Watchers

A Watcher in Vue is a way to watch for changes in reactive data (ref, reactive, or computed properties) and run custom logic whenever that data changes.

we need Watchers

- Even though Vue's reactivity system handles a lot with computed, sometimes you:
- Need to react to a data change (like API calls, form validation).
- Want to perform side effects (like logging, updating local storage, animations).
- Have asynchronous actions that must run only when specific data changes.

That’s where watchers shine. Unlike computed, which is for deriving values, watchers are for doing something when values change.

1. watch()

In Vue 3, watch() is a Composition API function that allows you to perform side effects (like logging, API calls, form validation, etc.) in response to reactive state changes.

we need watch()

- To run code when a reactive value changes
- To handle asynchronous operations like fetching data
- To watch multiple sources and respond to their changes
- To deep-watch nested reactive objects
- To validate form fields
- To cache or save data automatically (like saving draft on change)

You're a security guard watching only one door. You say: "Hey Vue, I want you to keep an eye on that specific door. And whenever someone opens or closes it, tell me what changed." That’s exactly what watch() does. It's explicit you tell Vue what to watch, and it tells you when it changes even giving you before and after.

syntax

```
watch(source, callback, options?)
```

source : A ref, reactive, computed, or getter function callback : A function with (newVal, oldVal, onCleanup) options : Optional object { immediate, deep, flush }

Watching a ref()

Watches a single ref (count) and logs whenever its value changes.

```
<template>
  <div>
    <h3>Count: {{ count }}</h3>
    <button @click="count++">Increment</button>
  </div>
</template>

<script setup>
import { ref, watch } from 'vue'

const count = ref(0)

watch(count, (newVal, oldVal) => {
  console.log(`Count changed from ${oldVal} to ${newVal}`)
})
</script>
```

Watch a Computed Getter

Watches the output of a computed property and reacts when it changes. Use when you want to react to specific data changes — like triggering side-effects, animations, or saving changes.

```
<template>
  <div>
    <input v-model="user.firstName" placeholder="First Name" />
    <input v-model="user.lastName" placeholder="Last Name" />
    <p>Full Name: {{ fullName }}</p>
  </div>
</template>

<script setup>
import { reactive, computed, watch } from 'vue'

const user = reactive({ firstName: 'Alice', lastName: 'Smith' })

const fullName = computed(() => `${user.firstName} ${user.lastName}`)

watch(() => fullName.value, (newVal, oldVal) => {
  console.log(`Full name changed from ${oldVal} to ${newVal}`)
})
</script>
```

Immediate Watch (Run Once Immediately)

Runs the watch callback immediately on component load (besides subsequent updates). Useful when you want to track changes in a derived value like fullName.

```
<template>
  <input v-model="email" placeholder="Enter email" />
</template>

<script setup>
```

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

const email = ref('test@example.com')

watch(email, (newVal) => {
  console.log('Watching immediately:', newVal)
}, { immediate: true })
</script>
```

Immediate Watch (Run Once Immediately)

Runs the watch callback immediately on component load (besides subsequent updates). When you want to initialize something right away using the current value.

```
<template>
  <input v-model="email" placeholder="Enter email" />
</template>

<script setup>
import { ref, watch } from 'vue'

const email = ref('test@example.com')

watch(email, (newVal) => {
  console.log('Watching immediately:', newVal)
}, { immediate: true })
</script>
```

Deep Watching

Watches deeply nested properties inside a reactive object. When working with nested objects (like form data) where you want to track any internal change.

```
<template>
  <div>
    <input v-model="profile.user.name" />
    <input type="number" v-model="profile.user.age" />
  </div>
</template>

<script setup>
import { reactive, watch } from 'vue'

const profile = reactive({
  user: {
    name: 'Alice',
    age: 25
  }
})
```

```
watch(profile, (newVal) => {
  console.log('Nested object changed:', newVal)
}, { deep: true })
</script>
```

Watch Multiple Sources

Watches an array of sources and reacts when any one of them changes. Perfect when multiple values (like filters or form fields) affect a process and you want to respond to all.

```
<template>
  <div>
    <input v-model="a" />
    <input v-model="b" />
  </div>
</template>

<script setup>
import { ref, watch } from 'vue'

const a = ref('A')
const b = ref('B')

watch([a, b], ([newA, newB], [oldA, oldB]) => {
  console.log(`a: ${oldA} → ${newA}, b: ${oldB} → ${newB}`)
})
</script>
```

2. watchEffect()

watchEffect() automatically tracks reactive dependencies and re-runs the effect whenever any of them change.

why we need

- Automatically collects dependencies, no need to declare them explicitly like watch().
- Cleaner and more declarative.
- Ideal for side effects that should stay in sync with reactive state.

Imagine, You're sitting in a room with a window. You tell Vue: "Whenever someone opens or closes the window, say something."

With watchEffect(), you don’t need to tell Vue which window to watch. Vue just looks at what you're looking at and says: "Ah! You're using window.status, so I’ll watch it for you." That’s the magic of watchEffect(): It automatically tracks what reactive data you're using. And re-runs your function when any of those change.

syntax

```
watchEffect(() => {
  // code using reactive values
})
```

Example : basic example

```
<template>
  <input v-model="name" placeholder="Enter your name" />
  <p>Hello, {{ name }}</p>
</template>

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

const name = ref('')

watchEffect(() => {
  console.log(`The name is now: ${name.value}`)
})
</script>
```

It automatically logs the new value of name every time it changes no need to tell Vue what to track.

Example : Cleanup with onCleanup

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

const count = ref(0)

watchEffect((onCleanup) => {
  const interval = setInterval(() => {
    console.log('count is', count.value)
  }, 1000)

  // clean up old interval before re-running effect
  onCleanup(() => {
    clearInterval(interval)
    console.log('interval cleared')
  })
})
</script>
```

Whenever count changes, the old interval is cleared and a new one is started ensuring no memory leaks.

When to use what

Use watchEffect() when You want a side effect to stay in sync with reactive values. You don't care about old values, just current state. You want auto-dependency tracking.

Use watch() when:

```
You need old vs new values.
You want to watch multiple sources explicitly.
You want to fine-tune execution or debounce/throttle logic.
```