**Vue.js**

**1. Directives**

Directives are special tokens in the markup that tell Vue to do something.

**Common Directives:**

* v-bind: Bind attribute (e.g. :href="url")
* v-model: Two-way binding
* v-for: Loop over data
* v-if, v-else-if, v-else: Conditional rendering
* v-show: Toggle visibility via CSS
* v-on: Event binding (@click="doSomething")

**v-bind – Bind an attribute dynamically**

vue

<template>

<a v-bind:href="url">Visit Vue.js</a>

</template>

<script setup>

const url = 'https://vuejs.org';

</script>

Shorthand: :href="url"

**v-model – Two-way data binding (forms)**

vue

<template>

<input v-model="name" placeholder="Enter name" />

<p>Your name is: {{ name }}</p>

</template>

<script setup>

import { ref } from 'vue';

const name = ref('');

</script>

**v-if / v-else-if / v-else – Conditional rendering**

vue

<template>

<p v-if="age < 18">You are a minor.</p>

<p v-else-if="age >= 18 && age < 60">You are an adult.</p>

<p v-else>You are a senior citizen.</p>

</template>

<script setup>

const age = 25;

</script>

**v-show – Toggle visibility (uses CSS display)**

vue

<template>

<button @click="show = !show">Toggle</button>

<p v-show="show">This is conditionally visible.</p>

</template>

<script setup>

import { ref } from 'vue';

const show = ref(true);

</script>

**v-for – Render list**

vue

<template>

<ul>

<li v-for="(fruit, index) in fruits" :key="index">

{{ index + 1 }}. {{ fruit }}

</li>

</ul>

</template>

<script setup>

const fruits = ['Apple', 'Banana', 'Cherry'];

</script>

**v-on – Event binding**

vue

<template>

<button v-on:click="increment">Clicked {{ count }} times</button>

</template>

<script setup>

import { ref } from 'vue';

const count = ref(0);

const increment = () => count.value++;

</script>

Shorthand: @click="increment"

**v-html – Render raw HTML**

vue

<template>

<div v-html="htmlContent"></div>

</template>

<script setup>

const htmlContent = '<strong>This is bold HTML</strong>';

</script>

**Use with caution** to avoid XSS attacks.

**v-text – Set text content**

vue

<template>

<p v-text="message"></p>

</template>

<script setup>

const message = 'Hello from Vue!';

</script>

**v-once – Render only once**

vue

<template>

<p v-once>{{ name }}</p>

</template>

<script setup>

const name = 'Static Name';

</script>

# Vue.js Directives – Full Examples and Variations

## v-bind – Bind dynamic attributes

### Basic Attribute Binding

vue

<a :href="url">Go to Site</a>

<script setup>

const url = 'https://vuejs.org';

</script>

### Binding Multiple Attributes

vue

<img v-bind="{ src: imageUrl, alt: 'Vue Logo' }" />

<script setup>

const imageUrl = 'https://vuejs.org/images/logo.png';

</script>

### Class & Style Binding

vue

<div :class="{ active: isActive }" :style="{ color: color }">

Dynamic class and style

</div>

<script setup>

const isActive = true;

const color = 'blue';

</script>

## v-model – Two-way data binding

### Input Field

vue

<input v-model="username" />

<p>Hi, {{ username }}</p>

<script setup>

import { ref } from 'vue';

const username = ref('');

</script>

### Checkbox & Radio

vue

<input type="checkbox" v-model="checked" />

<p>Checked: {{ checked }}</p>

<script setup>

const checked = ref(false);

</script>

### Select Dropdown

vue

<select v-model="selected">

<option disabled value="">Please select</option>

<option>Vue</option>

<option>React</option>

</select>

<script setup>

const selected = ref('');

</script>

## v-if / v-else-if / v-else – Conditional rendering

### Basic Condition

vue

<p v-if="isLoggedIn">Welcome!</p>

<p v-else>Please log in</p>

<script setup>

const isLoggedIn = ref(true);

</script>

### With Multiple Conditions

vue

<p v-if="role === 'admin'">Admin Panel</p>

<p v-else-if="role === 'user'">User Dashboard</p>

<p v-else>Guest</p>

<script setup>

const role = ref('user');

</script>

## v-show – Show/hide (CSS-based)

### Visibility Toggle

vue

<button @click="show = !show">Toggle</button>

<p v-show="show">Now you see me</p>

<script setup>

const show = ref(true);

</script>

v-show keeps the element in DOM but uses display: none.

## v-for – Looping through arrays or objects

### Array List

vue

<ul>

<li v-for="(item, index) in items" :key="index">

{{ index + 1 }}. {{ item }}

</li>

</ul>

<script setup>

const items = ['Apple', 'Banana', 'Orange'];

</script>

### Object Iteration

vue

<ul>

<li v-for="(value, key) in person" :key="key">

{{ key }}: {{ value }}

</li>

</ul>

<script setup>

const person = { name: 'John', age: 30, city: 'Paris' };

</script>

## v-on – Event handling

### Click Event

vue

<button @click="count++">Count: {{ count }}</button>

<script setup>

const count = ref(0);

</script>

### Mouse Event

vue

<div @mouseover="hover = true" @mouseleave="hover = false">

Hover: {{ hover }}

</div>

<script setup>

const hover = ref(false);

</script>

## v-text – Set plain text

vue

<p v-text="message"></p>

<script setup>

const message = 'Hello from v-text!';

</script>

## v-html – Set raw HTML

vue

<div v-html="rawHtml"></div>

<script setup>

const rawHtml = '<strong>This is bold</strong>';

</script>

Be cautious — can expose XSS vulnerabilities if from user input.

## v-once – Render element once and ignore reactivity

vue

<p v-once>{{ currentTime }}</p>

<script setup>

const currentTime = new Date().toLocaleTimeString();

// Even if time changes, it won’t update

</script>

## Custom Directive (Advanced Use)

### Autofocus Directive

vue

<script>

export default {

directives: {

focus: {

mounted(el) {

el.focus();

}

}

}

}

</script>

<template>

<input v-focus />

</template>

# What is v-bind in Vue.js?

### Definition:

v-bind is a directive used to bind **HTML attributes** to Vue **data** or **computed properties** reactively.

# Syntax

html

<!-- Long form -->

<tag v-bind:attribute="expression" />

<!-- Shorthand -->

<tag :attribute="expression" />

# Why use v-bind?

* Dynamically update attributes (like href, src, class, style)
* Reactively change DOM properties based on data
* Clean and expressive template logic

# Basic Examples

### ****Binding**** href

vue

<a v-bind:href="link">Visit Site</a>

<!-- OR -->

<a :href="link">Visit Site</a>

<script setup>

const link = 'https://vuejs.org';

</script>

### ****Binding**** src ****in**** <img>

vue

<img :src="imageUrl" alt="Logo" />

<script setup>

const imageUrl = 'https://vuejs.org/images/logo.png';

</script>

# Dynamic Class Binding

### ****Object Syntax****

vue

<div :class="{ active: isActive, error: hasError }">Status</div>

<script setup>

const isActive = true;

const hasError = false;

</script>

### ****Array Syntax****

vue

<div :class="[mainClass, conditionalClass]">Multiple Classes</div>

<script setup>

const mainClass = 'card';

const conditionalClass = 'highlight';

</script>

# Dynamic Style Binding

### ****Binding Inline Styles****

vue

<div :style="{ color: textColor, fontSize: size + 'px' }">

Styled Text

</div>

<script setup>

const textColor = 'red';

const size = 18;

</script>

# Binding to Boolean Attributes

### ****Disabling Button Conditionally****

vue

<button :disabled="isSubmitting">Submit</button>

<script setup>

const isSubmitting = true;

</script>

If isSubmitting is true, the button will be disabled.

# Binding to Input Properties

### ****Dynamic Placeholder****

vue

<input :placeholder="placeholderText" />

<script setup>

const placeholderText = 'Type your name';

</script>

# Binding Multiple Attributes

### ****Object Bind****

vue

<img v-bind="{ src: imageUrl, alt: imageAlt }" />

<script setup>

const imageUrl = 'logo.png';

const imageAlt = 'Vue Logo';

</script>

# v-bind with Props (Child Components)

### ****Pass props dynamically****

vue

<!-- Parent.vue -->

<ChildComponent :title="pageTitle" />

<script setup>

const pageTitle = 'Welcome Page';

</script>

vue

<!-- ChildComponent.vue -->

<template>

<h1>{{ title }}</h1>

</template>

<script setup>

defineProps(['title']);

</script>

# Dynamic ARIA or Custom Attributes

### ****Accessible Binding****

vue

<button :aria-label="description">🔍</button>

<script setup>

const description = 'Search button';

</script>

# Use Inside v-for Loops

### ****Bind IDs or Keys****

vue

<ul>

<li v-for="item in items" :key="item.id" :id="'item-' + item.id">

{{ item.name }}

</li>

</ul>

<script setup>

const items = [

{ id: 1, name: 'Vue' },

{ id: 2, name: 'React' }

];

</script>

# v-bind with Dynamic Component

### ****Binding is="..."****

vue

<component :is="currentView" />

<script setup>

const currentView = 'HomePage';

</script>

# Two-Way Data Binding

## What is it?

Data flows **both ways** — from component to template and **back to component**.  
Any change in the input field will **immediately update the variable** in Vue.

### Used with: v-model

### 🔹 Example: Two-Way Binding with v-model

vue

<template>

<input v-model="username" />

<p>Your name is: {{ username }}</p>

</template>

<script setup>

import { ref } from 'vue';

const username = ref('');

</script>

**Explanation:**

* Typing in the input updates username
* Changing username in code updates the input

| **Concept** | **What it does** |
| --- | --- |
| ref() | Creates reactive variable |
| v-model | Two-way binding (input) |
| v-if / v-else | Show/hide parts |
| v-for | Loops over names |
| @click | Handles clicks |
| :class | Adds dynamic classes (like .duplicate) |
| scoped CSS | Style applies only to this component |

**JavaScript vs TypeScript in Vue**

|  |  |  |
| --- | --- | --- |
| **Feature / Functionality** | **JavaScript (JS)** | **TypeScript (TS)** |
| **Language Type** | Dynamically typed | Statically typed (with optional types) |
| **Error Detection** | Only at runtime | At **compile time** — catches bugs early |
| **IDE Support (IntelliSense, etc.)** | Basic | Advanced auto-complete, type hints, docs |
| **Component Safety** | You can pass wrong props & events | Prevents wrong prop types/events at compile time |
| **Scalability** | Harder to maintain large apps | Easier to manage large, complex apps |
| **Refactoring** | Risky | Safe & confident refactoring |
| **Tooling** | Good | **Better tooling** (e.g. auto imports, renaming) |
| **Learning Curve** | Easy | Slightly harder for beginners |
| **Type Checking for Props/Emit** | Manual via props validation | **Typed with interfaces/types** |
| **Code Autocompletion** | Limited | Smarter and context-aware |
| **Vue Composition API** | Fully supported | Fully supported with **type inference** |
| **Setup** | Minimal | Requires lang="ts" and config |

**TypeScript-Specific Functionalities in Vue**

**Typed Props**

ts

defineProps<{

msg: string;

count?: number;

}>();

* Prevents passing wrong data types to components.

**Typed Emits**

ts

const emit = defineEmits<{

(e: 'update-count', count: number): void

}>();

* Ensures only valid events are emitted with correct payloads.

## Why defineEmits is used in Vue (<script setup>)

In Vue 3 (Composition API with <script setup>), defineEmits is used to **declare custom events** that a component emits to its parent.

### Use Case: Child → Parent communication

Let’s say you have a **child component** that needs to notify the **parent** about something — like:

* A button was clicked
* A form was submitted
* A name was deleted

You **emit** an event using defineEmits and the parent **listens** for it with @eventName.

### Example

#### ChildComponent.vue

vue

<script setup lang="ts">

const emit = defineEmits<{

(e: 'notify', message: string): void

}>();

function send() {

emit('notify', 'Child says hello!');

}

</script>

<template>

<button @click="send">Send Message</button>

</template>

#### ParentComponent.vue

vue

<template>

<ChildComponent @notify="handleNotify" />

</template>

<script setup lang="ts">

import ChildComponent from './ChildComponent.vue';

function handleNotify(message: string) {

alert(message); // "Child says hello!"

}

</script>

### Why use defineEmits?

|  |  |
| --- | --- |
| **Reason** | **Benefit** |
| No need to define emits: ['eventName'] in defineComponent() | Cleaner |
| Type safety | You get autocompletion and error-checking |
| Clearer API contract | Parent knows what events are emitted and with what payload |

### Notes

* You must **call** the returned emit() function to trigger the event.
* In traditional components (export default {}), you use this.$emit().
* In <script setup>, defineEmits replaces that.

### What happens if you don’t use defineEmits?

* You can still call emit() if you're not using TypeScript.
* But you'll lose:
  + Type safety
  + Autocompletion
  + Clarity of what events your component emits

### Summary

defineEmits is:

* The Composition API way to declare emitted events.
* Essential for child-to-parent communication.
* Type-safe and more expressive in TypeScript.

**Typed Refs & Reactive**

ts

const count = ref<number>(0);

* You know count.value will always be a number — safer & cleaner.

**Why ref() and reactive()?**

Vue 3 provides **ref()** and **reactive()** to make values reactive in the Composition API.

|  |  |  |
| --- | --- | --- |
| **Method** | **Use Case** | **Works Best With** |
| ref() | For **primitive values** or **single refs** | strings, numbers, booleans |
| reactive() | For **objects**, **arrays**, or **nested state** | objects, deep state trees |

**ref() – Reactive primitive with reactivity**

ts

const count = ref(0); // count is a Ref<number>

* ref(0) makes a reactive wrapper around the value 0.
* You access/update it via count.value.

ref is best when:

* You are tracking a number, string, boolean, or date.
* You want to type it, like: ref<string>('hello')

**reactive() – Reactive object**

ts

const user = reactive({

name: 'Rose',

age: 25

});

* reactive() deeply tracks changes in objects.
* You access properties directly: user.name, user.age.

reactive is best when:

* You’re working with structured data or many fields.
* You want to avoid .value usage.

**Why we use ref() in <script setup lang="ts">?**

Vue automatically infers types if you write:

ts

const name = ref<string>('Rose');

But you also get:

* **Reactivity**
* **Type safety**
* **Auto-completion**
* Integration with Vue templates

**Example with explanation**

ts

const newName = ref('');

* ref(''): Makes a reactive string.
* Used for v-model in <input v-model="newName">
* TypeScript infers it's Ref<string>

ts

const names = ref<string[]>([]);

* Reactive array
* Tracks all entered names

ts

const looser = ref<string | null>(null);

* Starts empty
* Can be assigned a name later

**Why .value is needed?**

In the <script setup>, you need .value to access a ref:

ts

if (names.value.includes(name)) { ... }

But in the **template**, Vue automatically unwraps .value for you:

html

<input v-model="newName" />

No need to write newName.value in the template.

**Summary**

|  |  |
| --- | --- |
| **Concept** | **Purpose** |
| ref() | Makes a single value reactive |
| reactive() | Makes an object reactive |
| .value | Required in <script> to access ref |
| No .value | In <template>, Vue unwraps it |

**Typed Functions**

ts

function greet(name: string): string {

return `Hello, ${name}`;

}

* Makes code self-documenting, and catches logic errors earlier.

**IDE Support (VSCode)**

* With TypeScript, Vue files show **hover docs**, **error underlines**, **autocomplete**, and **type inference**.

**Reusable Types/Interfaces**

ts

interface User {

id: number;

name: string;

}

const users = ref<User[]>([]);

* Ensures consistency across large applications.

**JavaScript in Vue**

**Pros:**

* Easier to start for beginners.
* Less setup.
* More concise (no types).

**Cons:**

* No type checking.
* Prone to runtime errors.
* Harder to debug in large apps.

**TypeScript in Vue**

**Pros:**

* **Catches bugs early**.
* Enforces structure and consistency.
* IDE gives **smart autocomplete and documentation**.
* Great for **team collaboration and long-term projects**.

**Cons:**

* Learning curve for new developers.
* Slightly more boilerplate.
* Needs proper config (vue-tsc, lang="ts" in <script>).

**Final Summary**

|  |  |  |
| --- | --- | --- |
|  | **JavaScript** | **TypeScript** |
| Type | Dynamic | Static |
| Best for | Beginners, small apps | Professionals, medium-large apps |
| Errors | Caught at runtime | Caught during development |
| Vue Use | Works well | Works **better with Composition API** |

**1. Using TypeScript in Vue (<script lang="ts">)**

|  |  |
| --- | --- |
| **Feature** | **What it does** |
| lang="ts" | Enables TypeScript for better tooling, safety, and catching bugs early. |
| TypeScript | Adds strict types (like string, number[]) so that your code is more predictable. |

**Example:**

ts

const names = ref<string[]>([]); // Enforces that only a string array can be stored

**2. Using defineComponent**

ts

export default defineComponent({

name: 'App',

components: { WhoFails }

});

|  |  |
| --- | --- |
| **Key** | **Purpose** |
| defineComponent() | Helps Vue and TypeScript understand your component better. |
| name | Useful in debugging tools (like Vue Devtools). |
| components | Registers any child component (e.g., WhoFails.vue) you want to use in <template>. |

**3. Component Import/Export**

ts

import WhoFails from './components/WhoFails.vue';

* Imports another Vue component you wrote.
* That component holds the real logic for name handling, looser selection, etc.

You can then use it as <WhoFails /> in the template.

**4. Reactive Variables with ref()**

ts

const newName = ref('');

const names = ref<string[]>([]);

const looser = ref<string | null>(null);

const errorMsg = ref('');

const showResult = ref(false);

|  |  |
| --- | --- |
| **Variable** | **Role** |
| newName | Holds the value typed in the input field. |
| names | Stores all added names in a list (string array). |
| looser | Stores the randomly selected looser name. |
| errorMsg | Stores validation errors like duplicates or empty names. |
| showResult | Toggles between input form and result screen. |

**TypeScript Enforces:**

* names must be a string array (string[])
* looser can either be string or null

|  |  |
| --- | --- |
| **What it does** | **Why it's important** |
| Removes whitespace using trim() | Prevents adding just spaces |
| Checks if empty | Prevents adding empty name |
| Checks if already exists | Prevents duplicate entries |
| Updates names | Pushes valid name into the list |
| Clears input | So user can type next name |

**6. Removing a Name**

ts

function removeName(name: string) {

names.value = names.value.filter(n => n !== name);

if (looser.value === name) looser.value = null;

}

* Clicking a name will remove it from the list.
* If it was the selected looser, clears the result.

**7. Highlighting Duplicate Names**

ts

function isDuplicate(name: string) {

return names.value.filter(n => n === name).length > 1;

}

This function:

* Checks if the name appears more than once.
* Returns true to highlight the name (CSS uses duplicate class).

**8. Random Looser Selection**

ts

function pickLooser() {

const random = names.value[Math.floor(Math.random() \* names.value.length)];

looser.value = random;

showResult.value = true;

}

|  |  |
| --- | --- |
| **Feature** | **Purpose** |
| Math.random() | Randomly picks one name from the list |
| Updates looser | Displays the selected name |
| Shows result screen | Hides input, shows result screen |

**9. Pick Another Looser**

ts

function pickAnother() {

if (names.value.length <= 1) return;

let newRandom;

do {

newRandom = names.value[Math.floor(Math.random() \* names.value.length)];

} while (newRandom === looser.value);

looser.value = newRandom;

}

|  |  |
| --- | --- |
| **Feature** | **What it does** |
| Checks list length | Only allows if more than 1 person is present |
| Loops until new name | Avoids repeating same looser |
| Updates looser | Displays new random looser |

**10. Reset Button**

ts

function reset() {

names.value = [];

looser.value = null;

errorMsg.value = '';

showResult.value = false;

}

* Clears everything and returns to the initial screen.
* Useful for restarting the game.

**In the Template (<template>)**

vue

<WhoFails />

* This calls the custom component where all above logic lives.

**What TypeScript Adds Compared to JavaScript**

|  |  |  |
| --- | --- | --- |
| **Feature** | **JavaScript** | **TypeScript** |
| Type Safety | ❌ | ✅ (e.g., ref<string>()) |
| Editor Support | Basic | Advanced (autocomplete, errors) |
| Runtime Checks | ❌ | ✅ with compile-time checks |
| Refactoring | Manual | Safer & easier |
| Catching Bugs Early | ❌ | ✅ |

**Vue 3 provide and inject**

## What Are provide and inject?

* **provide** makes a value available to all **descendant components**.
* **inject** is used by child (or grandchild, etc.) components to access that value.

They help **avoid prop drilling** (passing data manually through multiple layers of props).

## When to Use

Use provide/inject when:

* A **parent component** wants to share **data or methods** deeply with nested components.
* You want **dependency injection** (e.g., for services or configs).
* You want **loose coupling** between components.

## Syntax

### provide (in parent):

#### Options API

js

export default {

provide() {

return {

keyName: value,

}

}

}

#### Composition API / <script setup>

ts

import { provide, ref } from 'vue'

const message = ref('Hello from parent')

provide('keyName', message)

### inject (in child):

#### Options API

js

export default {

inject: ['keyName']

}

#### Composition API / <script setup>

ts

import { inject } from 'vue'

const message = inject('keyName')

## Data Types You Can Provide

* **Primitive values** (string, number, etc.)
* **Objects**, **arrays**
* **Refs** (reactive values using ref() or reactive())

## Reactivity Notes

|  |  |  |
| --- | --- | --- |
| **Provided Value** | **Reactive?** | **Notes** |
| ref() | ✅ Yes | Best choice for reactive sharing |
| reactive() | ✅ Yes | Reactive object; deeply tracks changes |
| Primitives | ❌ No | Use ref() if you want to track updates |

## Prevent Child from Changing Parent

Use readonly():

ts

import { provide, readonly, ref } from 'vue'

const parentValue = ref('data')

provide('keyName', readonly(parentValue))

This makes injected data **read-only** in the child component.

### Parent (App.vue)

ts

<script setup>

import { provide, ref } from 'vue'

const theme = ref('dark')

provide('theme', theme)

</script>

<template>

<ChildComponent />

</template>

### Child (deeply nested)

ts

<script setup>

import { inject } from 'vue'

const theme = inject('theme')

</script>

<template>

<p>Current theme: {{ theme }}</p>

</template>

## Summary Table

|  |  |  |
| --- | --- | --- |
| **Feature** | **provide** | **inject** |
| Direction | Downward (parent → child) | Receives from ancestor |
| Used in | Parent component | Child or descendant component |
| Usage | provide(key, value) | inject(key) |
| Reactivity | Use ref or reactive for it | Automatically reactive if ref used |
| Default value | Yes, in inject | inject('key', 'default') |
| Scoped? | No – shared across descendants | Can be used anywhere in tree |

## Real-World Use Cases

* Global configuration (e.g., theme, language)
* Sharing a service (e.g., event bus, API handler)
* Form validation state
* Dynamic layout or sizing data
* Shared composables (like a useStore() pattern)

**Slots**

In Vue.js, **slots** are a powerful feature that allow you to **inject content** into a component from the **outside**, providing flexible component composition. They are useful when you want to design components that are **reusable and customizable**.

**TYPES OF SLOTS IN VUE**

**Default Slot**

Used when you have only one slot or unnamed content.

**Child Component (BaseCard.vue)**

vue

<template>

<div class="card">

<slot></slot> <!-- default slot -->

</div>

</template>

**Parent Component**

vue

<BaseCard>

<p>This is inside the slot!</p>

</BaseCard>

**Named Slots**

Used when you want multiple content sections in a component.

**Child Component (BaseLayout.vue)**

vue

<template>

<header><slot name="header"></slot></header>

<main><slot></slot></main>

<footer><slot name="footer"></slot></footer>

</template>

**Parent Component**

vue

<BaseLayout>

<template #header>

<h1>Page Title</h1>

</template>

<p>This is the main content</p>

<template #footer>

<p>&copy; 2025 MyCompany</p>

</template>

</BaseLayout>

**Scoped Slots**

Used when the **child component passes data back** to the parent via slot.

**Child Component (UserList.vue)**

vue

<template>

<ul>

<li v-for="user in users" :key="user.id">

<slot :user="user">

{{ user.name }} <!-- fallback -->

</slot>

</li>

</ul>

</template>

<script setup>

const users = [

{ id: 1, name: 'Alice' },

{ id: 2, name: 'Bob' },

];

</script>

**Parent Component**

vue

<UserList>

<template #default="{ user }">

<strong>{{ user.name }}</strong> (User ID: {{ user.id }})

</template>

</UserList>

**Summary**

|  |  |  |
| --- | --- | --- |
| **Type** | **Syntax** | **Purpose** |
| Default Slot | <slot></slot> | For simple content insertion |
| Named Slot | <slot name="slotName"></slot> | For multiple named areas |
| Scoped Slot | <slot :prop="value"> | Pass data from child to parent |

**Form in vue**

Creating a **form in Vue 3** involves handling form inputs using reactive data and managing submissions, validations, etc. Here's a **complete step-by-step guide** on how to build and handle a form in **Vue 3** using both the **Options API** and **Composition API (with <script setup>)**.

|  |  |
| --- | --- |
| **Working Feature** | **💡 Tip** |
| Form with reactive state | Good use of reactive() |
| v-model form binding | Correctly done |
| Preventing default submit | @submit.prevent is correct |
| Provide/Inject | provide() is good; check inject() in child |
| Slots (named + scoped) | Perfect usage! |

**Vue 3 Event Modifiers – Full List with Explanation**

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| .stop | Calls event.stopPropagation() – **stops the event from bubbling** up to parent elements. |
| .prevent | Calls event.preventDefault() – **prevents the default browser behavior**, like form submission or anchor link jumping. |
| .capture | Adds the event listener in **capture mode** instead of bubbling phase. |
| .self | Only triggers the event handler if the event **originated on the element itself**, not a child. |
| .once | Only triggers the event **once** – listener is automatically removed after first trigger. |
| .passive | Sets { passive: true } on the event listener – improves scroll performance, but **you cannot call preventDefault()**. |
| .native *(Vue 2)* | Was used to bind native events to components; not used in Vue 3 anymore. Use v-on="$attrs" or emits. |

**vee-validate**

**1. What is vee-validate?**

vee-validate is a form validation library for Vue.js. It allows you to:

* Validate forms declaratively or programmatically.
* Integrate with form libraries and custom components.
* Use built-in or custom validation rules.
* Easily integrate with Composition API or <script setup>.

**2. Installation**

bash

npm install vee-validate yup

* vee-validate: core validation framework.
* yup: schema-based validator (optional but commonly used).

**3. Basic Setup (Composition API / <script setup>)**

vue

<script setup>

import { useForm, useField } from 'vee-validate'

import \* as yup from 'yup'

const { handleSubmit } = useForm({

validationSchema: yup.object({

name: yup.string().required(),

email: yup.string().email().required(),

}),

})

const { value: name, errorMessage: nameError } = useField('name')

const { value: email, errorMessage: emailError } = useField('email')

const onSubmit = handleSubmit(values => {

console.log(values)

})

</script>

<template>

<form @submit.prevent="onSubmit">

<input v-model="name" placeholder="Name" />

<span>{{ nameError }}</span>

<input v-model="email" placeholder="Email" />

<span>{{ emailError }}</span>

<button type="submit">Submit</button>

</form>

</template>

**4. Core Concepts**

|  |  |
| --- | --- |
| **Concept** | **Description** |
| useForm() | Creates a form context |
| useField(name) | Creates a field that syncs with the form |
| handleSubmit() | Wraps your submit handler and validates the form |
| yup | Optional schema validator used to declare field rules |
| errorMessage | Reactive error message for each field |

**5. Common Validation Rules (with yup)**

ts

yup.string().required()

yup.string().email()

yup.string().min(5)

yup.string().max(20)

yup.number().required().positive().integer()

yup.date().required()

yup.boolean().oneOf([true])

yup.string().matches(/^[a-zA-Z]+$/, 'Only letters allowed')

**6. Custom Validation Rules**

ts

import { defineRule } from 'vee-validate'

defineRule('starts\_with\_m', value => {

if (!value.startsWith('M')) return 'Must start with M'

return true

})

Use it in useField:

ts

const { value, errorMessage } = useField('username', 'starts\_with\_m')

**7. Programmatic Validation & Reset**

ts

const { handleSubmit, resetForm, validate } = useForm()

// Validate manually

const result = await validate()

// Reset form

resetForm()

**8. Validate on Blur or Input**

ts

useField('name', yup.string().required(), {

validateOnValueUpdate: true, // validate on input

validateOnMount: true, // validate immediately

})

**9. Handing Nested Forms / Arrays**

ts

yup.object({

user: yup.object({

name: yup.string().required(),

age: yup.number().min(18)

}),

hobbies: yup.array().of(yup.string().required())

})

For fields: 'user.name', 'hobbies[0]'

**10. FieldArray Support (e.g., Dynamic Inputs)**

vue

<script setup>

import { useFieldArray, useForm } from 'vee-validate'

import \* as yup from 'yup'

const { handleSubmit, values } = useForm({

initialValues: {

friends: ['']

},

validationSchema: yup.object({

friends: yup.array().of(yup.string().required())

})

})

const { fields, remove, push } = useFieldArray('friends')

const onSubmit = handleSubmit(values => console.log(values))

</script>

<template>

<form @submit.prevent="onSubmit">

<div v-for="(field, idx) in fields" :key="field.key">

<input v-model="values.friends[idx]" />

<button @click="remove(idx)">Remove</button>

</div>

<button @click="push('')">Add Friend</button>

<button type="submit">Submit</button>

</form>

</template>

**11. Styling Error Messages**

vue

<span v-if="nameError" class="error">{{ nameError }}</span>

<style scoped>

.error {

color: red;

font-size: 0.9em;

}

</style>

**12. Unit Testing with vee-validate**

* You can test useForm, useField values.
* Use flushPromises() when testing async validation.

**13. FormContext + Nested Components (using Form, Field components)**

Alternatively, vee-validate provides <Form>, <Field>, and <ErrorMessage> components:

vue

<template>

<Form @submit="onSubmit">

<Field name="email" as="input" type="email" />

<ErrorMessage name="email" />

<button type="submit">Submit</button>

</Form>

</template>

ts

import { Form, Field, ErrorMessage } from 'vee-validate'

**14. Vee-Validate vs Native Validation**

|  |  |  |
| --- | --- | --- |
| **Feature** | **vee-validate** | **Native HTML5 validation** |
| Custom rules | ✅ Yes | ❌ Limited |
| Schema validation | ✅ Yup / Zod / custom | ❌ No |
| Reactive error handling | ✅ Reactive refs | ❌ Not reactive |
| Styling/UX control | ✅ Total control | ❌ Limited |

**Summary Cheatsheet**

|  |  |
| --- | --- |
| **Function / Component** | **Description** |
| useForm() | Form context |
| useField(name) | Single field |
| useFieldArray(name) | Dynamic array |
| handleSubmit(callback) | Validated submit handler |
| validate() | Trigger full validation |
| <Form> | Declarative form wrapper |
| <Field> | Declarative input binding |
| <ErrorMessage> | Auto error message display |

**Why do we import useForm and useField?**

These are **Composition API helpers** from VeeValidate.

**1. useForm()**

This sets up the **form context** — it allows you to:

* Handle form submission (handleSubmit)
* Access all form **errors**
* Track form state (e.g., dirty, touched, valid)

Example:

ts

const { handleSubmit, errors } = useForm()

Here:

* handleSubmit(...) wraps your submission handler
* errors gives you error messages for each field

**2. useField()**

This connects an individual **form field** to validation rules and state tracking.

Example:

ts

const { value: emailValue } = useField('email', email)

* "email" = the name of the field
* email = a validation rule (imported from @vee-validate/rules)
* value: emailValue = the reactive v-model for that field

Now you can bind it:

html

<input v-model="emailValue" />

**TL;DR: Why Use These?**

|  |  |
| --- | --- |
| **Function** | **Why it's used** |
| useForm() | Creates the form context (submission, error tracking, validation state) |
| useField() | Registers each input field (value binding, validation, error display) |

**Visual Overview**

ts

// Setup

const { handleSubmit, errors } = useForm()

const { value: name } = useField('name', required)

// Usage

<form @submit.prevent="handleSubmit(onSubmit)">

<input v-model="name" />

<span>{{ errors.name }}</span>

</form>

**Without VeeValidate**

You'd have to manually:

* Watch each input field
* Validate it using if-else logic or regex
* Show error messages
* Manage when to show them (e.g., on blur, on submit)

This becomes **harder** as the form grows.

**Unit Testing in Vue.js**

Unit testing in Vue.js allows you to test **individual components, methods, and logic** to ensure they work as expected in isolation.

**Tools You Need for Vue Unit Testing**

The most common stack includes:

|  |  |
| --- | --- |
| **Tool** | **Purpose** |
| vitest | Test runner (like Jest, fast + modern) |
| @vue/test-utils | Vue testing utilities |
| happy-dom or jsdom | DOM environment for test simulation |

Install dependencies:

bash

npm install --save-dev vitest @vue/test-utils happy-dom

Then configure vite.config.ts:

ts

// vite.config.ts

import { defineConfig } from 'vite'

import vue from '@vitejs/plugin-vue'

export default defineConfig({

plugins: [vue()],

test: {

environment: 'happy-dom',

},

})

**Basic Unit Test Example (Counter Component)**

**Counter.vue**

vue

<template>

<div>

<p>Count: {{ count }}</p>

<button @click="increment">+</button>

</div>

</template>

<script setup lang="ts">

import { ref } from 'vue'

const count = ref(0)

function increment() {

count.value++

}

</script>

**Counter.test.ts**

ts

import { mount } from '@vue/test-utils'

import { describe, it, expect } from 'vitest'

import Counter from './Counter.vue'

describe('Counter.vue', () => {

it('renders count', () => {

const wrapper = mount(Counter)

expect(wrapper.text()).toContain('Count: 0')

})

it('increments count on button click', async () => {

const wrapper = mount(Counter)

await wrapper.find('button').trigger('click')

expect(wrapper.text()).toContain('Count: 1')

})

})

**What You Can Test**

| **What to Test** | **How? (Method)** |
| --- | --- |
| Component rendering | mount() and check DOM |
| Props | mount(Component, { props }) |
| Events emitted | wrapper.emitted() |
| Methods | Call method and assert |
| Slots | Provide slot content |
| Form behavior | Set input values + trigger |

**Best Practices**

* ✅ Test one thing per test case.
* ✅ Use mount() for full render, shallowMount() to avoid child components.
* ✅ Keep test logic separate from component logic.
* ✅ Mock external dependencies if needed.

**Run Tests**

In package.json, add:

json

"scripts": {

"test": "vitest"

}

Then run:

bash

npm run test

**Steps:**

**1: install unit test**  
**npm install --save-dev vitest @vue/test-utils happy-dom**

This allows you to mount Vue components, simulate interaction, and assert behavior — all without opening a real browser.

**Why?**

* vitest: The actual testing framework (like Jest, but faster and works natively with Vite).
* @vue/test-utils: Official utility for testing Vue components (used to mount and inspect them).
* happy-dom: A lightweight DOM implementation to simulate the browser in Node.js — used as a testing environment (instead of JSDOM).

**npm install -D vitest**

* npm install — tells npm to install a package.
* -D (or --save-dev) — means you're installing the package as a **development dependency**.
* vitest — is the actual testing framework you're installing.

**So What Is a Development Dependency?**

* These are **only needed during development**, **not** when your app runs in production.
* Examples: testing tools (vitest), linters (eslint), compilers (typescript), etc.

They are listed in your package.json under:

json

"devDependencies": {

"vitest": "^3.2.4"

}

**Why Use -D for Vitest?**

Because:

* You don't need **Vitest** to be bundled with your app when deploying.
* It's only used to **write and run unit tests during development**.

**What Happens If You Don't Use -D?**

bash

npm install vitest

* The package goes into regular "dependencies" instead of "devDependencies".
* This makes your production bundle unnecessarily larger.
* It may cause issues with deployment tools that expect "dependencies" to only include runtime code.

|  |  |
| --- | --- |
| **Command** | **Meaning** |
| npm install vitest | Adds to **production dependencies** (not recommended here) |
| npm install -D vitest | Adds to **devDependencies** ✅ (correct for testing tools) |

**2.unit test for components**

**Why?**

\*.spec.ts files are conventionally used for test files.

These contain your test cases using describe, it, expect, and mount(...).

**3.edit vitest.config.ts and add test there and add scripts test oin package.json**

**1. Add this to vite.config.ts:**

ts

test: {

environment: 'happy-dom'

}

**Why?**

* Tells Vitest to use happy-dom instead of default environment (node).
* Makes Vue DOM APIs work correctly.

**2. Add this to package.json:**

json

"scripts": {

"test": "vitest"

}

**Why?**

* So you can run tests using: npm run test
* It just runs vitest, which looks for .spec.ts files and executes them.

**4.and create a vitest.setup.ts // optional here used fontawesome**

// vitest.setup.ts

import { config } from '@vue/test-utils'

config.global.stubs = {

'font-awesome-icon': {

template: '<span class="mock-icon" />'

}

}

**Why?**

* Some components (like <font-awesome-icon>) are external and not registered in your test environment.
* Without stubbing, Vitest will throw warnings or errors like:

markdown

[Vue warn]: Failed to resolve component: font-awesome-icon

Stubbing replaces the original component with a dummy version so your test can continue.

**5.and add setupFiles: './vitest.setup.ts' inside the vitest.config.ts**

**vite.config.ts**

ts

setupFiles: './vitest.setup.ts'

**Why?**

* This tells Vitest to **automatically run this setup file before all tests**.
* Ensures all global mocks, stubs, or configurations are applied project-wide — no need to repeat in every .spec.ts.

**6.and run**

npm run test

# OR

npx vitest run

 npm run test: Uses your package.json script

 npx vitest run: Directly invokes Vitest to run once and exit

 You’ll see which tests pass/fail, and the output of assertions

|  |  |  |
| --- | --- | --- |
| **Step** | **What You Do** | **Why It’s Needed** |
| 1 | Install packages | Get test tools: Vitest, Vue utils, DOM support |
| 2 | Write test files | Actually test your components |
| 3 | Configure Vitest in Vite config | Tell Vitest how to run tests in a Vue environment |
| 4 | Create vitest.setup.ts | Setup global mocks for components like font-awesome |
| 5 | Register setup in config | Auto-apply setup before each test run |
| 6 | Run tests | See if components behave as expected |

**What is mount()?**

mount() is a function from @vue/test-utils.

It is used to:

Render your Vue component in a test environment,  
just like how Vue would render it in the browser.

**Where does it come from?**

ts

import { mount } from '@vue/test-utils'

This gives you access to the mount() function, which allows you to load and interact with Vue components **in your tests**.

**Basic Example:**

ts

import { mount } from '@vue/test-utils'

import Cards from '../Cards.vue'

const wrapper = mount(Cards)

|  |  |
| --- | --- |
| **Code** | **Meaning** |
| mount(Cards) | Renders the Cards.vue component in memory (not in real browser) |
| wrapper | Now holds the rendered component — you can inspect its HTML, data, events, etc. |

**What Can You Do With mount()?**

Once you mount() a component, you can:

|  |  |  |
| --- | --- | --- |
| **🧪 Action** | **Example Code** | **What it does** |
| Find elements | wrapper.find('.green-card') | Finds one element by class |
| Find multiple elements | wrapper.findAll('.green-card') | Finds all matching elements |
| Access text | wrapper.text() | Gets the full visible text |
| Get HTML | wrapper.html() | Returns the rendered HTML |
| Interact (click, input) | wrapper.find('button').trigger('click') | Simulates a user clicking |
| Access props, emits, etc. | wrapper.emitted() | Check events like @click |

**In Your Case**

ts

const wrapper = mount(Cards)

const cards = wrapper.findAll('.green-card')

expect(cards).toHaveLength(3)

This means:

1. Create a test version of the Cards.vue component.
2. Find all elements with class .green-card.
3. Expect there to be **3 cards rendered**.

**Analogy**

* Think of mount() like putting your Vue component **into a mini test browser window**, where you can see what it looks like and how it behaves — but only in code, not visually.

**Important Terms & Methods**

**1. mount()**

* Used to fully render a component (with all its child components).

ts

const wrapper = mount(MyComponent)

**2. shallowMount()**

* Same as mount(), but doesn’t render child components (uses stubs).

ts

const wrapper = shallowMount(MyComponent)

**3. wrapper**

* The result of mount() or shallowMount() is called a wrapper.
* It represents the component's instance.
* You use wrapper to query elements, trigger events, check props, etc.

**4. wrapper.find(selector)**

* Finds the first matching element/component in the rendered output.

ts

const button = wrapper.find('button')

**5. wrapper.findAll(selector)**

* Finds all matching elements/components.

ts

const items = wrapper.findAll('.item')

**6. wrapper.text()**

* Returns the text content of the component.

ts

expect(wrapper.text()).toContain('Welcome')

**7. wrapper.html()**

* Returns the full HTML content as a string.

**8. wrapper.classes()**

* Returns an array of all classes on the root element.

ts

expect(wrapper.classes()).toContain('active')

**9. wrapper.props()**

* Gets the props passed to a component.

ts

expect(wrapper.props().title).toBe('My Title')

**10. wrapper.emitted()**

* Checks if an event was emitted by the component.

ts

wrapper.vm.$emit('custom-event')

expect(wrapper.emitted()).toHaveProperty('custom-event')

**11. trigger(event)**

* Simulates a user interaction like click, input, submit.

ts

await wrapper.find('button').trigger('click')

**Example Test File**

ts

import { describe, it, expect } from 'vitest'

import { mount } from '@vue/test-utils'

import MyButton from '../MyButton.vue'

describe('MyButton.vue', () => {

it('renders the label', () => {

const wrapper = mount(MyButton, {

props: { label: 'Click me' }

})

expect(wrapper.text()).toContain('Click me')

})

it('emits a click event when clicked', async () => {

const wrapper = mount(MyButton)

await wrapper.trigger('click')

expect(wrapper.emitted()).toHaveProperty('click')

})

})

**Summary Table**

|  |  |
| --- | --- |
| **Method** | **Description** |
| mount() | Renders the full component |
| shallowMount() | Renders only the component, stubs children |
| find() | Finds one element |
| findAll() | Finds all elements |
| text() | Returns text content |
| html() | Returns HTML string |
| classes() | Returns classes array |
| props() | Gets props |
| emitted() | Gets emitted events |
| trigger() | Triggers DOM event |

**Best Practices**

* Test UI **structure**: number of elements, text, classes.
* Test **functionality**: emits, computed, methods, DOM events.
* Use shallowMount if you only want to test one component in isolation.
* Use mount if you need full child component rendering.

**Core Testing Concepts**

**1. mount(component)**

* Loads and renders a component.

ts

const wrapper = mount(MyComponent)

**2. wrapper**

* Object returned by mount().
* Lets you access and control the component.

ts

const wrapper = mount(MyComponent)

wrapper.text() // get text

wrapper.html() // get rendered HTML

wrapper.find('h1') // find one element

wrapper.findAll('.item') // find all elements with class

**3. describe()**

* Groups related test cases.

ts

describe('MyComponent', () => {

// test cases here

})

**4. it() / test()**

* Defines a single test case.

ts

it('does something', () => {

// your test logic

})

**5. expect()**

* Asserts expected results (like "I expect this to be true").

ts

expect(wrapper.text()).toContain('Hello')

expect(wrapper.findAll('.item')).toHaveLength(3)

**6. find() & findAll()**

* find() returns a single element.
* findAll() returns an array of elements.

ts

wrapper.find('button')

wrapper.findAll('.green-card')

**7. trigger()**

* Simulates user events like clicks or inputs.

ts

await wrapper.find('button').trigger('click')

**8. setValue() (For Inputs)**

ts

await wrapper.find('input').setValue('Mary')

**9. props()**

* Access props passed to the component.

ts

expect(wrapper.props('title')).toBe('Hello')

**10. emitted()**

* Check if an event was emitted (like @click).

ts

await wrapper.find('button').trigger('click')

expect(wrapper.emitted()).toHaveProperty('my-event')

**11. mock**

* Used to **fake components or functions** (like icons or APIs).

ts

vi.mock('api-service', () => ({

fetchData: () => Promise.resolve({ name: 'John' })

}))

**12. stubs**

* Replace components like font-awesome-icon in tests.

ts

import { config } from '@vue/test-utils'

config.global.stubs = {

'font-awesome-icon': {

template: '<span class="mock-icon" />'

}

}

**Common Use Cases**

|  |  |
| --- | --- |
| **Feature** | **Example** |
| Test input field | setValue() + expect() |
| Test event emit | trigger() + emitted() |
| Test props | props() |
| Test dynamic list | findAll() |
| Test v-if / v-for | wrapper.text(), find() |
| Test conditional rendering | wrapper.find().exists() |

**Tips**

* Use mount() to test a full component.
* Use shallowMount() (if available) to ignore child components.
* Always use await when triggering DOM events like click or input.
* Group tests using describe() to organize clearly.
* Use setupFiles in vitest.config.ts for global config like stubs.

**Sample Assertion Cheatsheet**

ts

expect(wrapper.text()).toContain('Hello')

expect(wrapper.findAll('li')).toHaveLength(3)

expect(wrapper.html()).toMatchSnapshot()

expect(wrapper.emitted()).toHaveProperty('submit')

**beforeEach() and afterEach() in Unit Tests**

These are special functions provided by testing frameworks like **Vitest** (or Jest) to run **setup or cleanup code** **before or after each test case**.

**Why use them?**

* To **reduce code duplication**.
* To set up **shared variables** or **initialize components**.
* To **clean up** after each test to prevent side effects between tests.

**Syntax**

ts

import { beforeEach, afterEach, describe, it, expect } from 'vitest'

import { mount } from '@vue/test-utils'

import MyComponent from '../MyComponent.vue'

let wrapper

beforeEach(() => {

// runs before each test

wrapper = mount(MyComponent, {

props: { title: 'Hello' }

})

})

afterEach(() => {

// runs after each test

wrapper.unmount()

})

describe('MyComponent', () => {

it('renders title', () => {

expect(wrapper.text()).toContain('Hello')

})

it('has correct class', () => {

expect(wrapper.classes()).toContain('my-component')

})

})

**How It Works**

|  |  |  |
| --- | --- | --- |
| **Hook** | **When it runs** | **Purpose** |
| beforeEach | Before **every it()** | Setup code (mount component, etc.) |
| afterEach | After **every it()** | Cleanup code (e.g., unmount) |

**Real Example**

Let's say you're testing a login form component:

ts

let wrapper

beforeEach(() => {

wrapper = mount(LoginForm)

})

afterEach(() => {

wrapper.unmount()

})

describe('LoginForm.vue', () => {

it('has an email input', () => {

expect(wrapper.find('input[type="email"]').exists()).toBe(true)

})

it('emits submit when form is submitted', async () => {

await wrapper.find('form').trigger('submit.prevent')

expect(wrapper.emitted()).toHaveProperty('submit')

})

})

**Without beforeEach (bad repetition):**

ts

it('test 1', () => {

const wrapper = mount(MyComponent)

// ...

})

it('test 2', () => {

const wrapper = mount(MyComponent)

// ...

})

**What Are Lifecycle Hooks?**

Lifecycle hooks are **built-in functions** in Vue that let you **run code at specific stages** of a component's life:

* **Creation**
* **Mounting**
* **Updating**
* **Unmounting**

**Lifecycle Phases Overview**

|  |  |  |
| --- | --- | --- |
| **Phase** | **Hook Names** | **Description** |
| **Creation** | beforeCreate, created | Called before and after the component is created (data/reactivity setup). |
| **Mounting** | beforeMount, mounted | Called before and after the component is added to the DOM. |
| **Updating** | beforeUpdate, updated | Called before and after the DOM is re-rendered (due to reactive data change). |
| **Unmounting** | beforeUnmount, unmounted | Called before and after a component is removed from the DOM. |
| **Error** | errorCaptured, onErrorCaptured | Handle errors from child components. |

**Lifecycle Hook Order**

text

beforeCreate → created → beforeMount → mounted

→ (reactive data changes) → beforeUpdate → updated

→ (when removed) → beforeUnmount → unmounted

**Options API Syntax**

js

export default {

data() {

return { count: 0 };

},

beforeCreate() {

console.log('beforeCreate');

},

created() {

console.log('created');

},

beforeMount() {

console.log('beforeMount');

},

mounted() {

console.log('mounted');

},

beforeUpdate() {

console.log('beforeUpdate');

},

updated() {

console.log('updated');

},

beforeUnmount() {

console.log('beforeUnmount');

},

unmounted() {

console.log('unmounted');

}

}

**Composition API Syntax (with <script setup>)**

ts

<script setup>

import { onBeforeMount, onMounted, onBeforeUpdate, onUpdated, onBeforeUnmount, onUnmounted } from 'vue';

onBeforeMount(() => {

console.log('beforeMount');

});

onMounted(() => {

console.log('mounted');

});

onBeforeUpdate(() => {

console.log('beforeUpdate');

});

onUpdated(() => {

console.log('updated');

});

onBeforeUnmount(() => {

console.log('beforeUnmount');

});

onUnmounted(() => {

console.log('unmounted');

});

</script>

onCreated() and onBeforeCreate() do **not exist** in Composition API. Use setup function directly instead.

**Lifecycle Diagram**

markdown

┌─────────────┐

│ beforeCreate│

└──────┬──────┘

↓

┌──────────┐

│ created │

└────┬─────┘

↓

┌────────────┐

│ beforeMount│

└──────┬─────┘

↓

┌──────────┐

│ mounted │

└────┬─────┘

↓

Reactive Data Changes

↓

┌─────────────────┐

│ beforeUpdate │

└──────┬──────────┘

↓

┌────────────┐

│ updated │

└─────┬──────┘

↓

┌──────────────┐

│ beforeUnmount│

└──────┬───────┘

↓

┌───────────┐

│ unmounted │

└───────────┘

**Error Handling Lifecycle Hook**

**Options API:**

js

errorCaptured(err, instance, info) {

console.error('Error from child:', err);

return false; // prevents further propagation

}

**Composition API:**

ts

import { onErrorCaptured } from 'vue';

onErrorCaptured((err, instance, info) => {

console.error('Error caught:', err);

return false; // prevent further propagation

});

**Best Practices**

* Use mounted() for API calls or DOM access.
* Clean up resources in beforeUnmount() or unmounted() (like setInterval, event listeners).
* Avoid heavy logic in created(); prefer mounted() for side effects.
* updated() can be performance-intensive — use it carefully.

**Use Case Examples**

**1. Fetching Data After Mount:**

ts

onMounted(() => {

fetch('https://api.example.com/data')

.then(res => res.json())

.then(data => console.log(data));

});

**2. Cleanup Example:**

ts

let intervalId: number;

onMounted(() => {

intervalId = setInterval(() => console.log('tick'), 1000);

});

onUnmounted(() => {

clearInterval(intervalId);

});

**1. beforeCreate *(Options API only)***

* **When:** Called **immediately after the instance is initialized**, **before data, methods, computed, and watch are set up**.
* **Use Case:** Rarely used — mostly for debugging or very early configuration.
* **Limitation:** this is not yet reactive, and you **cannot access data or props**.

js

beforeCreate() {

console.log('beforeCreate', this.someData); // undefined

}

**2. created *(Options API only)***

* **When:** Called after the component instance is **created** and reactive properties are **set up**, but **before DOM mounting**.
* **Use Case:** Good for:
  + Initializing data
  + Fetching data (can also be done in mounted)
  + Setting up watchers

js

created() {

console.log('created', this.someData); // accessible

}

**3. beforeMount**

* **When:** Called **right before the DOM is mounted** (rendered for the first time).
* **Use Case:** Rarely used, but sometimes helpful to prepare something right before mounting.

js

beforeMount() {

console.log('DOM is about to be mounted');

}

**Composition API equivalent:**

ts

onBeforeMount(() => {

console.log('DOM is about to be mounted');

});

**4. mounted**

* **When:** Called after the component has been **mounted into the DOM**.
* **Use Case:** Commonly used for:
  + Accessing the DOM (e.g., document.querySelector)
  + Third-party libraries (e.g., chart.js, jQuery plugins)
  + Fetching data (also possible here)

js

mounted() {

console.log('DOM is ready!');

}

**Composition API equivalent:**

ts

onMounted(() => {

console.log('DOM is ready!');

});

**5. beforeUpdate**

* **When:** Called **before the DOM is patched** due to a **reactive data change**.
* **Use Case:** Allows access to the current DOM **before changes are applied**.
* Can be used to compare previous and new states or save data.

js

beforeUpdate() {

console.log('DOM will update soon');

}

**Composition API equivalent:**

ts

onBeforeUpdate(() => {

console.log('DOM will update soon');

});

**6. updated**

* **When:** Called **after the DOM is re-rendered** due to reactive changes.
* **Use Case:** Access the updated DOM (but use with care — can be performance intensive).

js

updated() {

console.log('DOM updated');

}

**Composition API equivalent:**

ts

onUpdated(() => {

console.log('DOM updated');

});

**7. beforeUnmount *(was beforeDestroy in Vue 2)***

* **When:** Called right **before the component is unmounted** and removed from the DOM.
* **Use Case:** Cleanup work:
  + Clear intervals
  + Remove event listeners
  + Abort HTTP requests

js

beforeUnmount() {

console.log('Cleanup before unmount');

}

**Composition API equivalent:**

ts

onBeforeUnmount(() => {

console.log('Cleanup before unmount');

});

**8. unmounted *(was destroyed in Vue 2)***

* **When:** Called **after the component is removed** from the DOM.
* **Use Case:** Final cleanup or reporting/logging.

js

unmounted() {

console.log('Component removed');

}

**Composition API equivalent:**

ts

onUnmounted(() => {

console.log('Component removed');

});

**9. errorCaptured / onErrorCaptured**

* **When:** Called when an **error occurs in a child component**.
* **Use Case:** Gracefully catch and handle rendering or lifecycle errors.
* **Return false** to **prevent further bubbling**.

js

errorCaptured(err, instance, info) {

console.error('Error captured:', err);

return false; // stops propagation

}

**Composition API equivalent:**

ts

onErrorCaptured((err, instance, info) => {

console.error('Error captured:', err);

return false;

});

**Summary Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Hook Name** | **Phase** | **Composition API** | **When It Runs** |
| beforeCreate | Creation | ❌ Not available | Before data/init |
| created | Creation | ❌ Not available | After setup but before DOM |
| beforeMount | Mounting | onBeforeMount | Before first DOM render |
| mounted | Mounting | onMounted | After first DOM render |
| beforeUpdate | Updating | onBeforeUpdate | Before re-render |
| updated | Updating | onUpdated | After re-render |
| beforeUnmount | Unmounting | onBeforeUnmount | Before component removal |
| unmounted | Unmounting | onUnmounted | After component is removed |
| errorCaptured | Error | onErrorCaptured | When a child throws an error |