Reactivity in Vue3

Reactivity in Vue is, when the data/state changes, the DOM updates automatically to reflect those changes and no need to manually manipulate the DOM.

Vue uses a reactive system to track changes in data and update the UI accordingly.

ref()

In Vue 3, ref() is a function that creates a reactive reference to a value mostly used for primitive values like strings, numbers, booleans, etc.

```
import { ref } from 'vue'
const count = ref(0)
```

This count is now a reactive reference object. It wraps the value and allows Vue to track changes and trigger DOM updates automatically when the value changes.

- ref() returns an object and the value is wrapped inside a .value property.
- ref() works with all types strings, numbers, booleans, objects, arrays, even functions.
- Mostly used for primitives, ref() is best when used with primitives. For objects, reactive() is usually better.
- Use .value to read or write to the value.
- If a ref changes, the DOM updates automatically (if it's used in a template).

In Vue 2 (Options API), Vue made data reactive by default through data() and getters/setters. But in Vue 3, with the Composition API, we need an explicit way to tell Vue.

"Hey, this value should be reactive and track it and update the DOM when it changes!" This what ref() does.

ref() wraps a value (primitive or complex) inside a reactive object. Vue watches this object for changes and reactively updates anything that depends on it like DOM elements or computed values.

We need ref()

- To make Primitive values reactive and It’s essential for reactive state in the Composition API.

  ```
  const y = 5; // Vue can not tracks y and can't react when it changes
  const x = ref(5) // now Vue tracks x and can react when it changes
  ```

- In the Composition API, we don’t use the data() function anymore, so we use ref() or reactive() to define reactive state.

Example : counter app

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

const message = ref('Hello Vue!')
const count = ref(0)

const user = ref({ name: 'Sourabha', age: 25 })

// Reactivity is maintained, but we must do this:
user.value.name = 'Joshi'

function increment() {
  count.value++
}
</script>

<template>
  <div>
    <p>{{ message }}</p>
    <button @click="increment">Clicked {{ count }} times</button>
  </div>
</template>
```

In template no need to write .value, Vue unwraps it automatically

Example : ref() with all data types

```
<template>
  <div class="p-4">
    <h2>Vue 3 ref() Demo - All Data Types</h2>

    <p>Number: {{ count }}</p>
    <button @click="count++">Increment</button>

    <p>String: {{ message }}</p>
    <input v-model="message" class="border p-1" />

    <p>Boolean: {{ isVisible ? 'Visible' : 'Hidden' }}</p>
    <button @click="isVisible = !isVisible">Toggle Visibility</button>

    <p>Array: {{ items.join(', ') }}</p>
    <button @click="items.push(`Item ${items.length + 1}`)">Add Item</button>

    <p>Object: {{ user.name }} ({{ user.age }} yrs)</p>
    <button @click="user.age++">Age +1</button>

    <p>Function Output: {{ getGreeting() }}</p>

    <p>Null: {{ empty === null ? 'Null Value' : empty }}</p>
    <p>Undefined: {{ something === undefined ? 'Undefined Value' : something }}
</p>
```

```
    <div class="mt-3">
      <input ref="inputRef" placeholder="Focus me on mount" class="border p-1" />
    </div>
  </div>
</template>

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

// Primitive Values
const count = ref(0)
const message = ref('Hello!')
const isVisible = ref(true)

// Array
const items = ref(['Item 1', 'Item 2'])

// Object
const user = ref({ name: 'Sourabha', age: 25 })

// Function
const getGreeting = ref(() => `Welcome, ${user.value.name}!`)

// null and undefined
const empty = ref(null)
const something = ref(undefined)

// DOM Reference
const inputRef = ref(null)

onMounted(() => {
  inputRef.value.focus()
})
</script>

<style scoped>
button {
  margin-right: 10px;
  margin-top: 5px;
}
</style>
```

reactive()

reactive() is a Composition API function in Vue 3 that converts a plain JavaScript object into a reactive object, allowing Vue to track its properties and automatically update the UI when any property changes.

Use of reactive() to:

- Create a reactive store (for small-scale state management)
- Track nested object changes

Need of reactive()

Vue's UI is reactive — it updates automatically when state changes. To enable this:

- We need to make data reactive, so Vue can track and respond to changes.
- reactive() helps us to write code that's more organized, modular, and easier to debug in the Composition API.
- Without it, we have to manually track changes and update the DOM, which is inefficient and error-prone.

Pros

- Deep reactivity — nested properties are reactive
- Cleaner syntax for objects — no .value needed
- Great for managing structured data like forms
- Integrates well with Vue’s reactivity system

Cons

- Doesn’t work with primitives like number, string
- Can't be destructured directly (breaks reactivity)

Example

```
<script setup>
import { reactive } from 'vue';

// Create a reactive object
const form = reactive({
  name: '',
  email: ''
});

function submitForm() {
  alert(`Name: ${form.name}, Email: ${form.email}`);
}
</script>

<template>
  <h2>Contact Form</h2>
  <input v-model="form.name" placeholder="Enter name" />
  <input v-model="form.email" placeholder="Enter email" />
  <button @click="submitForm">Submit</button>
</template>
```

Key points to remember for ref() and reactive()

ref()

- Use for primitive values (number, string, boolean, etc.)

```
const count = ref(0);
```

- Access the value using .value

```
console.log(count.value);
```

- Can also hold objects, but only shallowly reactive

```
const user = ref({ name: 'Joshi' });
user.value.name = 'Sourabha';
```

- Perfect for single values like loading state, toggle flags, counters

```
const isLoading = ref(true);
```

- .value is required everywhere except in `<template>`

```
<p>{{ count }}</p> <!-- no .value needed here -->
```

reactive()

- Use for objects, arrays, and structured data

```
const form = reactive({ name: '', email: '' });
```

- No need for .value — directly access properties

```
form.name = 'Sourabha';
```

- Provides deep reactivity — nested objects are tracked

```
const state = reactive({ user: { name: 'Sachin' } });
```

- Don’t destructure directly — you’ll lose reactivity

```

```

```
const { name } = form; // breaks reactivity
```