses reactivity
- Need to use callback for `watch`

Ref Auto Unwrapping

Core

Get rid of `.value` for most of the time.

- `watch` accepts ref as the watch target, and returns the unwrapped value in the callback

```
const counter = ref(0)

watch(counter, count => {
  console.log(count) // same as `counter.value`
})
```

- Ref is auto unwrapped in the template

```
<template>
  <button @click="counter += 1">
    Counter is {{ counter }}
  </button>
</template>
```

- Reactive will auto-unwrap nested refs.

```
import { ref, reactive } from 'vue'
const foo = ref('bar')
const data = reactive({ foo, id: 10 })
data.foo // 'bar'
```

`unref` - Oppsite of Ref Core

- If it gets a Ref, returns the value of it.
- Otherwise, returns as-is.

IMPLEMENTATION

```
function unref<T>(r: Ref<T> | T): T {
  return isRef(r) ? r.value : r
}
```

USAGE

```
import { unref, ref } from 'vue'

const foo = ref('foo')
unref(foo) // 'foo'

const bar = 'bar'
unref(bar) // 'bar'
```

Patterns & Tips

of writing composable functions

What's Composable Functions

Sets of reusable logic, separation of concerns.

```
export function useDark(options: UseDarkOptions = {}) {
  const preferredDark = usePreferredDark()          // ⟵
  const store = useStorage('vueuse-dark', 'auto')  // ⟵

  return computed<boolean>({
    get() {
      return store.value === 'auto'
        ? preferredDark.value
        : store.value === 'dark'
    },
    set(v) {
      store.value = v === preferredDark.value
        ? 'auto' : v ? 'dark' : 'light'
    },
  })
}
```

Light

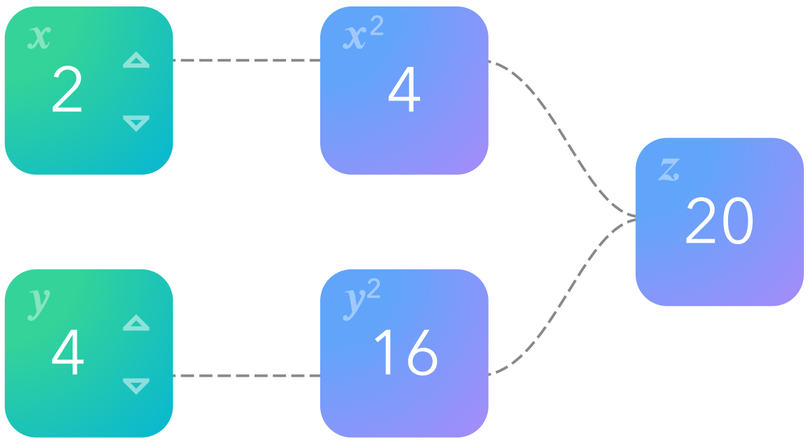
Avaliable in VueUse: `useDark`

Think as "Connections"

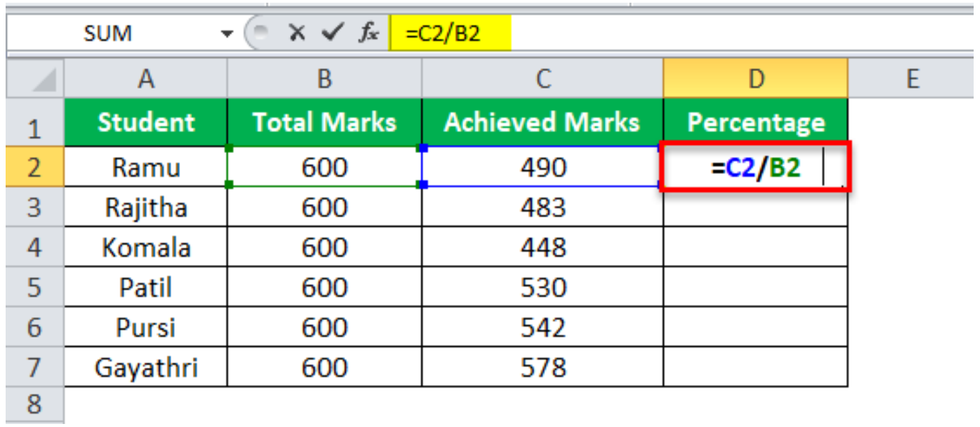
The `setup()` only runs **once** on component initialization, to construct the relations between your state and logic.

- Input → Output Effects
- Output reflects to input's changes automatically

$z=x^2+y^2$=2x2+4x4=20



SPREADSHEET FORMULA



	A	B	C	D	E
1	Student	Total Marks	Achieved Marks	Percentage	
2	Ramu	600	490	=C2/B2	
3	Rajitha	600	483		
4	Komala	600	448		
5	Patil	600	530		
6	Pursi	600	542		
7	Gayathri	600	578		
8					

One Thing at a Time

Just the same as authoring JavaScript functions.

- Extract duplicated logics into composable functions
- Have meaningful names
- Consistent naming conversions - `useXX` `createXX` `onXX`
- Keep function small and simple
- "Do one thing, and do it well"

Passing Refs as Arguments

Pattern

	IMPLEMENTATION	USAGE

Plain function	`function add(a: number, b: number) {`
	`  return a + b`
	`}`
		`let a = 1`
		`let b = 2`
		
		`let c = add(a, b) // 3`

Accpets refs, returns a reactive result.	`function add(a: Ref<number>, b: Ref<number>) {`
	`  return computed(() => a.value + b.value)`
	`}`
		`const a = ref(1)`
		`const b = ref(2)`
		
		`const c = add(a, b)`
		`c.value // 3`

Accpets both refs and plain values.	`function add(`
	`  a: Ref<number> | number,`
	`  b: Ref<number> | number`
	`) {`
	`  return computed(() => unref(a) + unref(b))`
	`}`
		`const a = ref(1)`
		
		`const c = add(a, 5)`
		`c.value // 6`

MaybeRef Tips

A custom type helper

```
type MaybeRef<T> = Ref<T> | T
```

In VueUse, we use this helper heavily to support optional reactive arguments

```
export function useTimeAgo(
  time: Date | number | string | Ref<Date | number | string>,
) {
  return computed(() => someFormating(unref(time)))
}
```

```
import { computed, unref, Ref } from 'vue'

type MaybeRef<T> = Ref<T> | T

export function useTimeAgo(
  time: MaybeRef<Date | number | string>,
) {
  return computed(() => someFormating(unref(time)))
}
```

Make it Flexible

Pattern

Make your functions like LEGO, can be used with different components in different ways.

CREATE A "SPECIAL" REF

```
import { useTitle } from '@vueuse/core'

const title = useTitle()

title.value = 'Hello World'
// now the page's title changed
```

BINDING AN EXISTING REF

```
import { ref, computed } from 'vue'
import { useTitle } from '@vueuse/core'

const name = ref('Hello')
const title = computed(() => {
  return `${name.value} - World`
})

useTitle(title) // Hello - World

name.value = 'Hi' // Hi - World
```

Avaliable in VueUse: `useTitle`

`useTitle` Case

Take a look at `useTitle`'s implementation

```
import { ref, watch } from 'vue'
import { MaybeRef } from '@vueuse/core'

export function useTitle(
  newTitle: MaybeRef<string | null | undefined>
) {
  const title = ref(newTitle || document.title)

  watch(title, (t) => {
    if (t != null)
      document.title = t
  }, { immediate: true })

  return title
}
```

```
⟵ 1. use the user provided ref or create a new one

⟵ 2. sync ref changes to the document title
```

"Reuse" Ref `Core`

If you pass a `ref` into `ref()`, it will return the original ref as-is.

```
const foo = ref(1)   // Ref<1>
const bar = ref(foo) // Ref<1>

foo === bar // true
```

```
function useFoo(foo: Ref<string> | string) {
  // no need!
  const bar = isRef(foo) ? foo : ref(foo)

  // they are the same
  const bar = ref(foo)

  /* ... */
}
```

Extremely useful in composable functions that take uncertain argument types.

`ref` / `unref` Tips

- `MaybeRef<T>` works well with `ref` and `unref`.
- Use `ref()` when you want to normalized it as a Ref.
- Use `unref()` when you want to have the value.

```
type MaybeRef<T> = Ref<T> | T

function useBala<T>(arg: MaybeRef<T>) {
  const reference = ref(arg) // get the ref
  const value = unref(arg)   // get the value
}
```

Object of Refs `Pattern`

Getting benefits from both `ref` and `reactive` for authoring composable functions

```
import { ref, reactive } from 'vue'

function useMouse() {
  return {
    x: ref(0),
    y: ref(0)
  }
}

const { x, y } = useMouse()
const mouse = reactive(useMouse())

mouse.x === x.value // true
```

- Destructurable as Ref
- Convert to reactive object to get the auto-unwrapping when needed

Async to "Sync" Tips

With Composition API, we can actually turn async data into "sync"

ASYNC

```
const data = await fetch('https://api.github.com/').then(r => r.json())

// use data
```

COMPOSITION API

```
const { data } = useFetch('https://api.github.com/').json()

const user_url = computed(() => data.value?.user_url)
```

Establish the "Connections" first, then wait data to be filled up. The idea is similar to SWR (stale-while-revalidate)

`useFetch`

Case

```
export function useFetch<R>(url: MaybeRef<string>) {
  const data = shallowRef<T | undefined>()
  const error = shallowRef<Error | undefined>()

  fetch(unref(url))
    .then(r => r.json())
    .then(r => data.value = r)
    .catch(e => error.value = e)

  return {
    data,
    error
  }
}
```

Avaliable in VueUse: useFetch

Side-effects Self Cleanup `Pattern`

The `watch` and `computed` will stop themselves on components unmounted.
We'd recommend following the same pattern for your custom composable functions.

```
import { onUnmounted } from 'vue'

export function useEventListener(target: EventTarget, name: string, fn: any) {
  target.addEventListener(name, fn)

  onUnmounted(() => {
    target.removeEventListener(name, fn) // ←
  })
}
```

Avaliable in VueUse: `useEventListener`

`effectScope` RFC Upcoming

A new API to collect the side effects automatically. Likely to be shipped with Vue 3.1
https://github.com/vuejs/rfcs/pull/212

```
// effect, computed, watch, watchEffect created inside the scope will be collected

const scope = effectScope(() => {
  const doubled = computed(() => counter.value * 2)

  watch(doubled, () => console.log(double.value))

  watchEffect(() => console.log('Count: ', double.value))
})

// dispose all effects in the scope
stop(scope)
```

Typed Provide / Inject `Core`

Use the `InjectionKey<T>` helper from Vue to share types across context.

```
// context.ts
import { InjectionKey } from 'vue'

export interface UserInfo {
  id: number
  name: string
}

export const injectKeyUser: InjectionKey<UserInfo> = Symbol()
```

Typed Provide / Inject

Core

Import the key from the same module for `provide` and `inject`.

```vue
// parent.vue
import { provide } from 'vue'
import { injectKeyUser } from './context'

export default {
  setup() {
    provide(injectKeyUser, {
      id: '7', // type error: should be number
      name: 'Anthony'
    })
  }
}
```

```vue
// child.vue
import { inject } from 'vue'
import { injectKeyUser } from './context'

export default {
  setup() {
    const user = inject(injectKeyUser)
    // UserInfo | undefined

    if (user)
      console.log(user.name) // Anthony
  }
}
```

Shared State

Pattern

By the nature of Composition API, states can be created and used independently.

```
// shared.ts
import { reactive } from 'vue'

export const state = reactive({
  foo: 1,
  bar: 'Hello'
})
```

```
// A.vue
import { state } from './shared.ts'

state.foo += 1
```

```
// B.vue
import { state } from './shared.ts'

console.log(state.foo) // 2
```

⚠ BUT IT'S NOT SSR COMPATIBLE!

Shared State (SSR friendly) `Pattern`

Use `provide` and `inject` to share the app-level state

```
export const myStateKey: InjectionKey<MyState> = Symbol()

export function createMyState() {
  const state = {
    /* ... */
  }

  return {
    install(app: App) {
      app.provide(myStateKey, state)
    }
  }
}

export function useMyState(): MyState {
  return inject(myStateKey)!
}
```

```
// main.ts
const App = createApp(App)

app.use(createMyState())
```

```
// A.vue

// use everywhere in your app
const state = useMyState()
```

- Vue Router v4 is using the similar approach

useVModel

Tips

A helper to make props/emit easier

```
export function useVModel(props, name) {
  const emit = getCurrentInstance().emit

  return computed({
    get() {
      return props[name]
    },
    set(v) {
      emit(`update:${name}`, v)
    }
  })
}
```

```
export default defineComponent({
  setup(props) {
    const value = useVModel(props, 'value')

    return { value }
  }
})
```

```
<template>
  <input v-model="value" />
</template>
```

Avaliable in VueUse: `useVModel`

All of them work for both Vue 2 and 3

`@vue/composition-api` Lib

Composition API support for Vue 2.

vuejs/composition-api

```
import Vue from 'vue'
import VueCompositionAPI from '@vue/composition-api'

Vue.use(VueCompositionAPI)
```

```
import { ref, reactive } from '@vue/composition-api'
```

Vue 2.7 `Upcoming`

Plans in Vue 2.7

- Backport `@vue/composition-api` into Vue 2's core.
- `<script setup>` syntax in Single-File Components.
- Migrate codebase to TypeScript.
- IE11 support.
- LTS.

Vue Demi

Lib

Creates Universal Library for Vue 2 & 3

vueuse/vue-demi

```
// same syntax for both Vue 2 and 3
import { ref, reactive, defineComponent } from 'vue-demi'
```

23 VueDemi

Creates Universal Library for Vue 2 & 3

Recap

- Think as "Connections"
- One thing as a time
- Accepting ref as arguments
- Returns an object of refs
- Make functions flexible
- Async to "sync"
- Side-effect self clean up
- Shared state

Thank You!

Slides can be found on antfu.me