

**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")

vue

<p v-if="loggedIn">Welcome back!</p>

<a v-bind:href="profileLink">Profile</a>

**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>

Changing name later won’t affect the rendered text.

**Custom Directive (Advanced)**

vue

<script>

export default {

directives: {

focus: {

mounted(el) {

el.focus();

}

}

}

}

</script>

<template>

<input v-focus />

</template>

# 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

CopyEdit

defineProps<{

msg: string;

count?: number;

}>();

* Prevents passing wrong data types to components.

**Typed Emits**

ts

CopyEdit

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

CopyEdit

<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

CopyEdit

<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

CopyEdit

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

CopyEdit

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

CopyEdit

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

CopyEdit

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

But you also get:

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

**Example with explanation**

ts

CopyEdit

const newName = ref('');

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

ts

CopyEdit

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

* 🔄 Reactive array
* 📋 Tracks all entered names

ts

CopyEdit

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

CopyEdit

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

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

html

CopyEdit

<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

CopyEdit

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

CopyEdit

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

CopyEdit

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

**✅ 2. Using defineComponent**

ts

CopyEdit

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

CopyEdit

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

CopyEdit

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

CopyEdit

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

CopyEdit

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

CopyEdit

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

CopyEdit

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

CopyEdit

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

CopyEdit

<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

CopyEdit

export default {

provide() {

return {

keyName: value,

}

}

}

#### Composition API / <script setup>

ts

CopyEdit

import { provide, ref } from 'vue'

const message = ref('Hello from parent')

provide('keyName', message)

### inject (in child):

#### Options API

js

CopyEdit

export default {

inject: ['keyName']

}

#### Composition API / <script setup>

ts

CopyEdit

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

CopyEdit

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

CopyEdit

<script setup>

import { provide, ref } from 'vue'

const theme = ref('dark')

provide('theme', theme)

</script>

<template>

<ChildComponent />

</template>

### Child (deeply nested)

ts

CopyEdit

<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

CopyEdit

<template>

<div class="card">

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

</div>

</template>

**Parent Component**

vue

CopyEdit

<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

CopyEdit

<template>

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

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

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

</template>

**Parent Component**

vue

CopyEdit

<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

CopyEdit

<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

CopyEdit

<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

CopyEdit

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

CopyEdit

<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

CopyEdit

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

CopyEdit

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

CopyEdit

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

**🧠 7. Programmatic Validation & Reset**

ts

CopyEdit

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

// Validate manually

const result = await validate()

// Reset form

resetForm()

**🔁 8. Validate on Blur or Input**

ts

CopyEdit

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

validateOnValueUpdate: true, // validate on input

validateOnMount: true, // validate immediately

})

**💡 9. Handling Nested Forms / Arrays**

ts

CopyEdit

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

CopyEdit

<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

CopyEdit

<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

CopyEdit

<template>

<Form @submit="onSubmit">

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

<ErrorMessage name="email" />

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

</Form>

</template>

ts

CopyEdit

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

CopyEdit

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

CopyEdit

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

CopyEdit

<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

CopyEdit

// 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.

local registation and examples

import HelloWorld from "./helloword.vue"; // local registration

<template>

<helloworld :msg="'hello worlds app.vue'"/>//passing props

<span> hello world</span>

</template>

<style scoped>

span{

color: green

}

</style>  
glocal registration  
main.ts

import { createApp } from 'vue'

import App from './App.vue'

app.component("helloword",HelloWolrd")//global registration

createApp(App).mount('#app')

.vue

<template>

<helloword></helloword>

</template>

directrives all

<template>

<div>

<!-- v-bind -->

<img :src="image" alt="Vue Logo" />

<!-- v-model -->

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

<!-- v-if / v-else -->

<p v-if="username">Welcome, {{ username }}!</p>

<p v-else>Please enter your name.</p>

<!-- v-show -->

<p v-show="username.length > 3">Name is long enough</p>

<!-- v-for -->

<ul>

<li v-for="(item, i) in hobbies" :key="i">{{ item }}</li>

</ul>

<!-- v-on -->

<button @click="greet">Click Me</button>

<!-- v-html -->

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

<!-- v-text -->

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

<!-- v-once -->

<p v-once>This will not change: {{ now }}</p>

</div>

</template>

<script setup>

import { ref } from 'vue';

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

const username = ref('');

const hobbies = ['Coding', 'Music', 'Books'];

const htmlContent = '<em>This is italic text.</em>';

const plainText = 'Plain text example';

const now = new Date().toLocaleString();

function greet() {

alert(`Hello, ${username.value || 'Guest'}!`);

}

</script>